Оптимизация содержимого для платформы ADOBE® FLASH® PLATFORM

Юридическая информация

Последнее обновление 11.5.2012.

Содержание

Глава 1. Введение

Приложения Adobe® AIR® и Adobe® Flash® Player работают на различных платформах, включая настольные системы, мобильные устройства, планшетные ПК и телевизионные устройства. В этом документе рассматриваются конкретные задачи и фрагменты кода, которые познакомят читателя с основными принципами разработки приложений. Охватываются следующие темы:

- Экономия памяти
- Уменьшение загрузки ЦП
- Улучшение производительности ActionScript 3.0
- Увеличение скорости визуализации
- Оптимизация сетевого взаимодействия
- Работа с аудио и видео
- Оптимизация производительности базы данных SQL
- Тестирование и развертывание приложений

Большинство из этих методов оптимизации применимы к приложениям на всех устройствах, к среде выполнения AIR и Flash Player. Кроме того, рассматриваются исключения для отдельных устройств.

Некоторые из методов оптимизации основаны на возможностях, реализованных в Flash Player 10.1 и AIR 2.5. Однако многие методы оптимизации также работают в более ранних версиях AIR и Flash Player.

Основные сведения по выполнению кода в среде выполнения

Одним из ключевых моментов, необходимых для проведения оптимизации приложения, является понимание того, как среда выполнения платформы Flash Platform выполняет код. Среда выполнения работает циклично, выполняя определенные действия в каждом «кадре». Кадр представляет собой отрезок времени, который определяется частотой кадров для данного приложения. Частоту кадров определяет период времени, выделенный для отображения каждого кадра. Например, при частоте 30 кадров в секунду среда выполнения постарается отобразить каждый кадр в течение одной тридцатой доли секунды.

Начальная частота кадров для приложения задается во время разработки. Задать частоту кадров можно с помощью параметров Adobe® Flash® Builder™ или Flash Professional. Также можно указать начальную частоту кадров в коде. Задать частоту кадров в приложении, использующем только ActionScript, можно, добавив тег метаданных `[SWF(frameRate="24")]` в класс корневого документа. В MXML используйте для тега Application или WindowedApplication атрибут `frameRate`.

Каждый цикл кадра состоит из двух фаз, разделенных на три части: события, событие `enterFrame` и визуализация.

Первая фаза состоит из двух частей (события и событие `enterFrame`), каждое из которых потенциально приводит к вызову кода. В первой части первой фазы поступают и отправляются события среды выполнения. Этими событиями могут быть завершение или ход выполнения асинхронных операций, например при загрузке данных по сети. В число этих событий также входит обработка команд пользователя. После отправки событий среда выполнения выполняет код в зарегистрированных прослушивателях. Если события отсутствуют, среда выполнения ожидает завершения фазы выполнения, не выполняя никаких действий. При отсутствии выполняемых действий среда выполнения никогда не увеличивает частоту кадров. Если событие происходит в других частях цикла выполнения, среда выполнения помещает их в очередь и отправляет в следующем кадре.

Второй частью первой фазы является событие `enterFrame`. Это событие отличается от остальных тем, что оно отправляется только один раз за кадр.

После отправки всех событий начинается фаза визуализации. На этом этапе среда выполнения вычисляет состояние всех видимых элементов и рисует их на экране. Затем этот процесс повторяется (подобно бегу спортсмена вокруг стадиона).

Примечание. *Для событий, включающих свойство* `updateAfterEvent`*, можно выполнить немедленную визуализацию, не дожидаясь этапа визуализации. Однако избегайте использования свойства* `updateAfterEvent`*, если из-за него часто возникают проблемы с производительностью.*

Проще всего представить, что цикл кадра состоит из двух фаз, равных по продолжительности. В этом случае половина каждого цикла кадра отводится для выполнения обработчиков событий и кода приложения, а половина — для визуализации. Однако на практике зачастую все выглядит иначе. Иногда для выполнения кода приложения требуется больше половины времени кадра, что приводит к увеличению первой части фазы и уменьшению второй части фазы, предназначенной для визуализации. И наоборот, при наличии сложного визуального содержимого, например фильтров и режимов наложения, для выполнения визуализации может потребоваться больше половины времени кадра. Так как фактическая длительность фаз может изменяться, то можно сказать, что цикл кадра «эластичен».

Если для выполнения обеих фаз (выполнение кода и визуализация) одного цикла кадра не хватает, среда выполнения изменяет частоту кадров. Длительность кадра увеличивается, поэтому возникает задержка перехода к следующему кадру. Например, если продолжительность цикла кадра составляет более одной тридцатой секунды, среда выполнения не поддерживает обновление экрана с частотой 30 кадров в секунду. Замедление частоты кадров ощущается в падении производительности. В лучшем случае анимация не будет воспроизводится плавно. В худшем случае приложение зависает, а изображение пропадает.

Дополнительные сведения о выполнении кода средой выполнения на платформе Flash Platform и модели визуализации см. в следующих ресурсах.

- Умозрительная модель проигрывателя Flash Player — Эластичный гоночный трек (автор Тэд Патрик (Ted Patrick))
- Асинхронное выполнение ActionScript (автор Тревор МакКоли (Trevor McCauley))
- Оптимизация выполнения кода, памяти и визуализации в Adobe AIR: http://www.adobe.com/go/learn_fp_air_perf_tv_ru (Видео презентации конференции MAX, проведенной Шоном Кристманном (Sean Christmann))

Ощущаемая и фактическая производительность

В конечном счете оценивают производительность вашего приложения его пользователи. Разработчики могут измерить производительность приложения по времени выполнения определенных операций или по количеству создаваемых объектов. Однако для конечных пользователей эти критерии не имеют никакого значения. Иногда для оценки производительности пользователи используют другие критерии. Например, насколько быстро и плавно приложение работает и реагирует на ввод данных? Приводит ли его использование к снижению производительности системы? Чтобы оценить производительность приложения с позиции пользователя, задайте себе следующие вопросы.

- Воспроизводится ли анимация плавно или нет?
- Воспроизводится ли видеосодержимое плавно или нет?
- Воспроизводятся ли аудиофрагменты непрерывно или нет?
- Мерцает ли или исчезает ли изображение во время длительного выполнения операций?
- Отображается ли введенный текст сразу же или с задержкой?
- Выполняется ли соответствующее действие сразу после нажатия кнопки или с задержкой?
- Увеличивается ли шум вентилятора процессора во время работы приложения?
- Быстро ли разряжается аккумулятор портативного компьютера или мобильного устройства во время работы с приложением?
- Сказывается ли работа приложения на реакцию на действия пользователя других приложений?

Важно понимать разницу между ощущаемой и фактической производительностью приложения. Способ достижения наилучшей ощущаемой производительности не всегда совпадает со способом достижения наилучшей абсолютной производительности. Убедитесь, что приложению не требуется выполнять большие объемы кода, из-за чего среда выполнения может не успевать с заданной частотой обновлять экран и быстро реагировать на ввод данных пользователем. В некоторых случаях достичь хороших результатов можно путем разделения выполняемой приложением задачи на несколько частей, между которыми среда выполнения успевала бы обновлять экран. (Подробные инструкции см. в разделе ««Производительность визуализации» на странице 50».)

Ниже описаны способы оптимизации фактической производительности выполнения кода и ощущаемой пользователями производительности.

Выбор области оптимизации

Не все действия по оптимизации производительности дают ощутимый для пользователя результат. Важно выполнять оптимизацию производительности именно в тех областях, которые являются проблемными для конкретного приложения. Некоторые операции по оптимизации производительности являются общими лучшими практическими приемами, к которым можно прибегать всегда. Степень эффективности других операций по оптимизации зависит от направленности приложения и предполагаемой целевой аудитории. Например, производительность приложения всегда выше, если в нем не используются анимации, видео или графические фильтры и эффекты. Однако одним из преимуществ использования платформы Flash Platform является то, что данная платформа предоставляет разработчику обширные возможности по работе с мультимедиа, что позволяет создавать красочные и яркие приложения. Оцените, соответствует ли предпочитаемый набор функциональных возможностей характеристикам производительности компьютеров и устройств, в которых выполняется приложение.

Общий совет: не проводите оптимизацию на ранних этапах разработки. Некоторые способы оптимизации преобразуют код в более сложный для прочтения вид или делают его менее гибким. После оптимизации работать с таким кодом становится сложно. При использовании этих способов оптимизации лучше немного подождать и проанализировать эффективность выполнения конкретной части кода перед тем, как принять решение о его оптимизации.

Иногда при оптимизации производительности необходимо отыскать «золотую середину». В идеале, уменьшение объема памяти, используемой приложением, также увеличивает скорость, с которой приложение выполняет задачу. Однако такие идеальные возможности для оптимизации существуют не всегда. Например, если приложение зависает при выполнении операции, то чаще всего проблему можно устранить разделением задачи на несколько кадров. Поскольку работа разделяется, вероятно, в целом выполнение процесса займет больше времени. Однако это замедление может быть незаметным для пользователя, если приложение продолжает реагировать на ввод данных и не зависает.

Для определения области оптимизаций и их эффекта лучше всего провести проверку производительности. Некоторые способы проверки производительности, а также полезные советы см. в разделе ««Тестирование и развертывание» на странице 101».

Дополнительные сведения об определении мест в приложении, нуждающихся в оптимизации, см. в следующих ресурсах.

- Приложения по настройке производительности для AIR: http://www.adobe.com/go/learn_fp_goldman_tv_ru (Видео презентации конференции MAX, проведенной Оливером Голдманом (Oliver Goldman))
- Приложения по настройке производительности Adobe AIR: http://www.adobe.com/go/learn_fp_air_perf_devnet_ru (Статья Adobe Developer Connection Оливера Голдмана (Oliver Goldman), основанная на презентации)

Глава 2. Экономия памяти

Экономия памяти всегда играет важную роль при разработке приложений, даже при разработке приложений для настольных систем. Однако в мобильных устройствах использование памяти является исключительно важным, поэтому имеет смысл ограничить объем памяти, используемый приложением.

Экранные объекты

Выберите соответствующий экранный объект.

ActionScript 3.0 включает широкий набор экранных объектов. Одним из самых простых советов по оптимизации для ограничения потребления памяти является использование подходящего типа экранного объекта. Для простых неинтерактивных фигур используйте объекты Shape. Для интерактивных объектов, для которых не требуется временная шкала, используйте объекты Sprite. Для анимации с применением временной шкалы используйте объекты MovieClip. В приложении всегда выбирайте наиболее эффективный тип объекта.

Следующий код показывает использование памяти для разных экранных объектов.

```
trace(getSize(new Shape()));
// output: 236

trace(getSize(new Sprite()));
// output: 412

trace(getSize(new MovieClip()));
// output: 440
```

Метод `getSize()` показывает объем памяти, используемой объектом, в байтах. Как видим, использование нескольких объектов MovieClip вместо простых объектов Shape является напрасной тратой ресурсов памяти; при этом возможности объекта MovieClip все равно не используются.

Примитивные типы

Используйте метод `getSize()` для тестирования кода и определения наиболее эффективного объекта для выполнения задачи.

Все примитивные типы, за исключением типа String, используют от 4 до 8 байт памяти. Бесполезно оптимизировать память за счет использования определенного типа примитива.

```
// Primitive types
var a:Number;
trace(getSize(a));
// output: 8

var b:int;
trace(getSize(b));
// output: 4

var c:uint;
trace(getSize(c));
// output: 4

var d:Boolean;
trace(getSize(d));
// output: 4

var e:String;
trace(getSize(e));
// output: 4
```

Для типа Number, который представляет 64-битное значение, виртуальная машина ActionScript Virtual Machine (AVM) выделяет 8 байт, если ему не присвоено значение. Для хранения всех остальных примитивных типов выделяются 4 байта.

```
// Primitive types
var a:Number = 8;
trace(getSize(a));
// output: 4

a = Number.MAX_VALUE;
trace(getSize(a));
// output: 8
```

Поведение отличается для типа String. Объем выделенной для хранения памяти зависит от длины строки String:

```
var name:String;
trace(getSize(name));
// output: 4

name = "";
trace(getSize(name));
// output: 24

name = "Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum
has been the industry's standard dummy text ever since the 1500s, when an unknown printer took
a galley of type and scrambled it to make a type specimen book. It has survived not only five
centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It
was popularized in the 1960s with the release of Letraset sheets containing Lorem Ipsum
passages, and more recently with desktop publishing software like Aldus PageMaker including
versions of Lorem Ipsum.";
trace(getSize(name));
// output: 1172
```

Используйте метод `getSize()` для тестирования кода и определения наиболее эффективного объекта для выполнения задачи.

Повторное использование объектов

По возможности используйте объекты повторно вместо их повторного создания.

Другим простым способом оптимизации памяти является повторное использование объектов и по возможности предотвращение их повторного создания. Например, в цикле не используйте следующий код:

```
const MAX_NUM:int = 18;
const COLOR:uint = 0xCCCCCC;

var area:Rectangle;

for (var:int = 0; i < MAX_NUM; i++)
{
    // Do not use the following code
    area = new Rectangle(i,0,1,10);
    myBitmapData.fillRect(area,COLOR);
}
```

Воссоздание объекта Rectangle в каждом цикле итерации требует больше ресурсов памяти и выполняется медленнее, так как каждый раз приходится снова создавать объект. Используйте следующий подход:

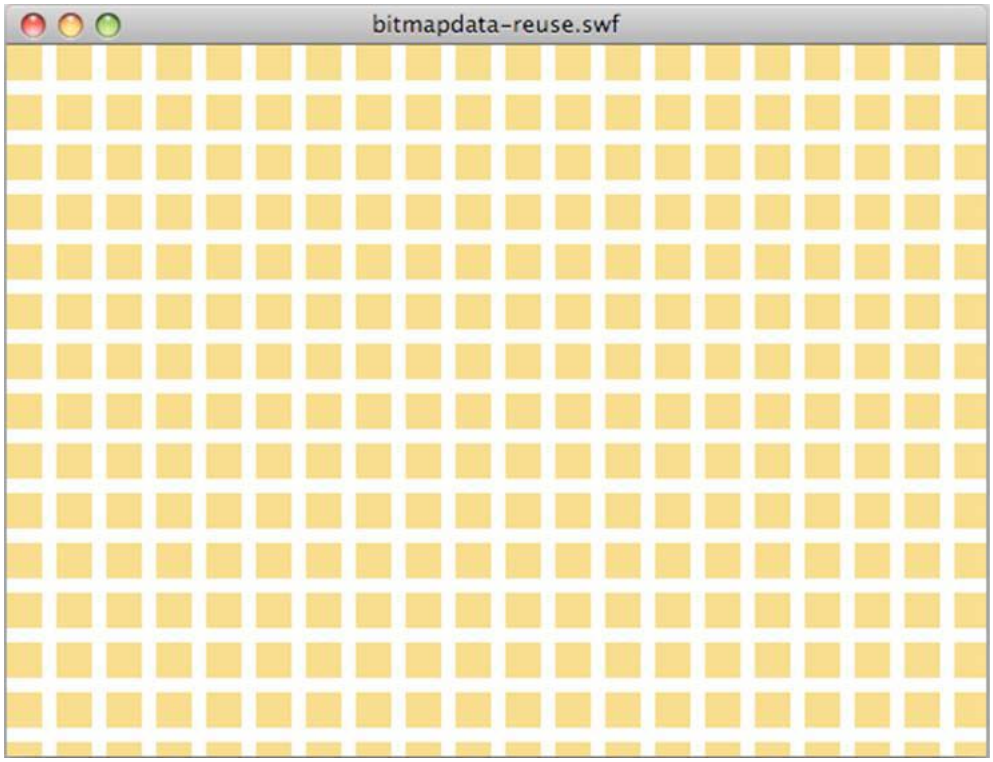
```
const MAX_NUM:int = 18;
const COLOR:uint = 0xCCCCCC;

// Create the rectangle outside the loop
var area:Rectangle = new Rectangle(0,0,1,10);

for (var:int = 0; i < MAX_NUM; i++)
{
    area.x = i;
    myBitmapData.fillRect(area,COLOR);
}
```

В предыдущем примере объект использовался с относительно небольшим потреблением ресурсов памяти. Следующий пример демонстрирует еще большую экономию памяти за счет повторного использования объекта BitmapData. Следующий код для создания эффекта плитки неэффективно использует память.

```
var myImage:BitmapData;
var myContainer:Bitmap;
const MAX_NUM:int = 300;

for (var i:int = 0; i< MAX_NUM; i++)
{
    // Create a 20 x 20 pixel bitmap, non-transparent
    myImage = new BitmapData(20,20,false,0xF0D062);

    // Create a container for each BitmapData instance
    myContainer = new Bitmap(myImage);

    // Add it to the display list
    addChild(myContainer);

    // Place each container
    myContainer.x = (myContainer.width + 8) * Math.round(i % 20);
    myContainer.y = (myContainer.height + 8) * int(i / 20);
}
```

Примечание. *При использовании положительных значений преобразование округленного значения в тип int выполняется гораздо быстрее, чем при использовании метода* `Math.floor()`.

На следующем рисунке показан результат разбиения растрового изображения на плитки.



Результат разбиения растрового изображения на плитки

В оптимизированной версии создается один экземпляр BitmapData, на который ссылаются несколько экземпляров Bitmap. Результат получается таким же.

```
// Create a single 20 x 20 pixel bitmap, non-transparent
var myImage:BitmapData = new BitmapData(20,20,false,0xF0D062);
var myContainer:Bitmap;
const MAX_NUM:int = 300;

for (var i:int = 0; i< MAX_NUM; i++)
{
    // Create a container referencing the BitmapData instance
    myContainer = new Bitmap(myImage);

    // Add it to the display list
    addChild(myContainer);

    // Place each container
    myContainer.x = (myContainer.width + 8) * Math.round(i % 20);
    myContainer.y = (myContainer.height + 8) * int(i / 20);
}
```

Этот подход позволяет сэкономить около 700 КБ памяти, что является значительным показателем в традиционных мобильных устройствах. Каждым контейнером растрового изображения можно управлять без изменения исходного экземпляра BitmapData с использованием свойств Bitmap.

```
// Create a single 20 x 20 pixel bitmap, non-transparent
var myImage:BitmapData = new BitmapData(20,20,false,0xF0D062);
var myContainer:Bitmap;
const MAX_NUM:int = 300;

for (var i:int = 0; i< MAX_NUM; i++)
{
    // Create a container referencing the BitmapData instance
    myContainer = new Bitmap(myImage);

    // Add it to the DisplayList
    addChild(myContainer);

    // Place each container
    myContainer.x = (myContainer.width + 8) * Math.round(i % 20);
    myContainer.y = (myContainer.height + 8) * int(i / 20);

    // Set a specific rotation, alpha, and depth
    myContainer.rotation = Math.random()*360;
    myContainer.alpha = Math.random();
    myContainer.scaleX = myContainer.scaleY = Math.random();
}
```

На следующем рисунке показан результат преобразований растрового изображения.

Результат преобразований растрового изображения

Дополнительные разделы справки

«Кэширование растрового изображения» на странице 57

Создание пула объектов

По возможности прибегайте к созданию пула объектов.

Другой важный прием по оптимизации называется созданием пула объектов и включает повторное использование объектов со временем. При инициализации приложения создается заданное число объектов, которые сохраняются в пуле, таких как Array или Vector. По завершении работы с объектом он деактивируется и не использует ресурсы центрального процессора, а все взаимные ссылки удаляются. Однако не следует задавать этим ссылкам значение `null`, поскольку в этом случае они могут быть удалены при сборке мусора. Следует просто поместить объект обратно в пул и извлечь его, когда потребуется новый объект.

При повторном использовании объектов их не требуется создавать заново, поэтому ресурсы экономятся. Оно также уменьшает вероятность запуска функции сборки мусора, которая может замедлять работу приложения. Следующий код демонстрирует метод создания пула объектов.

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

    public final class SpritePool
    {
        private static var MAX_VALUE:uint;
        private static var GROWTH_VALUE:uint;
        private static var counter:uint;
        private static var pool:Vector.<Sprite>;
        private static var currentSprite:Sprite;

        public static function initialize( maxPoolSize:uint, growthValue:uint ):void
        {
            MAX_VALUE = maxPoolSize;
            GROWTH_VALUE = growthValue;
            counter = maxPoolSize;

            var i:uint = maxPoolSize;

            pool = new Vector.<Sprite>(MAX_VALUE);
            while( --i > -1 )
                pool[i] = new Sprite();
        }

        public static function getSprite():Sprite
        {
            if ( counter > 0 )
                return currentSprite = pool[--counter];

            var i:uint = GROWTH_VALUE;
            while( --i > -1 )
                    pool.unshift ( new Sprite() );
            counter = GROWTH_VALUE;
            return getSprite();

        }

        public static function disposeSprite(disposedSprite:Sprite):void
        {
            pool[counter++] = disposedSprite;
        }
    }
}
```

Класс SpritePool создает пул новых объектов при инициализации приложения. Метод `getSprite()` возвращает экземпляры этих объектов, а метод `disposeSprite()` освобождает их. Код обеспечивает увеличение пула при его полном заполнении. Кроме того, можно создать пул фиксированного размера. В результате при полном заполнении пула новые объекты выделяться не будут. По возможности избегайте создания новых объектов в циклах. Дополнительные сведения см. в разделе ««Освобождение памяти» на странице 12». В следующем примере кода класс SpritePool используется для извлечения новых экземпляров.

```
const MAX_SPRITES:uint = 100;
const GROWTH_VALUE:uint = MAX_SPRITES >> 1;
const MAX_NUM:uint = 10;

SpritePool.initialize ( MAX_SPRITES,  GROWTH_VALUE );

var currentSprite:Sprite;
var container:Sprite = SpritePool.getSprite();

addChild ( container );

for ( var i:int = 0; i< MAX_NUM; i++ )
{
    for ( var j:int = 0; j< MAX_NUM; j++ )
    {
        currentSprite = SpritePool.getSprite();
        currentSprite.graphics.beginFill ( 0x990000 );
        currentSprite.graphics.drawCircle ( 10, 10, 10 );
        currentSprite.x = j * (currentSprite.width + 5);
        currentSprite.y = i * (currentSprite.width + 5);
        container.addChild ( currentSprite );
    }
}
```

В следующем примере кода все экранные объекты удаляются из списка отображения при щелчке мышью, а затем повторно используются для выполнения другой задачи.

```
stage.addEventListener ( MouseEvent.CLICK, removeDots );

function removeDots ( e:MouseEvent ):void
{
    while (container.numChildren > 0 )
        SpritePool.disposeSprite (container.removeChildAt(0) as Sprite );
}
```

Примечание. *Вектор пула всегда ссылается на объекты Sprite. Если необходимо полностью удалить объект из памяти, воспользуйтесь методом* `dispose()` *класса SpritePool, который удаляет все остальные ссылки.*

Освобождение памяти

Удалите все ссылки на объекты для запуска функции сборки мусора.

Сборщик мусора нельзя запустить в окончательной версии Flash Player. Чтобы при удалении объекта он удалялся при сборке мусора, удалите все ссылки на объект. Помните, что старый оператор `delete`, знакомый по версиям ActionScript 1.0 и 2.0, ведет себя иначе в ActionScript 3.0. Его можно использовать только для удаления динамических свойств динамического объекта.

Примечание. *Сборщик мусора можно вызвать в среде Adobe® AIR® и отладочной версии Flash Player.*

Например, следующий код задает для ссылки Sprite значение `null`:

```
var mySprite:Sprite = new Sprite();

// Set the reference to null, so that the garbage collector removes
// it from memory
mySprite = null;
```

Помните, что при присвоении объекту значения `null` он необязательно будет удален из памяти. Иногда функция сборки мусора не выполняется, если доступный объем памяти не считается достаточно низким. Поведение функции сборки мусора не является предсказуемым. Сборщик мусора вызывается в результате выделения памяти, а не при удалении объекта. В процессе выполнения сборщик мусора находит структуры объектов, которые еще не были собраны. Он обнаруживает неактивные объекты в структурах путем нахождения ссылающихся друг на друга объектов, которые больше не используются приложением. Обнаруженные таким образом неактивные объекты удаляются.

В больших приложениях этот процесс может интенсивно использовать ресурсы центрального процессора, снижать производительность и заметно замедлять работу приложения. Попробуйте ограничить число проходов по сборке мусора за счет максимально возможного повторного использования объектов. Кроме того, по возможности присваивайте ссылкам значение null, чтобы нахождение объектов при сборке мусора выполнялось быстрее. Сборку мусора следует рассматривать в качестве страховки. По возможности всегда явно управляйте сроком службы объектов.

Примечание. *Задание для ссылки на экранный объект значения null не обеспечивает замораживание объекта. Объект и далее создает нагрузку на ЦП, пока не будет удален при сборке мусора. Деактивируйте объект, прежде чем задавать ссылке на него значение* `null`.

Сборщик мусора можно запустить с помощью метода `System.gc()`, доступного в среде Adobe AIR и в отладочной версии Flash Player. Профилировщик, входящий в состав пакета Adobe® Flash® Builder™, позволяет вручную запустить сборщик мусора. Выполнение сборщика мусора позволяет проследить за реакцией приложения и правильностью удаления объектов из памяти.

Примечание. *Если объект использовался в качестве прослушивателя событий, на него может ссылаться другой объект. В таком случае удалите прослушиватели событий с помощью метода* `removeEventListener()`*, прежде чем задавать для ссылок значение* `null`.

К счастью, объем памяти, направленной на обработку растровых изображений, можно сразу же уменьшить. Например, класс BitmapData включает метод `dispose()`. В следующем примере создается экземпляр BitmapData размером 1,8 МБ. Объем текущей используемой памяти увеличивается до 1,8 МБ, и свойство `System.totalMemory` возвращает меньшее значение.

```
trace(System.totalMemory / 1024);
// output: 43100

// Create a BitmapData instance
var image:BitmapData = new BitmapData(800, 600);

trace(System.totalMemory / 1024);
// output: 44964
```

После этого объект BitmapData вручную удаляется (освобождается) из памяти и выполняется повторная проверка используемой памяти.

```
trace(System.totalMemory / 1024);
// output: 43100

// Create a BitmapData instance
var image:BitmapData = new BitmapData(800, 600);

trace(System.totalMemory / 1024);
// output: 44964

image.dispose();
image = null;

trace(System.totalMemory / 1024);
// output: 43084
```

И хотя метод `dispose()` удаляет пикселы из памяти, для ссылки по-прежнему необходимо задать значение `null`, чтобы полностью освободить его. Если объект BitmapData больше не требуется, всегда вызывайте метод `dispose()` и задавайте для ссылки значение `null` для немедленного освобождения памяти.

Примечание. *В проигрывателе Flash Player 10.1 и в среде AIR 1.5.2 представлен новый метод* `disposeXML()` *класса System. С его помощью можно сразу сделать XML доступным для сборки мусора, передав в качестве параметра дерево XML.*

Дополнительные разделы справки

«Замораживание и размораживание объектов» на странице 30

Использование растровых изображений

Использование векторных изображений вместо растровых также экономит память. Однако использование векторов, особенно в больших количествах, значительно увеличивает потребность в ресурсах центрального и графического процессоров. Использование растровых изображений служит хорошим способом оптимизации визуализации, поскольку среде выполнения требуется меньше процессорных ресурсов для рисования пикселов, чем для визуализации содержимого вектора.

Дополнительные разделы справки

«Кэширование растрового изображения вручную» на странице 64

Субдискретизация растровых изображений

Для более эффективного использования памяти, когда Flash Player обнаруживает 16-битный экран, разрешение 32-битных непрозрачных изображений уменьшается до 16 бит. Такая субдискретизация позволяет в два раза уменьшить использование ресурсов памяти и обеспечивает более быструю визуализацию изображения. Эта возможность доступна только в проигрывателе Flash Player 10.1 для Windows Mobile.

Примечание. *В версиях, предшествующих Flash Player 10.1, все пикселы, созданные в памяти, хранились с использованием 32 бит (4 байт). Простой логотип 300 x 300 пикселов занимал в памяти 350 КБ (300*300*4/1024). Благодаря новому поведению такой же непрозрачный логотип занимает всего 175 КБ. Если логотип является прозрачным, его разрешение не уменьшается до 16 бит и он занимает такой же объем памяти. Эта функция применима только к встроенным растровым изображениям и изображениям, загруженным во время выполнения, в форматах PNG, GIF и JPG.*

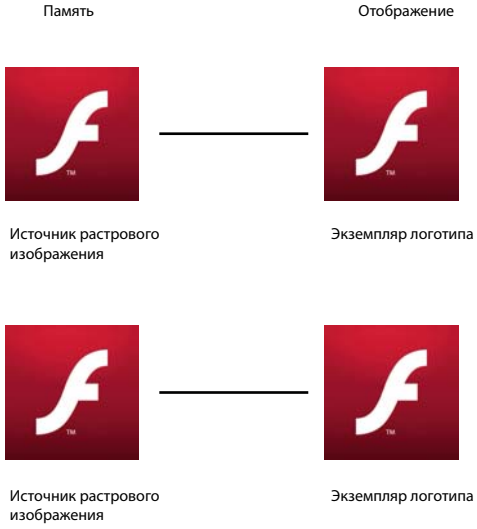
В мобильных устройствах сложно отличить изображение, визуализированное с использованием 16 бит, от 32-битной версии этого изображения. Для простого изображения, содержащего всего несколько цветов, разница не будет ощутима. Даже для более сложного изображения трудно заметить различия. Однако при увеличении изображения может происходить незначительное ухудшение цвета и 16-битный градиент может выглядеть менее плавным, чем в 32-битной версии.

Единая ссылка BitmapData

Эффективный способ оптимизации — повторное использование экземпляров класса BitmapData. В проигрывателях Flash Player 10.1 и AIR 2.5 представлена новая функция единой ссылки BitmapData, предназначенная для всех платформ. При создании экземпляров BitmapData на основе встроенного изображения для всех экземпляров BitmapData используется единое растровое изображение. При последующем изменении растрового изображения ему назначается собственное уникальное растровое изображение в памяти. Встроенное изображение может извлекаться из библиотеки или тегов [Embed].

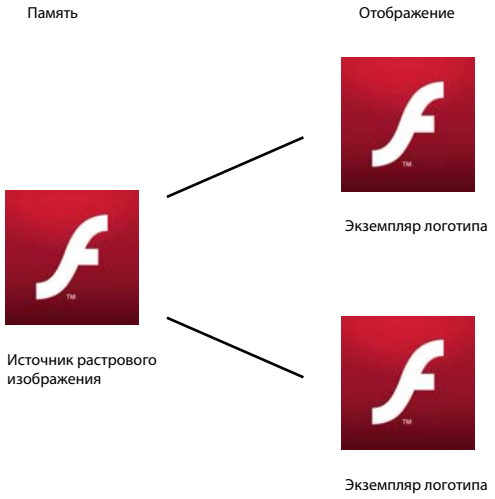
Примечание. *Существующее содержимое также эффективно использует эту новую функцию, поскольку Flash Player 10.1 и AIR 2.5 автоматически повторно используют растровые изображения.*

При создании экземпляра встроенного изображения в памяти создается связанное растровое изображение. В версиях, предшествующих Flash Player 10.1 и AIR 2.5, каждому экземпляру в памяти назначалось отдельное растровое изображение, как показано на следующем рисунке.



Загрузка растровых изображений в память до загрузки Flash Player 10.1 и AIR 2.5

В проигрывателях Flash Player 10.1 и AIR 2.5 при создании нескольких экземпляров одного изображения для всех экземпляров BitmapData используется единая версия растрового изображения. Следующий рисунок демонстрирует этот подход.

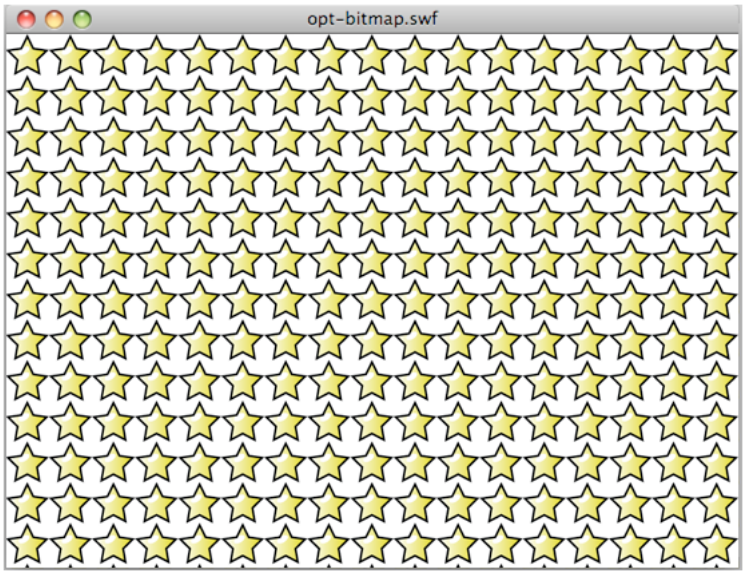


Растровые изображения в памяти в проигрывателе Flash Player 10.1 и AIR 2.5

Этот подход позволяет значительно уменьшить объем памяти, используемый приложением с большим числом растровых изображений. Следующий код создает много экземпляров символа `Star`:

```
const MAX_NUM:int = 18;

var star:BitmapData;
var bitmap:Bitmap;

for (var i:int = 0; i<MAX_NUM; i++)
{
    for (var j:int = 0; j<MAX_NUM; j++)
    {
        star = new Star(0,0);
        bitmap = new Bitmap(star);
        bitmap.x = j * star.width;
        bitmap.y = i * star.height;
        addChild(bitmap)
    }
}
```

На следующем рисунке показан результат выполнения кода.



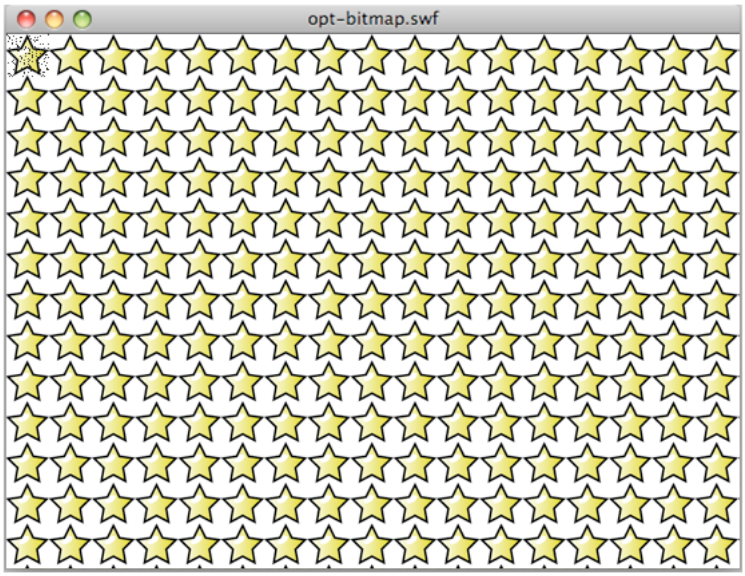
Результат выполнения кода для создания многочисленных экземпляров символа

Например, в проигрывателе Flash Player 10 для показа данной анимации требуется около 1008 КБ памяти. Проигрывателю Flash Player 10.1 на настольных системах и мобильных устройствах для этого требуется всего 4 КБ.

Следующий код изменяет один экземпляр BitmapData.

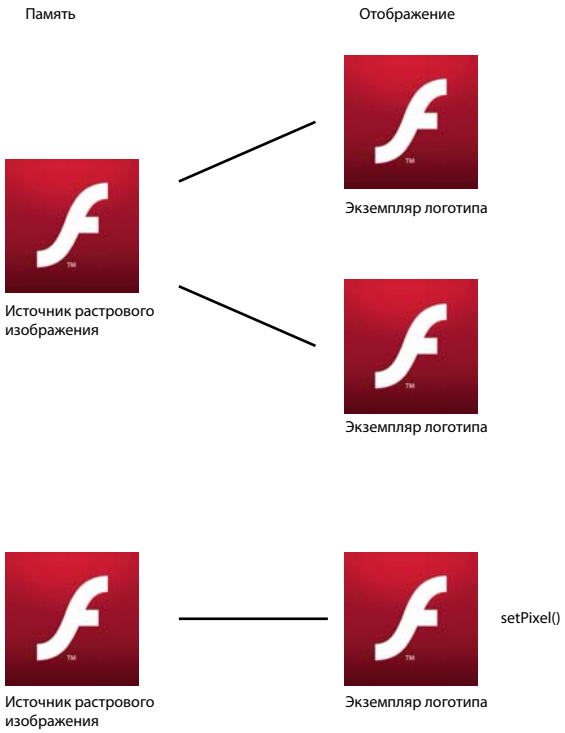
```
const MAX_NUM:int = 18;

var star:BitmapData;
var bitmap:Bitmap;

for (var i:int = 0; i<MAX_NUM; i++)
{
    for (var j:int = 0; j<MAX_NUM; j++)
    {
        star = new Star(0,0);
        bitmap = new Bitmap(star);
        bitmap.x = j * star.width;
        bitmap.y = i * star.height;
        addChild(bitmap)
    }
}

var ref:Bitmap = getChildAt(0) as Bitmap;

ref.bitmapData.pixelDissolve(ref.bitmapData, ref.bitmapData.rect, new
Point(0,0),Math.random()*200,Math.random()*200, 0x990000);
```

На следующем рисунке показан результат изменения одного экземпляра `Star`.



Результат изменения одного экземпляра

В теневом режиме среда выполнения автоматически назначает и создает растровое изображение в памяти для обработки изменений пикселов. При вызове метода класса BitmapData, выполняющего модификацию пикселов, в памяти создается новый экземпляр, а остальные экземпляры не обновляются. На следующем рисунке показан этот подход.



Память при изменении одного растрового изображения

При изменении одной звезды в памяти создается новая копия. Полученная анимация занимает в памяти около 8 КБ при воспроизведении с помощью Flash Player 10.1 и AIR 2.5.

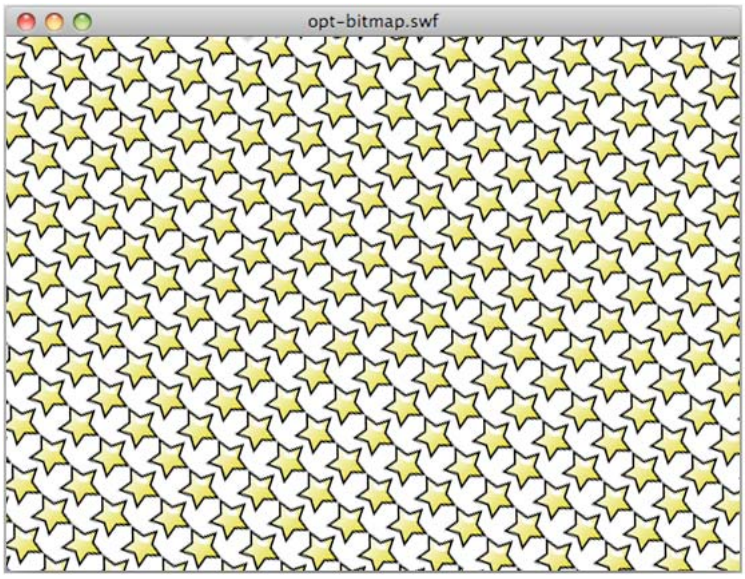
В предыдущем примере каждое растровое изображение преобразовывается отдельно. Если требуется лишь создать эффект плитки, лучше всего использовать метод `beginBitmapFill()`.

```
var container:Sprite = new Sprite();

var source:BitmapData = new Star(0,0);

// Fill the surface with the source BitmapData
container.graphics.beginBitmapFill(source);
container.graphics.drawRect(0,0,stage.stageWidth,stage.stageHeight);

addChild(container);
```

Этот подход позволяет получить тот же результат при создании только одного экземпляра BitmapData. Для непрерывного вращения звезд (вместо вращения каждого экземпляра Star по отдельности) используйте объект Matrix, который поворачивается в каждом кадре. Передайте объект Matrix методу `beginBitmapFill()`.

```
var container:Sprite = new Sprite();

container.addEventListener(Event.ENTER_FRAME, rotate);

var source:BitmapData = new Star(0,0);
var matrix:Matrix = new Matrix();

addChild(container);

var angle:Number = .01;

function rotate(e:Event):void
{
    // Rotate the stars
    matrix.rotate(angle);

    // Clear the content
    container.graphics.clear();

    // Fill the surface with the source BitmapData
    container.graphics.beginBitmapFill(source,matrix,true,true);
    container.graphics.drawRect(0,0,stage.stageWidth,stage.stageHeight);
}
```

Этот способ позволяет добиться нужного эффекта без создания циклов ActionScript. Среда выполнения осуществляет все операции в теневом режиме. На следующем рисунке показан результат преобразования звезд.



Результат поворота звезд

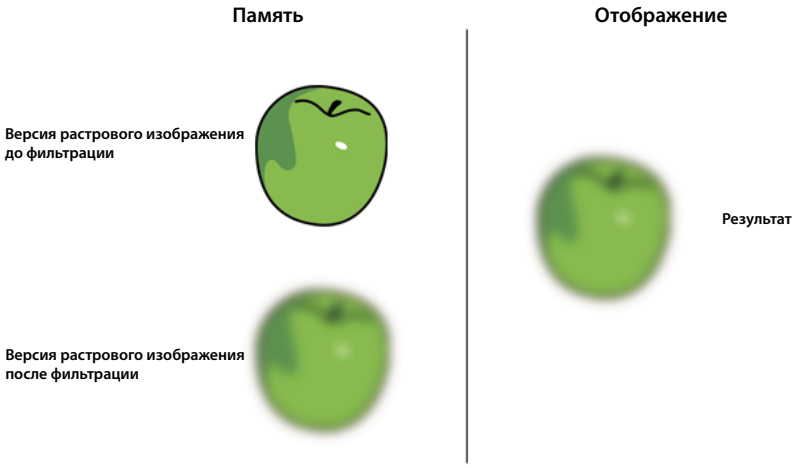
При использовании этого подхода обновление исходного объекта BitmapData влечет автоматическое обновление экземпляров этого объекта в любом месте рабочей области. Этот подход может быть эффективным. Однако при таком подходе нельзя отдельно масштабировать каждую звезду, как в предыдущем примере.

Примечание. *При использовании нескольких экземпляров одного изображения рисунок зависит от связи класса с исходным растровым изображением в памяти. Если класс не связан с растровым изображением, изображения создаются как объекты Shape с растровыми заливками.*

Фильтры и динамическая выгрузка растровых изображений

Избегайте использования фильтров, в том числе фильтров, обрабатываемых с помощью Pixel Bender.

Старайтесь не использовать такие эффекты, как фильтры, включая фильтры, обрабатываемые на мобильных устройствах с помощью Pixel Bender. При применении фильтра к экранному объекту среда выполнения создает два растровых изображения в памяти. Размер каждого растрового изображения равен размеру экранного объекта. Первое изображение создается в качестве растрированной версии экранного объекта, которая в свою очередь используется для создания второго растрового изображения с примененным фильтром.



Два растровых изображения в памяти при применении фильтра

При изменении одного из свойств фильтра оба растровых изображения обновляются в памяти для создания результирующего растрового изображения. Этот процесс включает ряд операций, обрабатываемых центральным процессором, и два растровых изображения могут использовать значительный объем памяти.

В проигрывателях Flash Player 10.1 и AIR 2.5 представлено новое поведение фильтра для всех платформ. Если фильтр не изменяется в течение 30 секунд или является скрытым либо закадровым, память, используемая растровым изображением без фильтра, освобождается.

Эта функция позволяет сэкономить половину памяти, используемой фильтром на всех платформах. Например, рассмотрим текстовый объект, к которому применен фильтр размытия. Текст в этом случае используется для простого оформления и не изменяется. По истечении 30 секунд растровое изображение без фильтра освобождается из памяти. Так же происходит, если текст скрыт в течение 30 секунд или находится вне экрана. При изменении одного из свойств текста растровое изображение без фильтра повторно создается в памяти. Эта функция называется динамической выгрузкой растрового изображения. Даже при использовании этих приемов по оптимизации проявляйте осторожность, работая с фильтрами; при их изменении по-прежнему будут интенсивно использоваться ресурсы центрального и графического процессоров.

Лучше всего использовать растровые изображения, созданные в средстве разработки, например в Adobe® Photoshop®, чтобы эмулировать фильтры, где это возможно. Не используйте динамические растровые изображения, созданные во время выполнения в ActionScript. Использование растровых изображений, созданных с помощью сторонних средств, помогает среде выполнения уменьшить загрузку центрального и графического процессоров, особенно когда свойства фильтра не изменяются со временем. По возможности применяйте к растровому изображению все необходимые эффекты в средстве разработки. Тогда оно будет отображаться в среде выполнения без дополнительной обработки, а значит, гораздо быстрее.

Прямое MIP-текстурирование

Используйте MIP-текстурирование для масштабирования крупных изображений.

Еще одна новая функция, доступная в проигрывателях Flash Player 10.1 и AIR 2.5 на всех платформах, связана с MIP-текстурированием. В проигрывателях Flash Player 9 и AIR 1.0 представлена функция MIP-текстурирования, позволяющая повысить качество и производительность уменьшенных растровых изображений.

Примечание. *Функция MIP-текстурирования подходит только для динамически загружаемых изображений или встроенных растровых изображений. MIP-текстурирование не применяется к отфильтрованным или кэшированным экранным объектам. Обработка MIP-текстурирования возможна, только если ширина и высота растрового изображения являются четными числами. Если ширина или высота растрового изображения является нечетным числом, обработка MIP-текстурирования останавливается. Например, к изображению 250 x 250 можно применить MIP-текстурирование до 125 x 125, но дальнейшее MIP-текстурирование будет невозможно. В этом случае хотя бы один из размеров является нечетным числом. Наиболее оптимальные результаты получаются, если растровые изображения имеют размеры, равные степени двойки, например: 256 x 256, 512 x 512, 1024 x 1024 и так далее.*

Например, если загружено изображение 1024 x 1024 и разработчику необходимо масштабировать его для создания миниатюры в галерее. Функция MIP-текстурирования обеспечивает правильную визуализацию изображения при масштабировании с использованием промежуточных субдискретизированных версий растрового изображения в качестве текстур. В предыдущих версиях среды выполнения промежуточные уменьшенные версии растрового изображения создавались в памяти. Если было загружено изображение 1024 x 1024, которое отображалось с размером 64 x 64, в более старых версиях среды выполнения каждый раз создавалось растровое изображение половинного размера. Например, в этом случае создавались растровые изображения 512 x 512, 256 x 256, 128 x 128 и 64 x 64.

Теперь проигрыватели Flash Player 10.1 и AIR 2.5 поддерживают прямое MIP-текстурирование исходного объекта до объекта с необходимым размером. В предыдущем примере создаются только исходное растровое изображение 4 МБ (1024 x 1024) и MIP-текстурированное растровое изображение 16 КБ (64 x 64).

Логика MIP-текстурирования также работает с функцией динамической выгрузки растровых изображений. Если используется только растровое изображение 64 x 64, исходное изображение размером 4 МБ освобождается из памяти. Если необходимо повторно создать MIP-текстурированную версию, исходное изображение загружается повторно. Кроме того, если требуются другие MIP-текстурированные растровые изображения разных размеров, цепочка MIP-текстурированных растровых изображений используется для создания растрового изображения. Например, если необходимо создать растровое изображение 1:8, исследуются растровые изображения 1:4, 1:2 и 1:1 на предмет того, какое изображение было загружено в память первым. Если другие версии не найдены, исходное растровое изображение 1:1 загружается из ресурса и используется.

Программа распаковки JPEG может выполнять MIP-текстурирование в своем собственном формате. Такое прямое MIP-текстурирование позволяет распаковывать большое растровое изображение непосредственно в формате MIP-текстурирования без необходимости загрузки всего несжатого изображения. Создание MIP-текстурированной версии выполняется существенно быстрее, и не происходит выделение и последующее освобождение памяти, используемой большими растровыми изображениями. Качество изображения JPEG сравнимо с качеством изображения, полученного общим методом MIP-текстурирования.

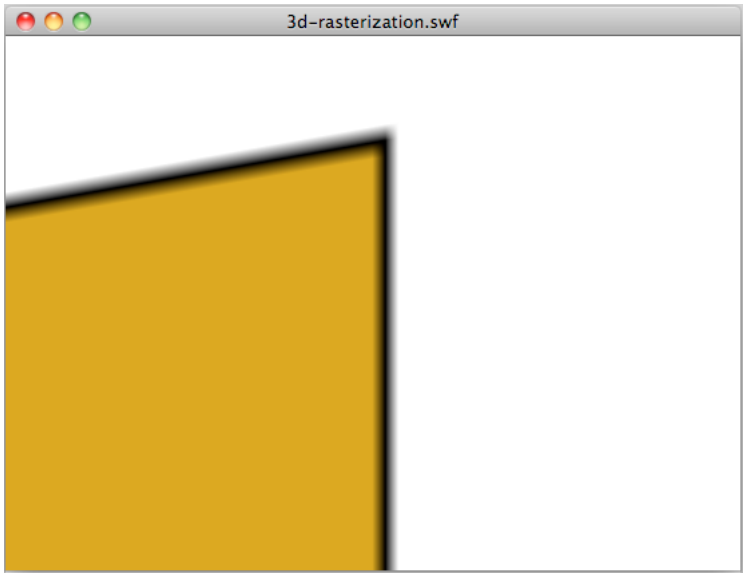
Примечание. *Не злоупотребляйте MIP-текстурированием. Хотя эта функция улучшает качество уменьшенных изображений, она влияет на пропускную способность, память и скорость. В некоторых случаях лучше уменьшить изображение в стороннем редакторе и импортировать его в приложение. Не используйте изображения большего размера, чем вам необходимо.*

Использование 3D-эффектов

Создавайте 3D-эффекты вручную.

В проигрывателях Flash Player 10 и AIR 1.5 представлен модуль 3D, позволяющий применять перспективные преобразования к экранным объектам. Эти преобразования можно применить с помощью свойств `rotationX` и `rotationY` или метода `drawTriangles()` класса Graphics. Можно также применить глубину с помощью свойства `z`. Помните, что каждый экранный объект, к которому применено перспективное преобразование, растрируется в качестве растрового изображения, и поэтому для него требуется больше памяти.

На следующем рисунке показано сглаживание, созданное путем растрирования при использовании перспективного преобразования.



Сглаживание в результате перспективного преобразования

Сглаживание является результатом динамической растеризации векторного изображения. Такое сглаживание возникает при использовании 3D-эффектов в версии проигрывателей AIR и Flash Player для настольных систем, а также в AIR 2.0.1 и AIR 2.5 для мобильных устройств. В проигрывателе Flash Player для мобильных устройств сглаживание не применяется.

Создание 3D-эффекта вручную без использования встроенного API-интерфейса позволяет уменьшить объем необходимой памяти. Однако новые функции для работы с 3D-объектами, представленные в проигрывателях Flash Player 10 и AIR 1.5, облегчают сопоставление текстуры благодаря таким методам, как `drawTriangles()`, который обрабатывает сопоставление текстуры своими ресурсами.

Разработчик должен решить, будет ли создаваемый 3D-эффект обеспечивать более высокий уровень производительности при обработке с использованием встроенного API-интерфейса или вручную. Учитывайте производительность при выполнении и визуализации кода ActionScript, а также использование памяти.

В приложениях AIR 2.0.1 и AIR 2.5 для мобильных устройств, в которых для свойства приложения `renderMode` установлено значение `GPU`, 3D-преобразования выполняет графический процессор. Если же для свойства `renderMode` установлено значение `CPU`, вместо графического процессора 3D-преобразования выполняет центральный процессор. В приложениях Flash Player 10.1 3D-преобразования выполняет центральный процессор.

Когда 3D-преобразования выполняет центральный процессор, для экранных объектов, к которым они применяются, требуется размещение двух растровых изображений в памяти. Первое является исходным растровым изображением, а второе — его версия с перспективным преобразованием. Таким образом, 3D-преобразования действуют подобно фильтрам. Поэтому в случаях, когда 3D-преобразования выполняет центральный процессор, не следует злоупотреблять свойствами 3D.

Текстовые объекты и память

Используйте Adobe® Flash® Text Engine для текста, доступного только для чтения. Используйте объекты TextField для вводимого текста.

В проигрывателях Flash Player 10 и AIR 1.5 представлен новый мощный механизм виртуализации текста Adobe Flash Text Engine (FTE), который снижает требования к системной памяти. Однако FTE является API-интерфейсом низкого уровня, поверх которого требуется дополнительный уровень ActionScript 3.0, поставляемый в виде пакета flash.text.engine.

Для текста, доступного только для чтения, лучше всего использовать Flash Text Engine, который обеспечивает более низкий уровень использования памяти и более высокий уровень визуализации. Для вводимого текста лучше использовать объекты TextField, поскольку при этом уменьшается размер кода ActionScript, необходимого для создания стандартных поведений, таких как обработка ввода и перенос слов.

Дополнительные разделы справки

«Визуализация текстовых объектов» на странице 70

Сравнение модели событий с обратными вызовами

Используйте обычные обратные вызовы вместо модели событий.

Модель событий ActionScript 3.0 основана на принципе отправки объектов. Она объектно ориентирована и оптимизирована для повторного использования кода. Метод `dispatchEvent()` циклически обрабатывает список прослушивателей и вызывает метод обработчика событий для каждого зарегистрированного объекта. Но у модели событий есть один недостаток: за время работы приложения может быть создано слишком много объектов.

Представьте себе, что временная шкала отправляет событие, указывающее на окончание последовательности анимации. Чтобы создать уведомление, можно отправить событие из конкретного кадра на временной шкале, как показано ниже.

```
dispatchEvent( new Event ( Event.COMPLETE ) );
```

Класс Document может прослушивать событие с помощью следующего кода.

```
addEventListener( Event.COMPLETE, onAnimationComplete );
```

Этот подход не является неправильным, однако использование собственной модели событий может замедлять работу и использовать больше ресурсов памяти, чем традиционная функция обратного вызова. Объекты событий создаются и хранятся в памяти, что снижает производительность. Например, при прослушивании события Event.ENTER_FRAME в каждом кадре создается объект события для обработчика. Особенно серьезное снижение производительности может наблюдаться при работе с экранными объектами за счет фаз захвата и цепочек. При сложном списке отображения для этого потребуется много ресурсов.

Глава 3. Уменьшение загрузки ЦП

Другим важным направлением оптимизации является загрузка центрального процессора. Оптимизация загрузки ЦП влияет на общую производительность, а следовательно, и на время работы мобильного устройства от аккумулятора.

Усовершенствованные возможности Flash Player 10.1 для использования ресурсов ЦП

В проигрывателе Flash Player 10.1 представлены две новые функции, позволяющие уменьшить загрузку центрального процессора. Эти функции позволяют приостанавливать и возобновлять воспроизведение SWF-содержимого, когда оно становится закадровым, и ограничивают число экземпляров Flash Player на странице.

Пауза, снижение скорости и возобновление

Примечание. *Функция паузы, снижения скорости и возобновления не применяется к приложениям Adobe® AIR®.*

Чтобы оптимизировать использование ресурсов ЦП и аккумулятора, в проигрыватель Flash Player 10.1 добавлена новая функция, предназначенная для работы с неактивными экземплярами. Она позволяет снизить нагрузку на ЦП за счет того, что когда содержимое появляется или исчезает с экрана, воспроизведение SWF-файла соответственно приостанавливается и возобновляется. Благодаря этой функции проигрыватель Flash Player освобождает всю возможную память путем удаления всех объектов, которые могут быть повторно созданы при возобновлении воспроизведения содержимого. Содержимое считается закадровым, если все содержимое находится за пределами экрана.

SWF-содержимое может стать закадровым в двух случаях:

- Во время прокрутки страницы пользователем SWF-содержимое перемещается за пределы экрана.

  В этом случае любое воспроизведение видео или аудио продолжает выполняться, но визуализация останавливается. Если аудио или видео не воспроизводится, чтобы воспроизведение или выполнение кода ActionScript не приостанавливалось, задайте для HTML-параметра `hasPriority` значение true. Однако помните, что визуализация SWF-содержимого приостанавливается, если содержимое является закадровым или скрытым вне зависимости от значения HTML-параметра `hasPriority`.

- При открытии вкладки в браузере SWF-содержимое перемещается на задний план.

  В этом случае независимо от значения тега HTML `hasPriority` скорость воспроизведения SWF-содержимого *снижается* до 2–8 кадров в секунду. Воспроизведение аудио и видео останавливается и визуализация содержимого не обрабатывается, пока SWF-содержимое снова не станет видимым.

В проигрывателе Flash Player 11.2 и более поздних версий, выполняющемся в браузерах на настольных компьютерах Windows и Mac, в приложении можно использовать событие ThrottleEvent. Flash Player отправляет событие ThrottleEvent в случае паузы, снижения скорости или возобновления воспроизведения.

Событие ThrottleEvent — это многоадресное событие, которое отправляется всеми объектами EventDispatcher, для которых зарегистрированы прослушиватели данного события. Дополнительные сведения о многоадресных событиях см. в описании класса DisplayObject.

Управление экземплярами

Примечание. *Функция управления экземплярами не применяется к приложениям Adobe® AIR®.*

Используйте параметр HTML `hasPriority`, чтобы отложить загрузку закадровых SWF-файлов.

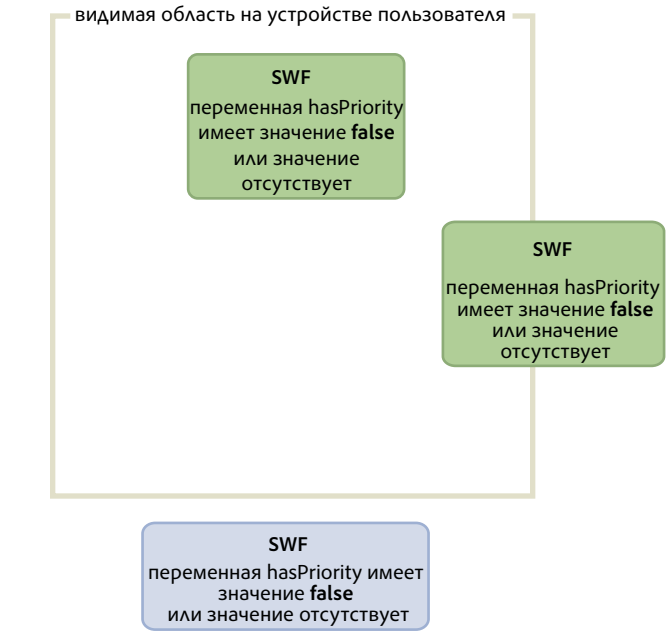
Во Flash Player 10.1 появился новый параметр HTML — `hasPriority`.

```
<param name="hasPriority" value="true" />
```

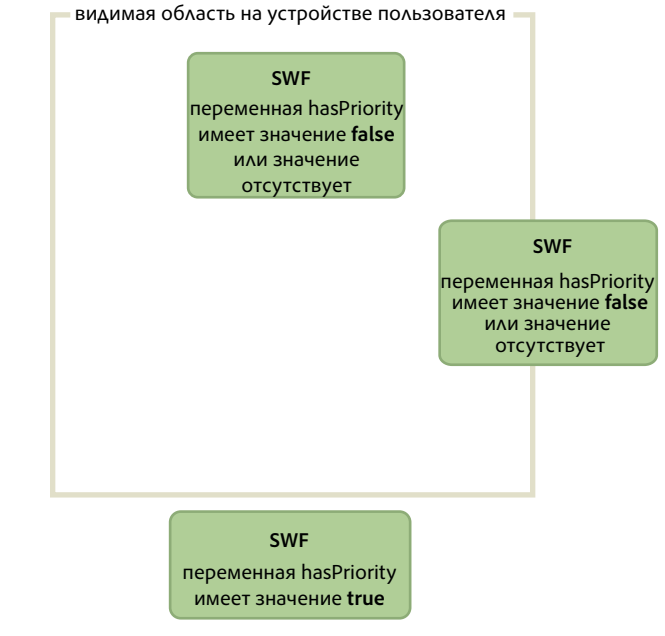
Эта функция ограничивает количество экземпляров Flash Player, запускаемых на странице. Это экономит ресурсы ЦП и аккумулятора. Целью является назначение определенного приоритета SWF-содержимому, благодаря чему одно содержимое становится более важным по сравнению с другим содержимым на странице. Рассмотрим простой пример: пользователь просматривает веб-сайт, а на странице указателей размещены три различных SWF-файла. Один из них является видимым, другой — частично видимым на экране, а последний — закадровым, требующим прокрутки. Первые две анимации запускаются в обычном режиме, а запуск последней анимации откладывается, пока она не станет видимой. Этот сценарий представляет поведение по умолчанию, когда параметр `hasPriority` отсутствует или имеет значение `false`. Для обеспечения запуска SWF-файла, даже если он находится за пределами экрана, задайте для параметра `hasPriority` значение `true`. Однако вне зависимости от значения параметра `hasPriority` визуализация SWF-файла, который невидим для пользователя, всегда приостанавливается.

Примечание. *Если доступных ресурсов ЦП недостаточно, новые экземпляры Flash Player не запускаются автоматически, даже если для параметра `hasPriority` задано значение `true`. Если новые экземпляры создаются с использованием JavaScript после загрузки страницы, эти экземпляры будут игнорировать флаг `hasPriority`. Любое содержимое размером 1x1 или 0x0 пикселов запускается, предотвращая отложенный запуск SWF-файлов помощника, если веб-мастер не включил флаг `hasPriority`. Однако SWF-файлы будут по-прежнему воспроизводиться по щелчку мыши. Это поведение называется «воспроизведением по щелчку» (click to play).*

На схемах ниже показано, как работает параметр `hasPriority` с разными значениями.



Результат использования разных значений параметра hasPriority



Результат использования разных значений параметра hasPriority

Спящий режим

В проигрывателях Flash Player 10.1 и AIR 2.5 представлена новая функция для мобильных устройств, позволяющая уменьшить загрузку ЦП и, как результат, продлить срок службы аккумулятора. Эта функция работает вместе с подсветкой, предусмотренной во многих мобильных устройствах. Например, если мобильное приложение запущено и пользователь больше не использует устройство, среда выполнения обнаруживает, когда подсветка переходит в спящий режим. Затем он сбрасывает частоту кадров до 4 кадров в секунду и приостанавливает визуализацию. Для приложений AIR переход в спящий режим выполняется, когда приложение перемещается на задний план.

Код ActionScript продолжает выполняться в спящем режиме, как если бы свойству `Stage.frameRate` было присвоено значение 4 кадра в секунду. Но этап визуализации пропускается, поэтому пользователь не видит того, что проигрыватель работает с частотой 4 кадра в секунду. Частота кадров, равная 4 кадрам в секунду, была выбрана вместо нуля, поскольку она позволяет сохранять открытыми все соединения (NetStream, Socket и NetConnection). При установке значения, равного нулю, будут разорваны все открытые соединения. Частота обновления, равная 250 мс (4 кадра в секунду) была выбрана, поскольку многие производители устройств используют эту частоту в качестве частоты обновления. Использование этого значения позволяет поддерживать частоту кадров среды выполнения примерно на том же уровне, что и частота кадров самого телефона.

Примечание. *Когда среда выполнения работает в спящем режиме, свойство* `Stage.frameRate` *возвращает частоту кадров исходного SWF-файла, а не 4 к/с.*

Когда подсветка снова переводится в активный режим, визуализация возобновляется. Для частоты кадров восстанавливается исходное значение. Рассмотрим приложение мультимедийного проигрывателя, в котором пользователь воспроизводит музыку. Если экран переходит в спящий режим, среда выполнения реагирует в зависимости от типа воспроизводимого содержимого. Далее приводится список ситуаций, а также описано соответствующее поведение среды выполнения.

- Подсветка переходит в спящий режим, воспроизводится содержимое, отличное от A/V. Визуализация приостанавливается, устанавливается частота 4 кадра в секунду.
- Подсветка переходит в спящий режим, воспроизводится содержимое A/V. Среда выполнения принудительно включает подсветку, обеспечивая непрерывное воспроизведение.
- Подсветка переходит из спящего режима в активный режим. Среда выполнения задает для таймера исходный параметр таймера SWF-файла и возобновляет визуализацию.
- Проигрыватель Flash Player приостанавливает работу при воспроизведении содержимого A/V. Проигрыватель Flash Player восстанавливает для состояния подсветки системное поведение по умолчанию, поскольку содержимое A/V больше не воспроизводится.
- Мобильное устройство получает вызов при воспроизведении содержимого A/V. Визуализация приостанавливается, устанавливается частота 4 кадра в секунду.
- Спящий режим подсветки отключен в мобильном устройстве. Среда выполнения работает в обычном режиме.

Когда для подсветки включается спящий режим, визуализация приостанавливается и частота кадров уменьшается. Эта функция позволяет снизить загрузку ЦП, но на нее нельзя полагаться при создании действительной паузы, например в игровом приложении.

Примечание. *Когда среда выполнения переходит в спящий режим или выходит из него, события ActionScript не отправляются.*

Замораживание и размораживание объектов

Замораживать и размораживать объекты можно с помощью событий `REMOVED_FROM_STAGE` и `ADDED_TO_STAGE`.

Для оптимизации кода всегда замораживайте и размораживайте объекты. Замораживание и размораживание особенно важны для экранных объектов. Даже если экранные объекты больше не включены в список отображения и ожидают удаления при сборке мусора, они по-прежнему могут использовать код, влияющий на загрузку ЦП. Например, они по-прежнему могут использовать событие Event.ENTER_FRAME. Таким образом, очень важно правильно замораживать и размораживать объекты. Для этого используйте события `Event.REMOVED_FROM_STAGE` и `Event.ADDED_TO_STAGE`. Ниже показан код фрагмента ролика, проигрываемого в рабочей области, с функциями управления с клавиатуры.

```
// Listen to keyboard events
stage.addEventListener(KeyboardEvent.KEY_DOWN, keyIsDown);
stage.addEventListener(KeyboardEvent.KEY_UP, keyIsUp);

// Create object to store key states
var keys:Dictionary = new Dictionary(true);

function keyIsDown(e:KeyboardEvent):void
{
    // Remember that the key was pressed
    keys[e.keyCode] = true;

    if (e.keyCode==Keyboard.LEFT || e.keyCode==Keyboard.RIGHT)
    {
        runningBoy.play();
    }
}

function keyIsUp(e:KeyboardEvent):void
{
    // Remember that the key was released
    keys[e.keyCode] = false;

    for each (var value:Boolean in keys)
        if ( value ) return;
    runningBoy.stop();
```

```
}

runningBoy.addEventListener(Event.ENTER_FRAME, handleMovement);
runningBoy.stop();

var currentState:Number = runningBoy.scaleX;
var speed:Number = 15;

function handleMovement(e:Event):void
{
    if (keys[Keyboard.RIGHT])
    {
        e.currentTarget.x += speed;
        e.currentTarget.scaleX = currentState;
    } else if (keys[Keyboard.LEFT])
    {
        e.currentTarget.x -= speed;
        e.currentTarget.scaleX = -currentState;
    }
}
```

Фрагмент ролика, который взаимодействует с клавиатурой

При нажатии кнопки «Удалить» фрагмент ролика удаляется из списка отображения.

```
// Show or remove running boy
showBtn.addEventListener (MouseEvent.CLICK,showIt);
removeBtn.addEventListener (MouseEvent.CLICK,removeIt);

function showIt (e:MouseEvent):void
{
    addChild (runningBoy);
}

function removeIt(e:MouseEvent):void
{
    if (contains(runningBoy)) removeChild(runningBoy);
}
```

Даже после удаления из списка отображения фрагмент ролика по-прежнему отправляет событие `Event.ENTER_FRAME`. Фрагмент ролика все еще выполняется, но не визуализируется. В этой ситуации необходимо прослушивать соответствующие события и удалять прослушиватели, чтобы код, повышающий нагрузку на ЦП, уже не выполнялся.

```
// Listen to Event.ADDED_TO_STAGE and Event.REMOVED_FROM_STAGE
runningBoy.addEventListener(Event.ADDED_TO_STAGE,activate);
runningBoy.addEventListener(Event.REMOVED_FROM_STAGE,deactivate);

function activate(e:Event):void
{
    // Restart everything
    e.currentTarget.addEventListener(Event.ENTER_FRAME,handleMovement);
}

function deactivate(e:Event):void
{
    // Freeze the running boy - consumes fewer CPU resources when not shown
    e.currentTarget.removeEventListener(Event.ENTER_FRAME,handleMovement);
    e.currentTarget.stop();
}
```

При нажатии на кнопку «Показать» воспроизведение возобновляется, снова прослушиваются события `Event.ENTER_FRAME`, а управление роликом с клавиатуры работает корректно.

Примечание. *Если экранный объект удаляется из списка отображения, задание для соответствующей ссылки значения `null` после его удаления не обеспечивает замораживание объекта. Если сборщик мусора не запускается, объект продолжает использовать ресурсы памяти и ЦП, даже если объект больше не отображается. Чтобы объект использовал минимум ресурсов ЦП, убедитесь, что он полностью заморожен при удалении из списка отображения.*

Начиная с версий Flash Player 10 и AIR 1.5 реализовано следующее поведение. Если точка воспроизведения обнаруживает пустой кадр, экранный объект замораживается автоматически, даже если разработчик не реализовал поведение замораживания.

Принцип замораживания также важен при загрузке удаленного содержимого с использованием класса Loader. При использовании класса Loader в проигрывателе Flash Player 9 и AIR 1.0 нужно было вручную размораживать содержимое путем прослушивания события `Event.UNLOAD`, отправленного объектом LoaderInfo. Каждый объект нужно было размораживать вручную, что являлось нетривиальной задачей. В проигрывателях Flash Player 10 и AIR 1.5 в классе Loader представлен новый важный метод `unloadAndStop()`. Этот метод позволяет выгрузить SWF-файл, автоматически разморозить каждый объект в загруженном SWF-файле и принудительно запустить выполнение сборки мусора.

В следующем коде SWF-файл загружается, а затем выгружается с использованием метода `unload()`, что требует дополнительной обработки и замораживания вручную.

```
var loader:Loader = new Loader();

loader.load ( new URLRequest ( "content.swf" ) );

addChild ( loader );

stage.addEventListener ( MouseEvent.CLICK, unloadSWF );

function unloadSWF ( e:MouseEvent ):void
{
    // Unload the SWF file with no automatic object deactivation
    // All deactivation must be processed manually
    loader.unload();
}
```

Наилучшим приемом является использование метода `unloadAndStop()`, в котором предусмотрены внутренняя обработка замораживания и принудительное включение процесса сборки мусора.

```
var loader:Loader = new Loader();

loader.load ( new URLRequest ( "content.swf" ) );

addChild ( loader );

stage.addEventListener ( MouseEvent.CLICK, unloadSWF );

function unloadSWF ( e:MouseEvent ):void
{
    // Unload the SWF file with automatic object deactivation
    // All deactivation is handled automatically
    loader.unloadAndStop();
}
```

При вызове метода `unloadAndStop()` выполняются следующие действия.

- Останавливается воспроизведение звука.
- Прослушиватели, зарегистрированные для основной временной шкалы SWF-файла, удаляются.
- Объекты таймера останавливаются.
- Аппаратные периферийные устройства (например, камера и микрофон) освобождаются.
- Каждый фрагмент ролика останавливается.
- Отправка событий `Event.ENTER_FRAME`, `Event.FRAME_CONSTRUCTED`, `Event.EXIT_FRAME`, `Event.ACTIVATE` и `Event.DEACTIVATE` останавливается.

Активация и отключение событий

Определять бездействие при перемещении на задний план и оптимизировать работу приложения соответствующим образом можно с помощью событий `Event.ACTIVATE` и `Event.DEACTIVATE`.

Два события (`Event.ACTIVATE` и `Event.DEACTIVATE`) помогут настроить приложение таким образом, чтобы для него требовалось как можно меньше ресурсов ЦП. Эти события позволяют обнаружить, когда среда выполнения получает или теряет фокус. Таким образом, код можно оптимизировать в соответствии с изменениями контекста. Следующий код прослушивает оба события и динамически изменяет частоту кадров до нуля, когда приложение теряет фокус. Например, приложение может потерять фокус, если пользователь переключится на другую вкладку или переместит приложение на задний план.

```
var originalFrameRate:uint = stage.frameRate;
var standbyFrameRate:uint = 0;

stage.addEventListener ( Event.ACTIVATE, onActivate );
stage.addEventListener ( Event.DEACTIVATE, onDeactivate );

function onActivate ( e:Event ):void
{
    // restore original frame rate
    stage.frameRate = originalFrameRate;
}

function onDeactivate ( e:Event ):void
{
    // set frame rate to 0
    stage.frameRate = standbyFrameRate;
}
```

Когда приложение вновь получает фокус, восстанавливается исходное значение частоты кадров. Вместо значительного изменения частоты кадров можно применить другие способы оптимизации, например замораживание и размораживание объектов.

С помощью активации и отключения событий можно реализовать аналогичный механизм для функции паузы и возобновления, которая иногда встречается на мобильных устройствах и нетбуках.

Дополнительные разделы справки

«Частота кадров приложения» на странице 55

«Замораживание и размораживание объектов» на странице 30

Взаимодействие с мышью

По возможности отключайте функции взаимодействия с мышью.

При работе с интерактивным объектом, таким как фрагмент ролика или спрайт, среда выполнения исполняет собственный код для обнаружения и обработки взаимодействия с мышью. Если на экране много интерактивных объектов, тем более если они наложены друг на друга, обнаружение взаимодействия с мышью требует значительных ресурсов ЦП. Чтобы избежать ненужной обработки, отключите взаимодействие с мышью объектов, которым оно не требуется. Ниже показано использование в коде свойств `mouseEnabled` и `mouseChildren`.

```
// Disable any mouse interaction with this InteractiveObject
myInteractiveObject.mouseEnabled = false;
const MAX_NUM:int = 10;

// Create a container for the InteractiveObjects
var container:Sprite = new Sprite();

for ( var i:int = 0; i< MAX_NUM; i++ )
{
    // Add InteractiveObject to the container
    container.addChild( new Sprite() );
}

// Disable any mouse interaction on all the children
container.mouseChildren = false;
```

Отключение взаимодействия с мышью экономит ресурсы ЦП, а следовательно, и аккумулятора на мобильных устройствах, поэтому по возможности отключайте его.

Сравнение таймеров с событиями ENTER_FRAME

Выберите таймеры или события `ENTER_FRAME` — в зависимости от наличия анимации.

Для неанимированного содержимого, выполняемого в течение длительного времени, таймеры предпочтительнее событий `Event.ENTER_FRAME`.

В ActionScript 3.0 предусмотрено два способа вызова функции через определенные интервалы. Первый — с помощью события `Event.ENTER_FRAME`, отправляемого экранными объектами (DisplayObject). Второй — с помощью таймера. Разработчики ActionScript часто используют первый способ (событие `ENTER_FRAME`). Событие `ENTER_FRAME` отправляется в каждом кадре, поэтому интервал вызова функции зависит от текущей частоты кадров. Частота кадров задается свойством `Stage.frameRate`. Однако в ряде случаев лучше использовать таймер, чем событие `ENTER_FRAME`. Например, если у вас нет анимации, но вам требуется, чтобы код вызывался через заданные интервалы, следует использовать таймер.

Таймер действует подобно событию `ENTER_FRAME`, однако событие может отправляться без привязки к частоте кадров. Такое поведение дает простор для оптимизации. Рассмотрим в качестве примера проигрыватель видео. В этом случае высокая частота кадров не нужна, так как движение контролируется только приложением.

Примечание. *Частота кадров не влияет на видео, потому что видео не встроено во временную шкалу. Вместо этого видео загружается динамически, путем последовательной загрузки либо потокового видео.*

В этом примере частота кадров составляет 10 к/с. Таймер обновляет элементы управления с частотой раз в секунду. Метод `updateAfterEvent()` объекта TimerEvent позволяет повысить частоту обновления. Этот метод вызывает обновление экрана при каждой отправке события. Это проиллюстрировано в следующем коде.

```
// Use a low frame rate for the application
stage.frameRate = 10;

// Choose one update per second
var updateInterval:int = 1000;
var myTimer:Timer = new Timer(updateInterval,0);

myTimer.start();
myTimer.addEventListener( TimerEvent.TIMER, updateControls );

function updateControls( e:TimerEvent ):void
{
    // Update controls here
    // Force the controls to be updated on screen
    e.updateAfterEvent();
}
```

Вызов метода `updateAfterEvent()` не изменяет частоту кадров. Он просто заставляет среду выполнения обновлять на экране измененное содержимое. Временная шкала по-прежнему обновляется с частотой 10 к/с. Имейте в виду, что точность таймеров и событий `ENTER_FRAME` может снижаться при выполнении на низкопроизводительных устройствах или в случаях, когда функции обработчика событий содержат код, требующий интенсивной обработки. Как и частота кадров SWF-файла, частота обновления кадров таймера может зависеть от ряда факторов.

Сократите до минимума количество объектов Timer и зарегистрированных обработчиков события `enterFrame` в приложении.

Каждый кадр среды выполнения отправляет событие `enterFrame` всем экранным объектам из списка отображения. Можно зарегистрировать прослушиватели события `enterFrame` и для нескольких экранных объектов, однако это приведет к увеличению объема кода, выполняемого в каждом кадре. Вместо этого используйте единый централизованный обработчик `enterFrame` для выполнения всего объема кода в каждом кадре. Централизация кода облегчит управление часто выполняемым кодом.

Аналогичным образом обстоит ситуация и с объектами Timer. Создание и отправка событий множеством объектов Timer сильно нагружают код. Если необходимо запустить различные операции с различными интервалами, ниже приводятся некоторые рекомендуемые альтернативные варианты:

- Сократите до минимума количество объектов Timer и групповых операций в зависимости от частоты их выполнения.

  Например, используйте экземпляр Timer для частых операций, которые необходимо выполнять каждые 100 миллисекунд. Затем используйте еще один экземпляр Timer для частых операций, которые необходимо выполнять менее часто, каждые 2000 миллисекунд.

- Используйте единственный объект Timer, в котором период выполнения различных операций задан в свойстве `delay`.

  К примеру, необходимо, чтобы одни операции выполнялись каждые 100 миллисекунд, а другие — каждые 200 миллисекунд. В этом случае используйте один объект Timer, свойство `delay` которого настроено на 100 миллисекунд. В обработчике событий `timer` добавьте условную конструкцию, после чего каждая вторая операция будет выполняться с интервалом 200 миллисекунд. Следующий пример иллюстрирует такой подход:

```
var timer:Timer = new Timer(100);
timer.addEventListener(TimerEvent.Timer, timerHandler);
timer.start();

var offCycle:Boolean = true;

function timerHandler(event:TimerEvent):void
{
    // Do things that happen every 100 ms

    if (!offCycle)
    {
        // Do things that happen every 200 ms
    }

    offCycle = !offCycle;
}
```

Останавливайте неиспользуемые объекты Timer.

Если обработчик события `timer` объекта Timer выполняет операции только при определенных условиях, вызовите метод `stop()` объекта Timer, если ни одно из условий не принимает значение true.

В событии `enterFrame` или обработчиках Timer уменьшите число изменений внешнего вида экранных объектов, чтобы экран обновлялся как можно реже.

В каждом кадре в фазе визуализации производится перерисовка изменившихся элементов рабочей области. Если область перерисовки большая, или если область перерисовки небольшая, но содержит множество сложных экранных объектов, среде выполнения требуется больше времени для визуализации. Чтобы узнать объем перерисовки, воспользуйтесь функцией «Показать область перерисовки» в отладчике проигрывателя Flash Player или AIR.

Дополнительные сведения об улучшении производительности повторяемых действий см. в следующих статьях.

- Создание стабильных и производительных приложений AIR (статья и пример приложения Арно Гурдола (Arno Gourdol))

Дополнительные разделы справки

«Изоляция поведений» на странице 67

Признак анимации

Старайтесь не использовать анимацию, так как для ее обработки требуется много ресурсов ЦП и памяти, что сокращает время работы от аккумулятора.

Дизайнеры и разработчики, создающие Flash-приложения для настольных компьютеров, часто используют анимацию движения. При создании содержимого для мобильных устройств старайтесь свести анимацию к минимуму. Без анимации приложения будет быстрее работать на низкопроизводительных устройствах.

Глава 4. Производительность ActionScript 3.0

Сравнение класса Vector с классом Array

По возможности используйте класс Vector вместо класса Array.

Класс Vector обеспечивает более быстрый доступ для чтения и записи, чем класс Array.

Простой тест показывает преимущества класса Vector над классом Array. С помощью следующего кода показано тестирование класса Array.

```
var coordinates:Array = new Array();
var started:Number = getTimer();

for (var i:int = 0; i< 300000; i++)
{
    coordinates[i] = Math.random()*1024;
}

trace(getTimer() - started);
// output: 107
```

С помощью следующего кода показано тестирование класса Vector.

```
var coordinates:Vector.<Number> = new Vector.<Number>();
var started:Number = getTimer();

for (var i:int = 0; i< 300000; i++)
{
    coordinates[i] = Math.random()*1024;
}

trace(getTimer() - started);
// output: 72
```

Можно еще больше оптимизировать пример за счет назначения определенной длины вектору и настройки для него фиксированной длины:

```
// Specify a fixed length and initialize its length
var coordinates:Vector.<Number> = new Vector.<Number>(300000, true);

var started:Number = getTimer();

for (var i:int = 0; i< 300000; i++)
{
    coordinates[i] = Math.random()*1024;
}

trace(getTimer() - started);
// output: 48
```

Если размер вектора изначально не задан, он увеличивается, когда вектор выходит за текущие границы. Каждый раз при увеличении вектора ему выделяется новый блок памяти. Текущее содержимое вектора копируется в новый блок памяти. Дополнительные операции копирования и выделения памяти отражаются на производительности. Код, представленный выше, оптимизирован для лучшей производительности за счет указания исходного размера вектора. Однако он не оптимизирован для целей поддержки. Для этих целей сохраните часто используемое значение в качестве константы.

```
// Store the reused value to maintain code easily
const MAX_NUM:int = 300000;

var coordinates:Vector.<Number> = new Vector.<Number>(MAX_NUM, true);
var started:Number = getTimer();

for (var i:int = 0; i< MAX_NUM; i++)
{
    coordinates[i] = Math.random()*1024;
}

trace(getTimer() - started);
// output: 47
```

По возможности пользуйтесь API-интерфейсами объектов Vector, так как они работают быстрее.

API-интерфейс рисования

Используйте API-интерфейс рисования для ускорения выполнения кода.

Проигрыватели Flash Player 10 и AIR 1.5 предоставляют новый API-интерфейс рисования, который позволяет повысить производительность при выполнении кода. Этот новый API-интерфейс не обеспечивает повышение производительности визуализации, но позволяет значительно уменьшить число записываемых строк кода. Уменьшение числа строк кода обеспечивает более высокий уровень производительности при выполнении кода ActionScript.

Новый API-интерфейс рисования включает следующие методы:

- `drawPath()`
- `drawGraphicsData()`
- `drawTriangles()`

Примечание. *В этом документе подробно не рассматривается метод `drawTriangles()`, относящийся к трехмерной визуализации. Однако этот метод позволяет повысить производительность ActionScript, поскольку он обрабатывает собственное сопоставление текстуры.*

Следующий код явно вызывает подходящий метод для каждой рисуемой линии.

```
var container:Shape = new Shape();
container.graphics.beginFill(0x442299);

var coords:Vector.<Number> = Vector.<Number>([132, 20, 46, 254, 244, 100, 20, 98, 218, 254]);

container.graphics.moveTo ( coords[0], coords[1] );
container.graphics.lineTo ( coords[2], coords[3] );
container.graphics.lineTo ( coords[4], coords[5] );
container.graphics.lineTo ( coords[6], coords[7] );
container.graphics.lineTo ( coords[8], coords[9] );

addChild( container );
```

Следующий код выполняется быстрее, чем код в предыдущем примере, поскольку выполнению подлежит меньшее число строк. Чем сложнее путь, тем выше прирост производительности при использовании метода `drawPath()`.

```
var container:Shape = new Shape();
container.graphics.beginFill(0x442299);

var commands:Vector.<int> = Vector.<int>([1,2,2,2,2]);
var coords:Vector.<Number> = Vector.<Number>([132, 20, 46, 254, 244, 100, 20, 98, 218, 254]);

container.graphics.drawPath(commands, coords);

addChild( container );
```

Метод `drawGraphicsData()` обеспечивает похожие улучшения производительности.

Захват событий и цепочки событий

Захват и цепочки событий позволяют уменьшить количество обработчиков событий.

В модели события ActionScript 3.0 представлены такие принципы, как захват событий и цепочки событий. Цепочки событий позволяют быстрее выполнять код ActionScript. Обработчик событий можно зарегистрировать не для нескольких объектов, а всего для одного, что ускорит обработку.

Представим, что разработчик пишет код для игры, в которой нужно как можно скорее собирать яблоки щелчком мыши. При каждом щелчке яблоко исчезает, а игроку добавляются очки. Для прослушивания события `MouseEvent.CLICK`, отправляемого каждым яблоком, разработчик может написать следующий код.

```
const MAX_NUM:int = 10;
var sceneWidth:int = stage.stageWidth;
var sceneHeight:int = stage.stageHeight;
var currentApple:InteractiveObject;
var currentAppleClicked:InteractiveObject;

for ( var i:int = 0; i< MAX_NUM; i++ )
{
    currentApple = new Apple();
    currentApple.x = Math.random()*sceneWidth;
    currentApple.y = Math.random()*sceneHeight;
    addChild ( currentApple );

    // Listen to the MouseEvent.CLICK event
    currentApple.addEventListener ( MouseEvent.CLICK, onAppleClick );
}

function onAppleClick ( e:MouseEvent ):void
{
    currentAppleClicked = e.currentTarget as InteractiveObject;
    currentAppleClicked.removeEventListener(MouseEvent.CLICK, onAppleClick );
    removeChild ( currentAppleClicked );
}
```

Для каждого экземпляра Apple вызывается код `addEventListener()`. Когда игрок щелкает по яблоку, с помощью метода `removeEventListener()` также удаляется каждый прослушиватель. Однако в ActionScript 3.0 есть возможность создавать захваты и цепочки событий, чтобы прослушивать их с помощью родительских интерактивных объектов (InteractiveObject). Это позволяет оптимизировать показанный выше код и свести к минимуму вызовы методов `addEventListener()` и `removeEventListener()`. В следующем коде для прослушивания событий с родительского объекта используется фаза захвата.

```
const MAX_NUM:int = 10;
var sceneWidth:int = stage.stageWidth;
var sceneHeight:int = stage.stageHeight;
var currentApple:InteractiveObject;
var currentAppleClicked:InteractiveObject;
var container:Sprite = new Sprite();

addChild ( container );

// Listen to the MouseEvent.CLICK on the apple's parent
// Passing true as third parameter catches the event during its capture phase
container.addEventListener ( MouseEvent.CLICK, onAppleClick, true );

for ( var i:int = 0; i< MAX_NUM; i++ )
{
    currentApple = new Apple();
    currentApple.x = Math.random()*sceneWidth;
    currentApple.y = Math.random()*sceneHeight;
    container.addChild ( currentApple );
}

function onAppleClick ( e:MouseEvent ):void
{
    currentAppleClicked = e.target as InteractiveObject;
    container.removeChild ( currentAppleClicked );
}
```

Как видим, этот код проще и гораздо лучше оптимизирован. Метод `addEventListener()` вызывается лишь единожды на родительском контейнере. Прослушиватели больше не связаны с экземплярами Apple, поэтому их не нужно удалять каждый раз при щелчке по яблоку. Обработчик `onAppleClick()` можно оптимизировать и далее. Для этого нужно остановить распространение события.

```
function onAppleClick ( e:MouseEvent ):void
{
    e.stopPropagation();
    currentAppleClicked = e.target as InteractiveObject;
    container.removeChild ( currentAppleClicked );
}
```

Фазу захвата можно также использовать для перехвата события. Для этого нужно передать методу `addEventListener()` значение `false` в качестве третьего параметра.

```
// Listen to the MouseEvent.CLICK on apple's parent
// Passing false as third parameter catches the event during its bubbling phase
container.addEventListener ( MouseEvent.CLICK, onAppleClick, false );
```

Значение параметра фазы захвата по умолчанию — `false`, поэтому его можно не указывать.

```
container.addEventListener ( MouseEvent.CLICK, onAppleClick );
```

Работа с пикселами

Нарисуйте пикселы с помощью метода `setVector()`.

При рисовании пикселов можно выполнить простые приемы по оптимизации с использованием соответствующих методов класса BitmapData. Быстрым способом рисования пикселов является использование метода `setVector()`:

```
// Image dimensions
var wdth:int = 200;
var hght:int = 200;
var total:int = wdth*hght;

// Pixel colors Vector
var pixels:Vector.<uint> = new Vector.<uint>(total, true);

for ( var i:int = 0; i< total; i++ )
{
    // Store the color of each pixel
    pixels[i] = Math.random()*0xFFFFFF;
}

// Create a non-transparent BitmapData object
var myImage:BitmapData = new BitmapData ( wdth, hght, false );
var imageContainer:Bitmap = new Bitmap ( myImage );

// Paint the pixels
myImage.setVector ( myImage.rect, pixels );
addChild ( imageContainer );
```

При использовании медленных методов, например `setPixel()` или `setPixel32()`, используйте методы `lock()` и `unlock()` для ускорения выполнения команд. В следующем коде методы `lock()` и `unlock()` используются для повышения производительности.

```
var buffer:BitmapData = new BitmapData(200,200,true,0xFFFFFFFF);
var bitmapContainer:Bitmap = new Bitmap(buffer);
var positionX:int;
var positionY:int;

// Lock update
buffer.lock();
var starting:Number=getTimer();

for (var i:int = 0; i<2000000; i++)
{
    // Random positions
    positionX = Math.random()*200;
    positionY = Math.random()*200;
    // 40% transparent pixels
    buffer.setPixel32( positionX, positionY, 0x66990000 );
}

// Unlock update
buffer.unlock();
addChild( bitmapContainer );

trace( getTimer () - starting );
// output : 670
```

Метод `lock()` класса BitmapData блокирует изображение и предотвращает обновление объектов, ссылающихся на изображение, при внесении изменений в объект BitmapData. Например, если объект Bitmap ссылается на объект BitmapData, можно заблокировать объект BitmapData, изменить, а затем разблокировать его. Объект Bitmap остается неизменным, пока объект BitmapData не разблокирован. Чтобы повысить производительность, используйте этот метод вместе с методом `unlock()` до или после многочисленных вызовов метода `setPixel()` или `setPixel32()`. Вызов методов `lock()` и `unlock()` позволяет предотвратить нецелесообразное обновление экрана.

Примечание. *При обработке пикселов растрового изображения, не включенного в список отображения (двойной буферизации), эта техника может не дать результатов. Если растровое изображение не обращается к буферу растровых изображений, использование методов `lock()` и `unlock()` не повысит производительность. Проигрыватель Flash Player обнаруживает, что отсутствует ссылка на буфер, и растровое изображение не визуализируется на экране.*

Методы с попиксельной итерацией, такие как `getPixel()`, `getPixel32()`, `setPixel()` и `setPixel32()`, могут работать медленно, особенно на мобильных устройствах. По возможности используйте методы, извлекающие все пикселы одним вызовом. Для чтения пикселов используйте метод `getVector()`, который работает быстрее метода `getPixels()`. Кроме того, по возможности используйте API-интерфейсы, которые основаны на объектах Vector, поскольку они работают быстрее.

Регулярные выражения

Вместо регулярных выражений для стандартного поиска и извлечения строки используйте методы класса String, например `indexOf()`, `substr()` или `substring()`.

Определенные операции, выполняемые с использованием регулярного выражения, также можно выполнить с помощью методов класса String. Например, чтобы определить, содержится ли в строке еще одна строка, можно использовать регулярное выражение или метод `String.indexOf()`. По возможности используйте метод класса String, так как он работает быстрее эквивалентного регулярного выражения и не создает дополнительный объект.

Используйте в регулярном выражении незахватывающую группу («`(?:xxxx)`») вместо группы («`(xxxx)`»), если необходимо сгруппировать элементы, не изолируя содержимое группы в результате.

Часто в регулярных выражениях средней сложности части выражения можно сгруппировать. Например, в следующем шаблоне регулярного выражения используются круглые скобки для объединения текста ab в группу. Следовательно, квантификатор «+» применяется к группе, а не к одному символу.

```
/(ab)+/
```

По умолчанию содержимое каждой группы «захватывается». Содержимое каждой группы в данном шаблоне можно получить как часть результата выполнения регулярного выражения. Из-за создания дополнительных объектов для хранения результатов групп операция выполняется дольше и требует дополнительные ресурсы памяти. Вместо этого можно использовать синтаксис незахватывающей группы, поставив вопросительный знак с двоеточием после открывающей круглой скобки. Такой синтаксис указывает на то, что символы в круглых скобках образуют группу, однако результат вычисления для этой группы отдельно не сохраняется.

```
/(?:ab)+/
```

Использование этого синтаксиса повышает скорость вычислений и использует меньше памяти по сравнению со стандартным синтаксисом группы.

Если обработка регулярного выражения производится медленно, попробуйте использовать альтернативный шаблон регулярного выражения.

Иногда для проверки или идентификации одного и того же шаблона текста можно использовать несколько шаблонов регулярных выражений. По различным причинам некоторые шаблоны обрабатываются быстрее своих аналогов. Если выявлено, что из-за использования регулярного выражения код выполняется медленнее, чем необходимо, рассмотрите возможность использования альтернативных регулярных выражений, позволяющих получить такой же результат. Определите, какой из этих альтернативных шаблонов обрабатывается быстрее.

Разные приемы по оптимизации

Для объекта TextField используйте метод `appendText()` вместо оператора `+=`.

При работе со свойством `text` класса TextField используйте метод `appendText()` вместо оператора `+=`. Использование метода `appendText()` позволяет оптимизировать производительность.

Для примера в следующем коде используется оператор `+=`, и выполнение цикла занимает 1120 мс.

```
addChild ( myTextField );

myTextField.autoSize = TextFieldAutoSize.LEFT;
var started:Number = getTimer();

for (var i:int = 0; i< 1500; i++ )
{
    myTextField.text += "ActionScript 3";
}

trace( getTimer() - started );
// output : 1120
```

В следующем примере вместо оператора += используется метод `appendText()`.

```
var myTextField:TextField = new TextField();
addChild ( myTextField );
myTextField.autoSize = TextFieldAutoSize.LEFT;

var started:Number = getTimer();

for (var i:int = 0; i< 1500; i++ )
{
    myTextField.appendText ( "ActionScript 3" );
}

trace( getTimer() - started );
// output : 847
```

Теперь выполнение кода занимает 847 мс.

По возможности обновляйте текстовые поля вне циклов.

Можно выполнить дальнейшую оптимизацию этого кода с помощью простого приема. Для внутренней обработки обновления текстового поля внутри каждого цикла используются значительные ресурсы. За счет сцепления строки и ее присвоения текстовому полю вне цикла существенно уменьшается время выполнения кода. Теперь выполнение кода занимает 2 мс.

```
var myTextField:TextField = new TextField();
addChild ( myTextField );
myTextField.autoSize = TextFieldAutoSize.LEFT;

var started:Number = getTimer();
var content:String = myTextField.text;

for (var i:int = 0; i< 1500; i++ )
{
    content += "ActionScript 3";
}

myTextField.text = content;

trace( getTimer() - started );
// output : 2
```

При работе с текстом HTML используемый ранее подход был настолько медленным, что в некоторых случаях мог порождать исключение `Timeout` в проигрывателе Flash Player. Например, исключение могло быть создано, если используемое аппаратное обеспечение было недостаточно производительно.

Примечание. *В среде Adobe® AIR® такое исключение не создается.*

```
var myTextField:TextField = new TextField();
addChild ( myTextField );
myTextField.autoSize = TextFieldAutoSize.LEFT;

var started:Number = getTimer();

for (var i:int = 0; i< 1500; i++ )
{
    myTextField.htmlText += "ActionScript <b>2</b>";
}

trace( getTimer() - started );
```

За счет присвоения значения строке вне цикла выполнение кода занимает всего 29 мс.

```
var myTextField:TextField = new TextField();
addChild ( myTextField );
myTextField.autoSize = TextFieldAutoSize.LEFT;

var started:Number = getTimer();
var content:String = myTextField.htmlText;

for (var i:int = 0; i< 1500; i++ )
{
    content += "<b>ActionScript<b> 3";
}

myTextField.htmlText = content;

trace ( getTimer() - started );
// output : 29
```

Примечание. *В проигрывателях Flash Player 10.1 и AIR 2.5 был усовершенствован класс String, поэтому для строк используется меньше памяти.*

По возможности избегайте использования оператора квадратной скобки.

Использование оператора квадратной скобки может привести к снижению производительности. Избежать его использования можно путем хранения ссылки в локальной переменной. Следующий пример кода демонстрирует неэффективное использование оператора квадратной скобки.

```
var lng:int = 5000;
var arraySprite:Vector.<Sprite> = new Vector.<Sprite>(lng, true);
var i:int;

for ( i = 0; i< lng; i++ )
{
    arraySprite[i] = new Sprite();
}

var started:Number = getTimer();

for ( i = 0; i< lng; i++ )
{
    arraySprite[i].x = Math.random()*stage.stageWidth;
    arraySprite[i].y = Math.random()*stage.stageHeight;
    arraySprite[i].alpha = Math.random();
    arraySprite[i].rotation = Math.random()*360;
}

trace( getTimer() - started );
// output : 16
```

Следующая оптимизированная версия позволяет по возможности избежать использования оператора квадратной скобки.

```
var lng:int = 5000;
var arraySprite:Vector.<Sprite> = new Vector.<Sprite>(lng, true);
var i:int;

for ( i = 0; i< lng; i++ )
{
    arraySprite[i] = new Sprite();
}

var started:Number = getTimer();
var currentSprite:Sprite;

for ( i = 0; i< lng; i++ )
{
    currentSprite = arraySprite[i];
    currentSprite.x = Math.random()*stage.stageWidth;
    currentSprite.y = Math.random()*stage.stageHeight;
    currentSprite.alpha = Math.random();
    currentSprite.rotation = Math.random()*360;
}

trace( getTimer() - started );
// output : 9
```

По возможности встраивайте код для уменьшения числа вызовов функций в коде.

Вызов функций может быть ресурсоемким. Попробуйте уменьшить число вызовов функций путем встраивания кода. Встраивание кода является хорошим способом оптимизации абсолютных показателей производительности. Однако помните, что при встраивании код становится более сложным для повторного использования и может происходить увеличение размера SWF-файла. Некоторые вызовы функций, например методы класса Math, можно встроить без труда. В следующем коде метод `Math.abs()` используется для вычисления абсолютных значений.

```
const MAX_NUM:int = 500000;
var arrayValues:Vector.<Number>=new Vector.<Number>(MAX_NUM,true);
var i:int;

for (i = 0; i< MAX_NUM; i++)
{
    arrayValues[i] = Math.random()-Math.random();
}

var started:Number = getTimer();
var currentValue:Number;

for (i = 0; i< MAX_NUM; i++)
{
    currentValue = arrayValues[i];
    arrayValues[i] = Math.abs ( currentValue );
}

trace( getTimer() - started );
// output : 70
```

Вычисление с помощью метода `Math.abs()` можно выполнить вручную и встроить.

```
const MAX_NUM:int = 500000;
var arrayValues:Vector.<Number>=new Vector.<Number>(MAX_NUM,true);
var i:int;

for (i = 0; i< MAX_NUM; i++)
{
    arrayValues[i] = Math.random()-Math.random();
}

var started:Number = getTimer();
var currentValue:Number;

for (i = 0; i< MAX_NUM; i++)
{
    currentValue = arrayValues[i];
    arrayValues[i] = currentValue > 0 ? currentValue : -currentValue;
}

trace( getTimer() - started );
// output : 15
```

При встраивании вызова функции создается код, который выполняется более чем в четыре раза быстрее. Этот подход полезен во многих случаях, но помните, что он может влиять на пригодность для повторного использования и удобство сопровождения кода.

Примечание. *Размер кода оказывает существенное влияние на работу проигрывателя в целом. Если в приложении значительные объемы кода ActionScript, виртуальная машина потратит много времени на его проверку и JIT-компиляцию. Поиск свойств может выполняться медленнее из-за большого количества уровней иерархии и из-за того, что на обработку внутреннего кэша тратится неоправданно много ресурсов. Для сокращения размера кода не используйте среду Adobe® Flex®, библиотеку среды TLF и другие объемные библиотеки ActionScript от независимых разработчиков.*


Не оценивайте операторы в цикле.

Еще одним приемом оптимизации является отказ от оценки оператора в каждом цикле. Следующий код выполняет итерации в массиве, но не является оптимизированным, поскольку длина массива оценивается в каждой итерации.

```
for (var i:int = 0; i< myArray.length; i++)
{
}
```

Лучше сохранить значение и использовать его повторно.

```
var lng:int = myArray.length;

for (var i:int = 0; i< lng; i++)
{
}
```


Используйте обратный порядок для циклов while.

В обратном порядке циклы while выполняются быстрее, чем в прямом.

```
var i:int = myArray.length;

while (--i > -1)
{
}
```

Это лишь некоторые примеры оптимизации кода ActionScript, наглядно демонстрирующие, что даже одна строка может серьезно влиять на производительность и использование памяти. Возможны и многие другие способы оптимизации кода ActionScript. Дополнительные сведения см. на странице http://www.rozengain.com/blog/2007/05/01/some-actionscript-30-optimizations/.

Глава 5. Производительность визуализации

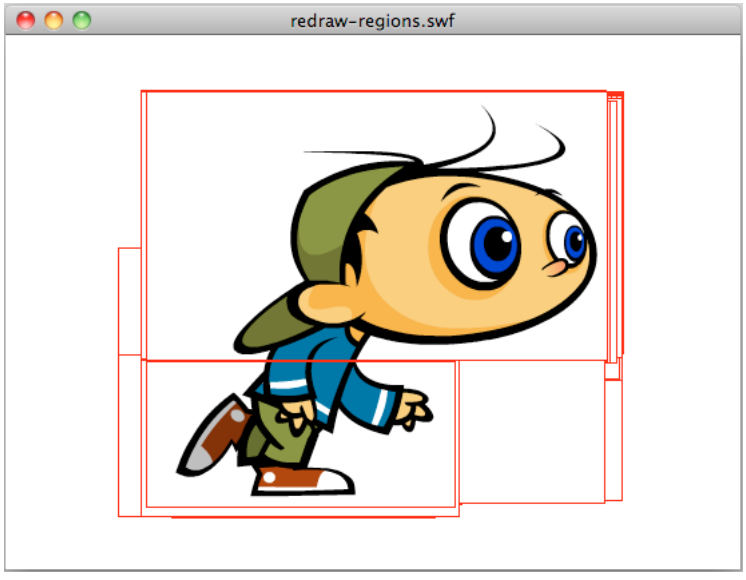
Области перерисовки

Всегда используйте параметр областей перерисовки при создании проекта.

Для повышения производительности визуализации при создании проекта важно использовать параметр областей перерисовки. С помощью этого параметра можно увидеть области, визуализируемые и обрабатываемые проигрывателем Flash Player. Этот параметр можно включить, выбрав команду «Показать области перерисовки» в контекстном меню отладочной версии Flash Player.

Примечание. *Команда «Показать области перерисовки» не доступна в Adobe AIR и в окончательной версии Flash Player (контекстное меню имеется только в приложениях Adobe AIR для настольных систем, однако оно не содержит встроенных или стандартных элементов, таких как «Показать области перерисовки»).*

На рисунке ниже показан включенные параметр с простым анимированным фрагментом ролика на временной шкале:



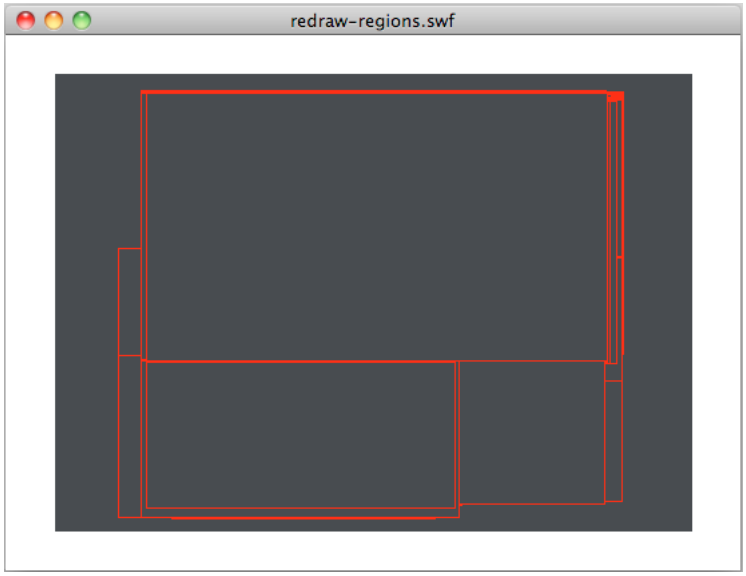
Параметр областей перерисовки включен

Этот параметр можно также включить программным способом с помощью метода `flash.profiler.showRedrawRegions()`:

```
// Enable Show Redraw Regions
// Blue color is used to show redrawn regions
flash.profiler.showRedrawRegions ( true, 0x0000FF );
```

В приложениях Adobe AIR это единственный способ включения параметра областей перерисовки.

Области перерисовки позволяют выявлять возможности для оптимизации. Помните, что хотя некоторые экранные объекты не показываются, они по-прежнему используют циклы ЦП, поскольку выполняется их визуализация. На следующем рисунке показана эта идея. Черная векторная фигура закрывает анимированную движущуюся фигуру. На рисунке показано, что экранный объект не был удален из списка отображения, поэтому его визуализация выполняется по-прежнему. Это приводит к неэффективному использованию циклов ЦП:



Перерисованные области

В целях повышения производительности для свойства скрытого выполнения `visible` установите значение `false` или удалите его из списка отображения. Также следует остановить его временную шкалу. Эти действия позволяют зафиксировать экранный объект, чтобы он использовал минимальное количество ресурсов ЦП.

Обязательно используйте параметр областей перерисовки в течение всего цикла разработки. Использование этого параметра позволяет предотвратить появление неожиданных результатов в конце проекта из-за ненужных областей перерисовки и пропущенных областей оптимизации.

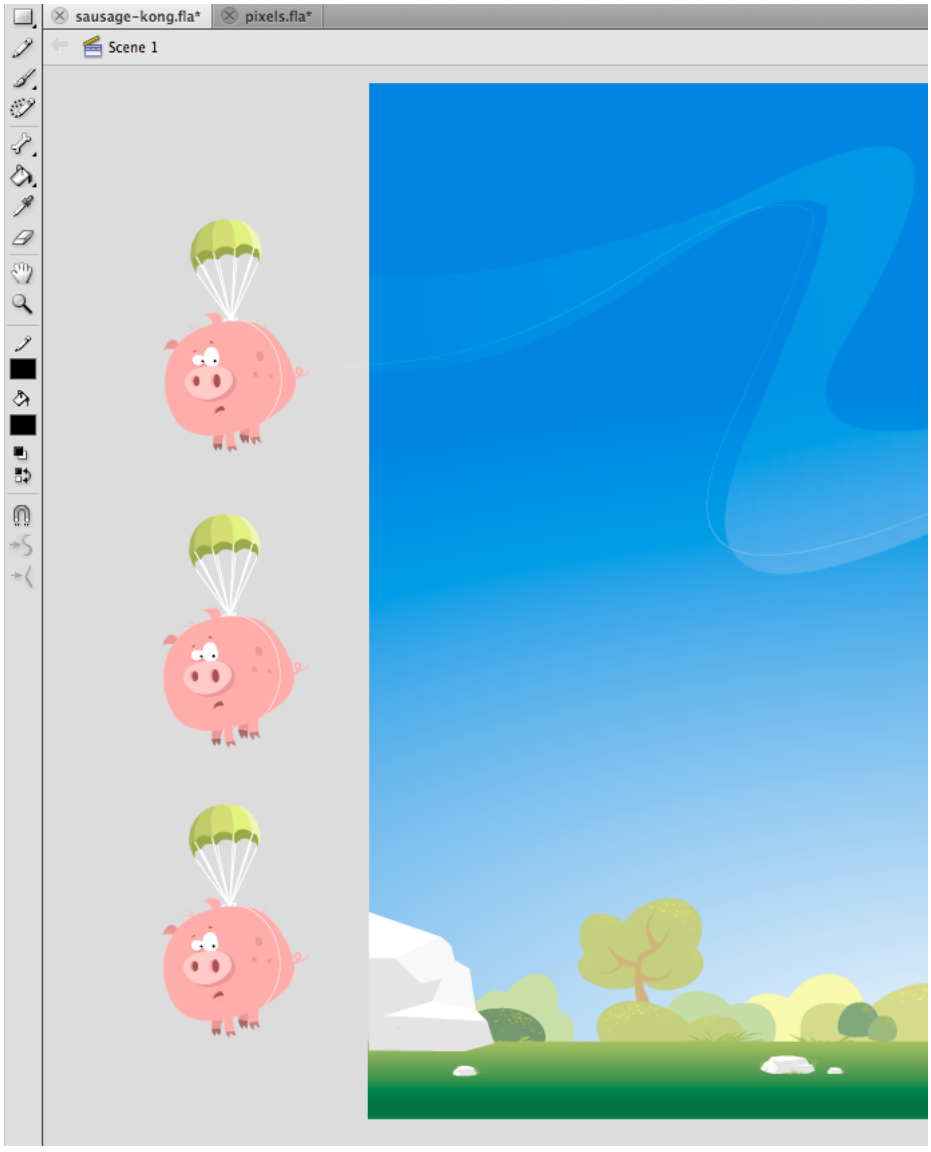
Дополнительные разделы справки

«Замораживание и размораживание объектов» на странице 30

Содержимое за границами рабочей области

Старайтесь не помещать содержимое за границами рабочей области. Вместо этого при необходимости просто поместите объекты в список отображения.

По возможности не размещайте графическое содержимое за границами рабочей области. Для дизайнеров и разработчиков обычной практикой является размещение элементов за границами рабочей области для повторного использования активов во время работы приложения. На следующем рисунке проиллюстрирован данный метод.



Содержимое за границами рабочей области

Элементы вне рабочей области, которые не отображаются на экране и для которых не выполняется визуализация, тем не менее находятся в списке отображения. Среда выполнения продолжает проводить внутренние тесты этих элементов в целях проверки того, что они все еще находятся за границами рабочей области, и пользователь не может взаимодействовать с ними. Таким образом, по мере возможности рекомендуется избегать размещения объектов за границами рабочей области и вместо этого удалить их из списка отображения.

Качество ролика

Используйте соответствующий параметр качества рабочей области для повышения качества визуализации.

При разработке содержимого для мобильных устройств с небольшими экранами, например телефонов, качество изображения играет менее важную роль, чем при разработке приложений для настольных систем. Настройка соответствующего параметра для качества рабочей области позволяет повысить производительность визуализации.

Для качества рабочей области доступны следующие параметры.

- `StageQuality.LOW`: поддерживает скорость воспроизведения за счет внешнего вида и не использует сглаживание. Данный параметр не поддерживается в Adobe AIR для настольных систем и ТВ.
- `StageQuality.MEDIUM`: применяет некоторое сглаживание, но не для масштабированных растровых изображений. Данный параметр является значением по умолчанию среды AIR для мобильных устройств, но не поддерживается в AIR для настольных компьютеров и ТВ-устройств.
- `StageQuality.HIGH`: поддерживает внешний вид за счет скорости воспроизведения и всегда использует сглаживание (по умолчанию в приложениях для настольных компьютеров). Если SWF-файл не содержит анимации, то масштабированные битовые изображения смягчаются. В противном случае растровые изображения не смягчаются.
- `StageQuality.BEST`: обеспечивает самое высокое качество отображения, даже за счет скорости. Смягчается весь вывод, в том числе масштабированные растровые изображения.

Параметр `StageQuality.MEDIUM` часто обеспечивает приемлемое качество для приложений для мобильных устройств, а в некоторых случаях приемлемое качество можно получить с помощью параметра `StageQuality.LOW`. Начиная с версии Flash Player 8, доступна возможность точной визуализации сглаженного текста, даже если для качества рабочей области задано значение `LOW`.

Примечание. *В некоторых мобильных устройствах, даже если для качества задано значение `HIGH`, для повышения производительности в приложениях Flash Player используется значение `MEDIUM`. Однако зачастую при настройке для качества значения HIGH разница незаметна, поскольку экраны мобильных устройств обычно имеют более высокое разрешение. (Разрешение может изменяться в зависимости от устройства.)*

На следующем рисунке для качества ролика задано значение `MEDIUM`, а для визуализации текста — «Сглаживание для анимации».

Среднее качество рабочей области и визуализация текста с параметром «Сглаживание для анимации»

Качество текста ухудшается из-за параметра качества рабочей области, поскольку соответствующий параметр визуализации текста не используется.

Среда выполнения позволяет установить для визуализации текста параметр «Сглаживание для читаемости». Этот параметр обеспечивает отличное качество текста (со сглаживанием) вне зависимости от используемого параметра качества рабочей области.

Низкое качество рабочей области и визуализация текста с параметром «Сглаживание для читаемости»

Такое же качество визуализации можно получить путем задания для визуализации текста параметра «Растровый текст» (без сглаживания).

Низкое качество рабочей области и визуализация текста с параметром «Растровый текст» (без сглаживания)

Последние два примера показывают, что качественный текст можно получить вне зависимости от используемого параметра качества рабочей области. Эта функция была представлена во Flash Player 8 и может использоваться на мобильных устройствах. Имейте в виду, что на некоторых устройствах Flash Player 10.1 автоматически выбирает качество `StageQuality.MEDIUM` для оптимизации производительности.

Наложение альфа-канала

Старайтесь по возможности не использовать свойство alpha.

Не используйте эффекты с наложением альфа-каналов, такие как постепенное появление, если используется свойство `alpha`. Когда отображаемый объект использует наложение альфа-каналов, для определения конечного цвета среда выполнения должна объединять значение цвета всех наложенных отображаемых объектов с цветом фона. Таким образом, для применения эффекта наложения альфа-каналов может требоваться больше ресурсов процессора, чем для отображения непрозрачного цвета. Дополнительные вычисления могут снизить производительность устройств с небольшой мощностью. Старайтесь по возможности не использовать свойство `alpha`.

Дополнительные разделы справки

«Кэширование растрового изображения» на странице 57

«Визуализация текстовых объектов» на странице 70

Частота кадров приложения

Общий совет: для увеличения производительности используйте минимально возможную частоту кадров.

Частота кадров приложения определяет время каждого цикла для выполнения кода приложения и визуализации, как описано в разделе ««Основные сведения по выполнению кода в среде выполнения» на странице 1». При более высокой частоте кадров создается более плавная анимация. Но, если анимация или другие визуальные изменения не происходят, часто отсутствует причина использовать высокую частоту кадров. При более высокой частоте кадров затрачивается больше циклов ЦП и заряда батареи, чем при низкой частоте.

Ниже приведены общие рекомендации по использованию подходящей частоты кадров по умолчанию для различных приложений.

- При использовании инфраструктуры Flex оставьте значение начальной частоты кадров по умолчанию без изменений.
- Если приложение включает анимацию, приемлемая частота кадров составляет не менее 20 кадров в секунду. Устанавливать частоту кадров, превышающую 30 кадров в секунду, зачастую нет необходимости.
- Если в приложении нет анимации, то, возможно, частоты 12 кадров в секунду будет вполне достаточно.

«Минимально возможная частота кадров» сильно зависит от выполняемых приложением действий. Дополнительные сведения см. в следующем совете «Динамическое изменение частоты кадров приложения».

Используйте низкую частоту кадров, если единственным динамическим содержимым в приложении является видео.

Среда выполнения воспроизводит загруженное видео с его собственной частотой кадров, не принимая во внимание частоту кадров приложения. Если в приложении нет анимации или динамического визуального содержимого, уменьшение частоты кадров не приводит к снижению производительности пользовательского интерфейса.

Динамическое изменение частоты кадров приложения.

Задать начальную частоту кадров приложения в проекте можно в параметрах компилятора, однако это значение можно сделать динамически изменяемым. Для этого необходимо настроить во Flex свойство `Stage.frameRate` (или свойство `WindowedApplication.frameRate`).

Изменяйте частоту кадров в соответствии с текущими потребностями приложения. Например, когда приложению не требуется выполнять рендеринг анимации, понизьте частоту кадров. И наоборот, в начале анимации поднимите частоту кадров. Аналогично, если приложение выполняется в фоновом режиме (после потери фокуса), обычно можно еще больше уменьшить частоту кадров. Вероятно, пользователь будет уделять основное внимание другому приложению или другой задаче.

Ниже приведены общие рекомендации, которые можно использовать в качестве отправной точки при определении подходящей частоты кадров для различных типов действий.

- При использовании инфраструктуры Flex оставьте значение начальной частоты кадров по умолчанию без изменений.
- При воспроизведении анимации частота кадров должна составлять не менее 20 кадров в секунду. Устанавливать частоту кадров, превышающую 30 кадров в секунду, зачастую нет необходимости.
- Если приложение не воспроизводит анимацию, то, возможно, частоты 12 кадров в секунду будет вполне достаточно.
- Загруженное видео воспроизводится с собственной частотой кадров, не принимая во внимание частоту кадров приложения. Если из динамического содержимого в приложении присутствует только видео, возможно, частоты 12 кадров в секунду будет вполне достаточно.
- Если приложение не находится в фокусе, возможно, частоты 5 кадров в секунду будет вполне достаточно.
- Когда приложение AIR не отображается, подходящая частота кадров составляет 2 кадра в секунду или менее. Например, данные рекомендации применяются, когда приложение свернуто. Они также применяются на настольных устройствах, если для свойства окна `visible` установлено значение `false`.

Приложения, созданные во Flex; компоненты spark. В класс WindowedApplication встроена поддержка динамически изменяющейся частоты кадров приложения. Частоту кадров неактивного приложения определяет свойство `backgroundFrameRate`. Значением по умолчанию является 1, при этом частота кадров приложения, созданного с использованием среды Spark, изменяется до 1 кадра в секунду. Изменить фоновую частоту кадров можно, задав свойство `backgroundFrameRate`. Можно задать для свойства другое значение или значение -1, чтобы отключить автоматическое регулирование частоты кадров.

Дополнительные сведения о динамическом изменении частоты кадров приложения см. в следующих статьях.

- Снижение нагрузки на ЦП в Adobe AIR (Статья разработчика Adobe Developer Center, Джонни Холлмана (Jonnie Hallman))
- Создание стабильных и производительных приложений AIR (статья и пример приложения Арно Гурдола (Arno Gourdol))

Грант Скиннер (Grant Skinner) создал класс регулировки частоты кадров. Этот класс можно использовать в приложениях для автоматического уменьшения частоты кадров во время работы приложения в фоновом режиме. Получить дополнительные сведения и загрузить исходный код для класса FramerateThrottler можно в статье Гранта «Использование бездействующего ЦП в Adobe AIR и Flash Player» на http://gskinner.com/blog/archives/2009/05/idle_cpu_usage.html.

Адаптивная частота кадров

При компиляции SWF-файла задается определенная частота кадров ролика. В средах с ограниченными ресурсами и низкой производительностью ЦП частота кадров иногда уменьшается во время воспроизведения. Для сохранения приемлемой частоты кадров среда выполнения пропускает визуализацию некоторых кадров. Пропуск визуализации некоторых кадров позволяет предотвратить падение частоты кадров ниже приемлемого значения.

Примечание. *В этом случае среда выполнения не пропускает кадры, она пропускает только визуализацию содержимого кадров. Код по-прежнему выполняется и список отображения обновляется, однако обновления не отображаются на экране. Способ задания порогового значения частоты кадров, обозначающего число кадров, пропускаемых средой выполнения в случае невозможности сохранения стабильной частоты кадров, отсутствует.*

Кэширование растрового изображения

Если необходимо, используйте функцию кэширования растровых изображений для сложного векторного содержимого.

Хорошей оптимизации можно достичь с помощью функции кэширования растровых изображений. Эта функция кэширует векторный объект, визуализирует его как растровое изображение при внутренней обработке и использует это растровое изображение для визуализации. Она позволяет получить огромное повышение производительности визуализации, однако может требовать использования значительного объема памяти. Используйте функцию кэширования растрового изображения для сложного векторного содержимого, такого как сложные градиенты или текст.

Производительность повышается, если включить кэширование растровых изображений для анимированного объекта со сложной векторной графикой, такого как текст или градиенты. Однако, если кэширование растрового изображения включено для экранного объекта, такого как фрагмент ролика, для которого воспроизводится временная шкала, получается противоположный результат. В каждом кадре среда выполнения должна обновлять кэшированное растровое изображение, а затем перерисовывать его на экране, а для этого требуется большое число циклов ЦП. Функция кэширования растровых изображений способствует оптимизации, только если кэшированное растровое создается единожды и более не обновляется.

Если включить кэширование растровых изображений для спрайтов, при их перемещении среда выполнения не будет повторно генерировать кэшированные растровые изображения. Изменение свойств `x` и `y` объекта не требует повторной генерации. Однако, если объекты будут вращаться, масштабироваться или будет меняться альфа-коэффициент, среда выполнения будет повторного генерировать кэшированные растровые изображения, что снизит производительность.

Примечание. *Свойство `DisplayObject.cacheAsBitmapMatrix`, доступное в среде AIR и инструменте Packager for iPhone, не подвержено этим ограничениям. Используя свойство `cacheAsBitmapMatrix`, можно поворачивать, масштабировать, наклонять и изменять альфа-коэффициент без повторной регенерации растровых изображений.*

Кэшированные растровые изображения могут занимать больше памяти, чем обычный экземпляр фрагмента ролика. Например, если фрагмент ролика в рабочей области имеет размер 250 x 250 пикселов, при кэшировании он занимает около 250 КБ (по сравнению с 1 КБ в некэшированном состоянии).

В следующем примере показан объект Sprite, содержащий изображение яблока. Следующий класс соединен с символом яблока.

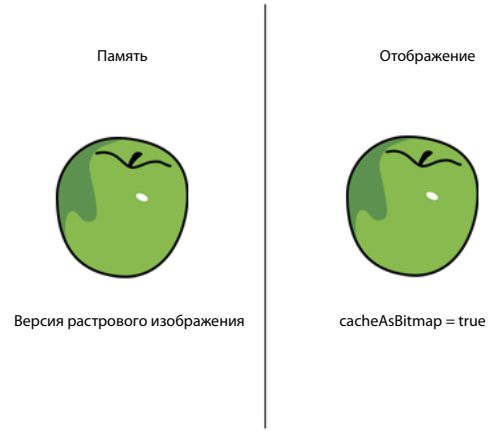
```
package org.bytearray.bitmap
{
    import flash.display.Sprite;
    import flash.events.Event;

    public class Apple extends Sprite
    {
        private var destinationX:Number;
        private var destinationY:Number;

        public function Apple ()
        {
            addEventListener(Event.ADDED_TO_STAGE,activation);
            addEventListener(Event.REMOVED_FROM_STAGE,deactivation);
        }

        private function activation(e:Event):void
        {
            initPos();
            addEventListener (Event.ENTER_FRAME,handleMovement);
        }

        private function deactivation(e:Event):void
        {
            removeEventListener(Event.ENTER_FRAME,handleMovement);
        }

        private function initPos():void
        {
            destinationX = Math.random()*(stage.stageWidth - (width>>1));
            destinationY = Math.random()*(stage.stageHeight - (height>>1));
        }

        private function handleMovement(e:Event):void
        {
            x -= (x - destinationX)*.5;
            y -= (y - destinationY)*.5;

            if (Math.abs(x - destinationX) < 1 && Math.abs(y - destinationY) < 1)
                initPos();
        }
    }
}
```

В этом коде вместо класса MovieClip используется класс Sprite, поскольку временная шкала не требуется для каждого яблока. Для получения наиболее оптимальной производительности по возможности используйте самый легковесный объект. Затем экземпляры класса создаются с помощью следующего кода.

```
import org.bytearray.bitmap.Apple;

stage.addEventListener(MouseEvent.CLICK,createApples);
stage.addEventListener(KeyboardEvent.KEY_DOWN,cacheApples);

const MAX_NUM:int = 100;
var apple:Apple;
var holder:Sprite = new Sprite();

addChild(holder);

function createApples(e:MouseEvent):void
{
    for (var i:int = 0; i< MAX_NUM; i++)
    {
        apple = new Apple();

        holder.addChild(apple);
    }
}

function cacheApples(e:KeyboardEvent):void
{
    if (e.keyCode == 67)
    {
        var lng:int = holder.numChildren;

        for (var i:int = 0; i < lng; i++)
        {
            apple = holder.getChildAt (i) as Apple;

            apple.cacheAsBitmap = Boolean(!apple.cacheAsBitmap);
        }
    }
}
```

Когда пользователь нажимает кнопку мыши, изображения яблок создаются без кэширования. Когда пользователь нажимает клавишу C (код клавиши 67), векторы яблок кэшируются в качестве растровых изображений и показываются на экране. Этот прием позволяет значительно повысить производительность визуализации как в настольных системах, так и в мобильных устройствах с низкой производительностью процессора.

И хотя использование функции кэширования растрового изображения позволяет повысить производительность визуализации, при этом могут использоваться большие объемы памяти. При кэшировании объекта его поверхность захватывается в качестве прозрачного растрового изображения и сохраняется в памяти, как показано на следующей схеме.

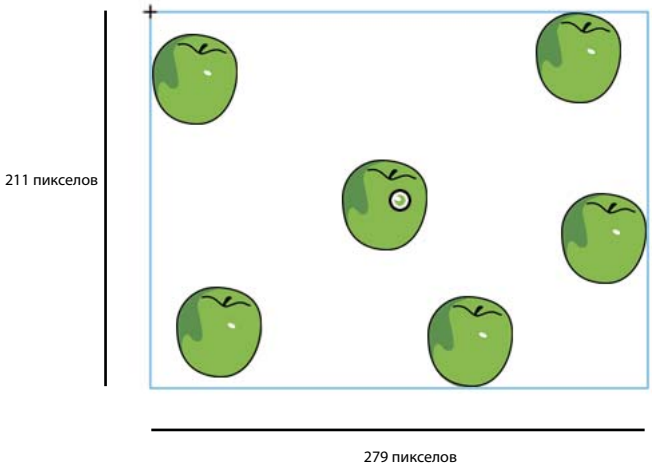


Объект и растровое изображение его поверхности, сохраненное в памяти

Flash Player 10.1 и AIR 2.5 оптимизируют использование памяти за счет подхода, описанного в разделе ««Фильтры и динамическая выгрузка растровых изображений» на странице 20». Если кэшированный экранный объект скрыт или находится вне экрана, а его растровое изображение, сохраненное в памяти, какое-то время не используется, оно удаляется из памяти.

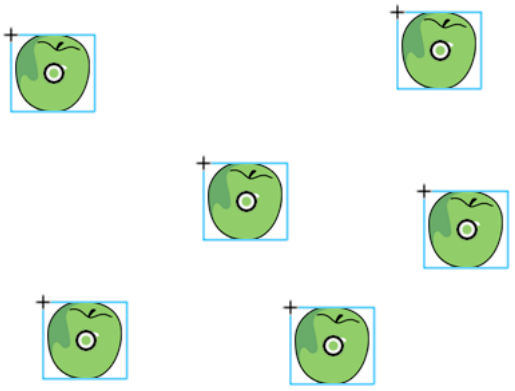
Примечание. *Если для свойства* `opaqueBackground` *экранного объекта задан определенный цвет, в среде выполнения экранный объект считается непрозрачным. Свойство* `cacheAsBitmap` *заставляет среду выполнения создавать в памяти непрозрачное 32-битное растровое изображение. Для альфа-канала задано значение 0xFF, что позволяет повысить производительность, поскольку для рисования объекта на экране не требуется прозрачность. Предотвращение наложения альфа-канала позволяет выполнить визуализацию еще быстрее. Если текущая глубина экрана ограничена 16 битами, растровое изображение хранится в памяти как 16-битное изображение. Свойство* `opaqueBackground` *напрямую не вызывает кэширование растровых изображений.*

В целях экономии памяти используйте свойство `cacheAsBitmap` и активируйте его для каждого экранного объекта, а не контейнера. При активации кэширования растрового изображения в контейнере конечное изображение занимает гораздо больше памяти, поскольку прозрачное растровое изображение создается с размерами 211 x 279 пикселов. Изображение занимает около 229 КБ памяти:



Активация кэширования растрового изображения в контейнере

Кроме того, если какое-либо яблоко в кадре будет двигаться, кэширование контейнера приведет к обновлению в памяти растрового изображения целиком. Активация кэширования растрового изображения в отдельных экземплярах приводит к кэшированию шести поверхностей размером 7 КБ в памяти, для этого требуется всего 42 КБ памяти:



Активация кэширования растрового изображения в экземплярах

При доступе к каждому экземпляру яблока через список отображения и вызове метода `getChildAt()` ссылки сохраняются в объекте Vector для упрощения доступа.

```
import org.bytearray.bitmap.Apple;

stage.addEventListener(KeyboardEvent.KEY_DOWN, cacheApples);

const MAX_NUM:int = 200;
var apple:Apple;
var holder:Sprite = new Sprite();

addChild(holder);

var holderVector:Vector.<Apple> = new Vector.<Apple>(MAX_NUM, true);

for (var i:int = 0; i< MAX_NUM; i++)
{
    apple = new Apple();

    holder.addChild(apple);

    holderVector[i] = apple;
}

function cacheApples(e:KeyboardEvent):void
{
    if (e.keyCode == 67)
    {
        var lng:int = holderVector.length

        for (var i:int = 0; i < lng; i++)
        {
            apple = holderVector[i];

            apple.cacheAsBitmap = Boolean(!apple.cacheAsBitmap);
        }
    }
}
```

Помните, что кэширование растровых изображений ускоряет визуализацию, только если содержимое не вращается, не масштабируется и не изменяется в каждом кадре. Однако для любого преобразования, отличного от переноса вдоль осей X и Y, качество визуализации не улучшается. В этих случаях проигрыватель Flash Player обновляет копию кэшированного растрового изображения при каждом преобразовании экранного объекта. Обновление кэшированной копии может приводить к увеличению загрузки ЦП, снижению производительности и повышению уровня использования аккумулятора. И снова это ограничение не распространяется на свойство `cacheAsBitmapMatrix`, доступное в среде AIR или инструменте Packager for iPhone.

В следующем коде изменяется альфа-коэффициент в методе перемещения, что приводит к изменению непрозрачности яблока в каждом кадре.

```
private function handleMovement(e:Event):void
{
        alpha = Math.random();
        x -= (x - destinationX)*.5;
        y -= (y - destinationY)*.5;

        if (Math.abs(x - destinationX) < 1 && Math.abs(y - destinationY) < 1)
            initPos();
}
```

Использование кэширования растрового изображение влечет снижение производительности. При изменении альфа-коэффициента среда выполнения обновляет кэшированное растровое изображение в памяти.

Фильтры зависят от растровых изображений, обновляемых при перемещении точки воспроизведения кэшированного фрагмента ролика. Поэтому при использовании фильтра для свойства `cacheAsBitmap` автоматически задается значение `true`. Ниже представлен пример с анимированным фрагментом ролика.

Фрагмент ролика с анимацией

Избегайте использования фильтров для анимированного содержимого, поскольку они могут вызывать проблемы с производительностью. На следующем рисунке дизайнер добавляет фильтр тени.

Фрагмент ролика с анимацией и фильтром «Тень»

В результате, если временная шкала сдвигается (т. е. клип воспроизводится), растровое изображение необходимо генерировать повторно. Кроме того, его требуется повторно генерировать в каждом случае, когда содержимое изменяется любым образом, кроме простого перемещения по осям x и y. Среда выполнения должна перерисовывать растровое изображение в каждом кадре, что приводит к использованию дополнительных ресурсов ЦП, снижению производительности и повышению уровня использования аккумулятора.

Пол Трани (Paul Trani) приводит примеры использования Flash Professional и ActionScript для оптимизации графики с использованием растровых изображений в следующих обучающих видеороликах:

- Оптимизация графики
- Оптимизация графики с помощью ActionScript

Матрица преобразования кэшированных растровых изображений в AIR

При использовании кэшированных изображений в приложениях AIR для мобильных устройств установите свойство `cacheAsBitmapMatrix`.

В профиле AIR на мобильных устройствах для свойства `cacheAsBitmapMatrix` можно назначить объект «Матрица». С помощью этого свойства можно применять любое двумерное преобразование для объекта без повторной регенерации кэшированных растровых изображений. Также можно изменять свойство alpha без повторной регенерации кэшированных изображений. Для свойства `cacheAsBitmap` должно быть установлено значение `true`, и для объекта не должно быть задано никаких свойств 3D.

При установке свойства `cacheAsBitmapMatrix` генерируется кэшированное растровое изображение, даже если отображаемый объект находится за пределами экрана, скрыт или имеет свойство `visible`, для которого установлено значение `false`. В результате сброса свойства `cacheAsBitmapMatrix` с использованием объекта матрицы, содержащего другую трансформацию, также выполняется повторная генерация кэшированного растрового изображения.

Матричное преобразование, применяемое к свойству `cacheAsBitmapMatrix`, распространяется на отображаемый объект, который визуализируется в кэше растровых изображений. Таким образом, если преобразование содержит двукратное масштабирование, растровое изображение имеет размер в два раза больше векторного изображения. При визуализации к кэшированному растровому изображению применяется обратное преобразование, чтобы итоговое изображение выглядело таким же образом. Кэшированные растровые изображения можно уменьшить в размере, чтобы снизить объем используемой памяти, возможно за счет точности визуализации. Кроме того, в некоторых случаях размер растрового изображения можно увеличить, чтобы повысить качество визуализации за счет увеличения объема используемой памяти. Однако в общем случае рекомендуется использовать матрицу тождественности, которая снимает все преобразования, чтобы избежать изменения при отображении, как показано на следующем примере.

```
displayObject.cacheAsBitMap = true;
displayObject.cacheAsBitmapMatrix = new Matrix();
```

После установки свойства `cacheAsBitmapMatrix` над объектом можно будет выполнять операции масштабирования, наклона, поворота и переноса без повторной генерации растрового изображения.

Также можно изменять альфа-коэффициент в диапазоне от 0 до 1. Если изменение альфа-коэффициента выполняется с помощью свойства `transform.colorTransform` с преобразованием цвета, альфа-коэффициент, используемый для преобразуемого объекта, должен иметь значение от 0 до 255. При изменении цветового преобразования любым другим способом происходит повторная генерация кэшированного растрового изображения.

В случаях когда для содержимого, предназначенного для мобильных устройств, для параметра `cacheAsBitmap` установлено значение `true`, следует задавать свойство `cacheAsBitmapMatrix`. Однако при этом необходимо учитывать следующий возможный недостаток. После того как для объекта был применен поворот, масштабирование или наклон, итоговая визуализация может содержать признаки масштабирования или сглаживания растрового изображения в отличии от обычной векторной визуализации.

Кэширование растрового изображения вручную

Используйте класс BitmapData для создания пользовательского поведения кэширования растрового изображения.

В следующем примере повторно используется одна растрированная версия экранного объекта и выполняется обращение к одному объекту BitmapData. При масштабировании каждого экранного объекта исходный объект BitmapData в памяти не обновляется и не перерисовывается. Этот подход позволяет сэкономить ресурсы ЦП и способствует более быстрому выполнению приложения. При масштабировании экранного объекта растровое изображение в контейнере растягивается.

Здесь приводится обновленный класс `BitmapApple`.

```
package org.bytearray.bitmap
{
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.events.Event;

    public class BitmapApple extends Bitmap
    {
        private var destinationX:Number;
        private var destinationY:Number;

        public function BitmapApple(buffer:BitmapData)
        {
            super(buffer);

            addEventListener(Event.ADDED_TO_STAGE,activation);
            addEventListener(Event.REMOVED_FROM_STAGE,deactivation);
        }

        private function activation(e:Event):void
        {
            initPos();
            addEventListener(Event.ENTER_FRAME,handleMovement);
        }

        private function deactivation(e:Event):void
        {
            removeEventListener(Event.ENTER_FRAME,handleMovement);
        }

        private function initPos():void
        {
            destinationX = Math.random()*(stage.stageWidth - (width>>1));
            destinationY = Math.random()*(stage.stageHeight - (height>>1));
        }

        private function handleMovement(e:Event):void
        {
            alpha = Math.random();

            x -= (x - destinationX)*.5;
            y -= (y - destinationY)*.5;

            if ( Math.abs(x - destinationX) < 1 && Math.abs(y - destinationY) < 1)
                initPos();
        }
    }
}
```

Альфа-коэффициент все так же изменяется в каждом кадре. Следующий код передает исходный буфер источника каждому экземпляру BitmapApple.

```
import org.bytearray.bitmap.BitmapApple;

const MAX_NUM:int = 100;
var holder:Sprite = new Sprite();

addChild(holder);

var holderVector:Vector.<BitmapApple> = new Vector.<BitmapApple>(MAX_NUM, true);
var source:AppleSource = new AppleSource();
var bounds:Object = source.getBounds(source);

var mat:Matrix = new Matrix();
mat.translate(-bounds.x,-bounds.y);

var buffer:BitmapData = new BitmapData(source.width+1, source.height+1, true, 0);
buffer.draw(source,mat);

var bitmapApple:BitmapApple;

for (var i:int = 0; i< MAX_NUM; i++)
{
    bitmapApple = new BitmapApple(buffer);

    holderVector[i] = bitmapApple;

    holder.addChild(bitmapApple);
}
```

Этот способ занимает мало памяти, потому что в памяти хранится единое кэшированное растровое изображение, которое используется всеми экземплярами `BitmapApple`. Помимо этого, несмотря на все изменения, вносимые в экземпляры `BitmapApple`, такие как изменение альфа-канала, поворот или масштабирование, исходное растровое изображение никогда не обновляется. Этот прием позволяет предотвратить снижение производительности.

Для получения сглаженного конечного растрового изображения задайте для свойства `smoothing` значение `true`:

```
public function BitmapApple(buffer:BitmapData)
{
    super (buffer);

    smoothing = true;

    addEventListener(Event.ADDED_TO_STAGE, activation);
    addEventListener(Event.REMOVED_FROM_STAGE, deactivation);
}
```

Настройка качества рабочей области также может повысить производительность. Задайте для качества рабочей области значение `HIGH` перед растрированием, а затем измените значение на `LOW`.

```
import org.bytearray.bitmap.BitmapApple;

const MAX_NUM:int = 100;
var holder:Sprite = new Sprite();

addChild ( holder );

var holderVector:Vector.<BitmapApple> = new Vector.<BitmapApple>(MAX_NUM, true);
var source:AppleSource = new AppleSource();
var bounds:Object = source.getBounds ( source );

var mat:Matrix = new Matrix();
mat.translate ( -bounds.x, -bounds.y );

var buffer:BitmapData = new BitmapData ( source.width+1, source.height+1, true, 0 );

stage.quality = StageQuality.HIGH;

buffer.draw ( source, mat );

stage.quality = StageQuality.LOW;

var bitmapApple:BitmapApple;

for (var i:int = 0; i< MAX_NUM; i++ )
{
    bitmapApple = new BitmapApple( buffer );

    holderVector[i] = bitmapApple;

    holder.addChild ( bitmapApple );
}
```

Изменение качества рабочей области до и после перевода векторного изображения в растровое — эффективный метод вывода на экран сглаженного содержимого. Его эффективность не зависит от финального качества рабочей области. Например, можно получить растровое изображение и текст со сглаживанием, даже если для качества рабочей области задано значение `LOW`. Этот прием нельзя использовать со свойством `cacheAsBitmap`. В этом случае при задании для качества рабочей области значения `LOW` обновляется качество вектора, а также поверхность растрового изображения в памяти и качество конечного изображения.

Изоляция поведений

По возможности изолируйте такие события, как `Event.ENTER_FRAME`, в едином обработчике.

Можно еще больше оптимизировать код путем изоляции события `Event.ENTER_FRAME` в классе `Apple` в одном обработчике. Этот прием экономит ресурсы ЦП. Ниже приведен пример, в котором класс BitmapApple уже не обрабатывает поведение движения.

```
package org.bytearray.bitmap
{
    import flash.display.Bitmap;
    import flash.display.BitmapData;

    public class BitmapApple extends Bitmap
    {
        private var destinationX:Number;
        private var destinationY:Number;

        public function BitmapApple(buffer:BitmapData)
        {
            super (buffer);

            smoothing = true;
        }
}
```

В коде ниже показано, как создаются экземпляры яблок, а их перемещения обрабатываются единым обработчиком.

```
import org.bytearray.bitmap.BitmapApple;

const MAX_NUM:int = 100;
var holder:Sprite = new Sprite();

addChild(holder);

var holderVector:Vector.<BitmapApple> = new Vector.<BitmapApple>(MAX_NUM, true);
var source:AppleSource = new AppleSource();
var bounds:Object = source.getBounds(source);

var mat:Matrix = new Matrix();
mat.translate(-bounds.x,-bounds.y);

stage.quality = StageQuality.BEST;

var buffer:BitmapData = new BitmapData(source.width+1,source.height+1, true,0);
buffer.draw(source,mat);

stage.quality = StageQuality.LOW;

var bitmapApple:BitmapApple;

for (var i:int = 0; i< MAX_NUM; i++)
{
    bitmapApple = new BitmapApple(buffer);

    bitmapApple.destinationX = Math.random()*stage.stageWidth;
    bitmapApple.destinationY = Math.random()*stage.stageHeight;

    holderVector[i] = bitmapApple;

    holder.addChild(bitmapApple);
}

stage.addEventListener(Event.ENTER_FRAME,onFrame);
```

```
var lng:int = holderVector.length

function onFrame(e:Event):void
{
        for (var i:int = 0; i < lng; i++)
        {
            bitmapApple = holderVector[i];
            bitmapApple.alpha = Math.random();

            bitmapApple.x -= (bitmapApple.x - bitmapApple.destinationX) *.5;
            bitmapApple.y -= (bitmapApple.y - bitmapApple.destinationY) *.5;

            if (Math.abs(bitmapApple.x - bitmapApple.destinationX ) < 1 &&
                Math.abs(bitmapApple.y - bitmapApple.destinationY ) < 1)
            {
                    bitmapApple.destinationX = Math.random()*stage.stageWidth;
                    bitmapApple.destinationY = Math.random()*stage.stageHeight;
            }
        }
}
```

В результате создается одно событие `Event.ENTER_FRAME` для обработки перемещения вместо использования 200 обработчиков для перемещения каждого яблока. Всю анимацию легко приостановить, что особенно полезно в играх.

Например, в простой игре можно использовать следующий обработчик.

```
stage.addEventListener(Event.ENTER_FRAME, updateGame);
function updateGame (e:Event):void
{
    gameEngine.update();
}
```

Следующим этапом является создание изображений яблок, реагирующих на события мыши или клавиатуры. Для этого потребуется изменить класс `BitmapApple`.

```
package org.bytearray.bitmap
{
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.Sprite;

    public class BitmapApple extends Sprite
    {
        public var destinationX:Number;
        public var destinationY:Number;
        private var container:Sprite;
        private var containerBitmap:Bitmap;

        public function BitmapApple(buffer:BitmapData)
        {
            container = new Sprite();
            containerBitmap = new Bitmap(buffer);
            containerBitmap.smoothing = true;
            container.addChild(containerBitmap);
            addChild(container);
        }
}
```

В результате создаются интерактивные экземпляры `BitmapApple`, похожие на обычные объекты Sprite. Однако экземпляры связаны с единым растровым изображением, которое не обновляется при изменении экранных объектов.

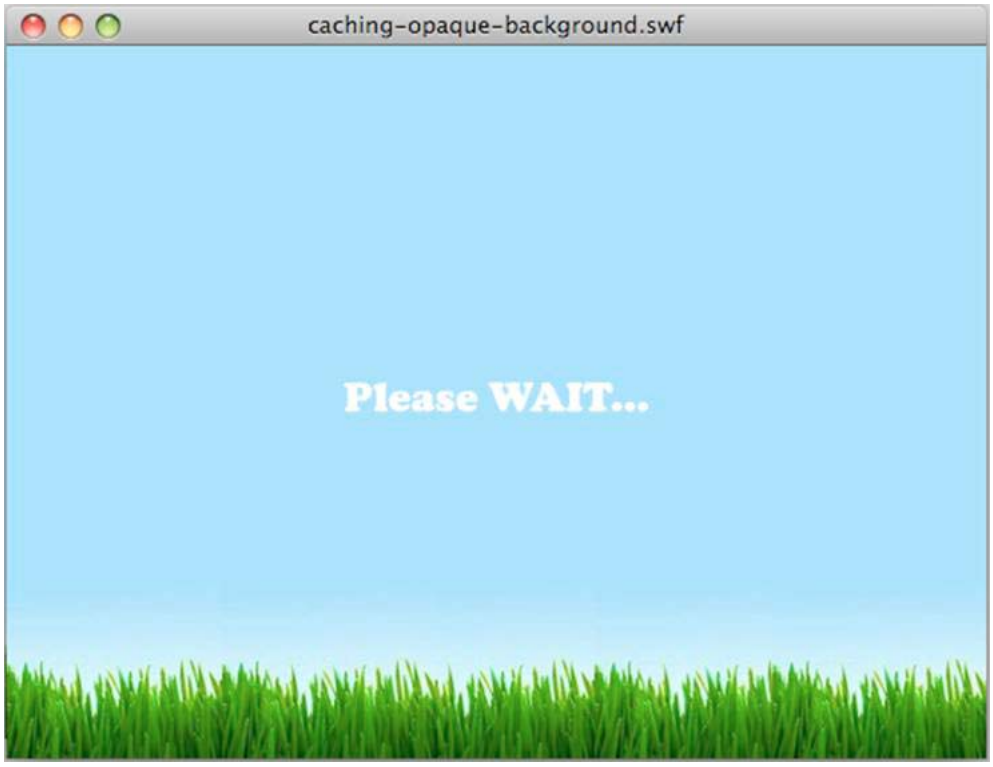
Визуализация текстовых объектов

Используйте функцию кэширования растрового изображения и свойство `opaqueBackground` для повышения производительности визуализации текста.

Механизм визуализации текста Flash Text Engine предоставляет отличные возможности для визуализации. Однако для многих классов требуется показывать одну строку текста. По этой причине для создания редактируемого текстового поля с использованием класса TextLine требуется значительный объем памяти и множество строк кода ActionScript. Класс TextLine лучше всего использовать для статического и нередактируемого текста, для которого визуализация выполняется быстрее и требуется меньше памяти.

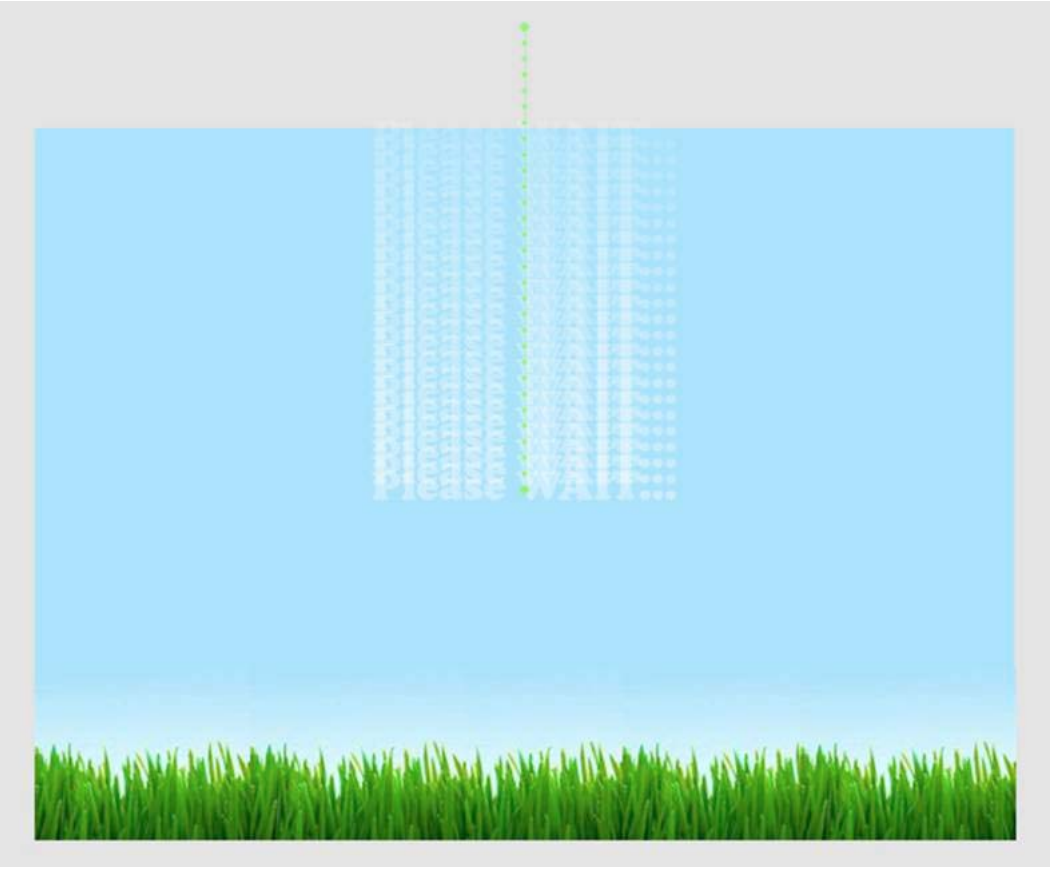
Функция кэширования растрового изображения позволяет кэшировать векторное содержимое в качестве растровых изображений для повышения производительности визуализации. Эта функция полезна для сложного векторного содержимого, а также при использовании с текстовым содержимым, для которого требуется визуализация обработки.

В следующем примере показано, как можно использовать функцию кэширования растрового изображения и свойство `opaqueBackground` для повышения производительности визуализации. На следующем рисунке показан стандартный экран приветствия, который может отображаться, когда пользователь ожидает загрузки содержимого.



Экран приветствия

На следующем рисунке показано замедление, которое применяется к объекту TextField программным способом. Текст замедляется от верха к центру монтажного кадра.



Замедление текста

Следующий код позволяет создать замедление. Переменная `preloader` сохраняет текущий целевой объект, снижая количество операций по поиску свойств, которые снижают производительность.

```
wait_mc.addEventListener( Event.ENTER_FRAME, movePosition );

var destX:Number=stage.stageWidth/2;
var destY:Number=stage.stageHeight/2;
var preloader:DisplayObject;

function movePosition( e:Event ):void
{
    preloader = e.currentTarget as DisplayObject;

    preloader.x -= ( preloader.x - destX ) * .1;
    preloader.y -= ( preloader.y - destY ) * .1;

    if (Math.abs(preloader.y-destY)<1)
        preloader.removeEventListener( Event.ENTER_FRAME, movePosition );
}
```

Функция `Math.abs()` может быть встроена для минимизации вызовов функции и повышения производительности. Лучшим приемом является использование типа int для свойств `destX` и `destY` для получения значений с фиксированной точкой. Использование типа int обеспечивает точную привязку к пикселам без необходимости округления значений вручную с помощью медленных методов, таких как `Math.ceil()` или `Math.round()`. Этот код не округляет значения координат до целых, потому что при постоянном округлении значений объект не будет двигаться плавно. Если значения координат округляются до ближайших целых, движения объекта будут неровными. Однако этот метод удобен для задания финального положения экранного объекта. Не используйте следующий код.

```
// Do not use this code
var destX:Number = Math.round ( stage.stageWidth / 2 );
var destY:Number = Math.round ( stage.stageHeight / 2);
```

Следующий код выполняется гораздо быстрее.

```
var destX:int = stage.stageWidth / 2;
var destY:int = stage.stageHeight / 2;
```

Предыдущий код был еще больше оптимизирован за счет использования операторов побитового смещения для деления значений.

```
var destX:int = stage.stageWidth >> 1;
var destY:int = stage.stageHeight >> 1;
```

Функция кэширования растрового изображения упрощает визуализацию объектов в среде выполнения за счет использования динамических растровых изображений. В текущем примере кэшируется фрагмент ролика, содержащий объект TextField.

```
wait_mc.cacheAsBitmap = true;
```

Одним из дополнительных способов повышения производительности является удаление альфа-прозрачности. Альфа-прозрачность повышает загрузку среды выполнения при рисовании прозрачных растровых изображений, как показано в предыдущем примере кода. В качестве обходного приема можно использовать свойство `opaqueBackground` путем указания цвета фона.

При использовании свойства `opaqueBackground` поверхность растрового изображения, создаваемая в памяти, по-прежнему занимает 32 бита. Однако для сдвига альфа задано значение 255 и прозрачность не применяется. В результате использование свойства `opaqueBackground` не приводит к уменьшению объема занимаемой памяти, но позволяет повысить производительность визуализации при применении функции кэширования растрового изображения. В следующем примере кода используются все приемы по оптимизации.

```
wait_mc.addEventListener( Event.ENTER_FRAME, movePosition );
wait_mc.cacheAsBitmap = true;

// Set the background to the color of the scene background
wait_mc.opaqueBackground = 0x8AD6FD;
var destX:int = stage.stageWidth >> 1;
var destY:int = stage.stageHeight >> 1;
var preloader:DisplayObject;

function movePosition ( e:Event ):void
{
    preloader = e.currentTarget as DisplayObject;

    preloader.x -= ( preloader.x - destX ) * .1;
    preloader.y -= ( preloader.y - destY ) * .1;

    if ( Math.abs ( preloader.y - destY ) < 1 )
        e.currentTarget.removeEventListener ( Event.ENTER_FRAME, movePosition );
}
```

Теперь анимация оптимизирована, а кэширование растрового изображения оптимизировано за счет удаления прозрачности. На мобильных устройствах при использовании функции кэширования растрового изображения рассмотрите возможность задания для качества рабочей области значений `LOW` и `HIGH` в различных состояниях анимации.

```
wait_mc.addEventListener( Event.ENTER_FRAME, movePosition );
wait_mc.cacheAsBitmap = true;
wait_mc.opaqueBackground = 0x8AD6FD;

// Switch to low quality
stage.quality = StageQuality.LOW;
var destX:int = stage.stageWidth>>1;
var destY:int = stage.stageHeight>>1;
var preloader:DisplayObject;

function movePosition( e:Event ):void
{
    preloader = e.currentTarget as DisplayObject;

    preloader.x -= ( preloader.x - destX ) * .1;
    preloader.y -= ( preloader.y - destY ) * .1;

    if (Math.abs(e.currentTarget.y-destY)<1)
    {
        // Switch back to high quality
        stage.quality = StageQuality.HIGH;
        preloader.removeEventListener( Event.ENTER_FRAME, movePosition );
    }
}
```

Однако в этом случае при изменении качества рабочей области среды выполнения повторно создает поверхность растрового изображения объекта TextField в соответствии с текущим качеством рабочей области. По этой причине лучше не изменять качество рабочей области при использовании функции кэширования растрового изображения.

Здесь можно использовать подход, включающий кэширование растрового изображения вручную. Для имитации применения свойства `opaqueBackground` фрагмент ролика можно нарисовать в непрозрачном объекте BitmapData, при использовании которого среде выполнения не требуется повторно создавать поверхность растрового изображения.

Этот прием хорошо работает с содержимым, которое не изменяется со временем. Однако, если содержимое текстового поля может изменяться, рассмотрите возможность использования другой стратегии. Например, представьте текстовое поле, в котором показано постоянно обновляющееся значение в процентах, обозначающее ход загрузки приложения. Если текстовое поле или содержащий его экранный объект кэшируется как растровое изображение, его поверхность должна создаваться каждый раз при изменении содержимого. Кэширование растрового изображения вручную невозможно, поскольку содержимое экранного объекта постоянно изменяется. Изменение этой константы заставляет вручную вызывать метод `BitmapData.draw()` для обновления кэшированного растрового изображения.

Помните, что после выхода версии Flash Player 8 (и AIR 1.0) вне зависимости от значения качества рабочей области текстовое поле остается превосходно сглаженным, если для визуализации задан параметр «Сглаживание для читаемости». Этот подход позволяет уменьшить объем используемой памяти, однако приводит к увеличению загрузки ЦП и небольшому замедлению визуализации по сравнению с использованием функции кэширования растрового изображения.

В следующем примере кода используется этот подход.

```
wait_mc.addEventListener( Event.ENTER_FRAME, movePosition );

// Switch to low quality
stage.quality = StageQuality.LOW;
var destX:int = stage.stageWidth >> 1;
var destY:int = stage.stageHeight >> 1;
var preloader:DisplayObject;
function movePosition ( e:Event ):void
{
    preloader = e.currentTarget as DisplayObject;

    preloader.x -= ( preloader.x - destX ) * .1;
    preloader.y -= ( preloader.y - destY ) * .1;

    if ( Math.abs ( preloader.y - destY ) < 1 )
    {
        // Switch back to high quality
        stage.quality = StageQuality.HIGH;
        preloader.removeEventListener ( Event.ENTER_FRAME, movePosition );
    }
}
```

Не рекомендуется использовать этот параметр (Сглаживание для читаемости) для движущегося текста. При масштабировании текста с этим параметром предпринимается попытка сохранить выравнивание текста, что создает эффект смещения. Однако, если содержимое экранного объекта постоянно изменяется и необходимо масштабировать текст, производительность на мобильных устройствах можно повысить путем задания для качества значения `LOW`. По завершении движения восстановите для качества значение `HIGH`.

Графический процессор

Визуализация с помощью графического процессора в приложениях Flash Player

Важным преимуществом Flash Player 10.1 является то, что для визуализации графического содержимого на мобильных устройствах можно использовать графический процессор. Ранее графика визуализировалась только за счет ЦП. Использование графического процессора оптимизирует визуализацию фильтров, растровых изображений, видео и текста. Помните, что визуализация с помощью графического процессора иногда оказывается менее точной, чем программная визуализация. При аппаратной визуализации содержимое может выглядеть не так гладко. Кроме того, Flash Player 10.1 ограничивает визуализацию эффектов Pixel Bender на экране. При использовании аппаратного ускорения результатом визуализации этих эффектов может быть черный квадрат.

Несмотря на то что в проигрывателе Flash Player 10 функция аппаратного ускорения также присутствовала, графический процессор не использовался для графических вычислений. Он только отображал графику на экране. Flash Player 10.1 также использует графический процессор для вычисления графики, что ощутимо сокращает время визуализации. Также снижается и нагрузка на ЦП, что особенно важно на устройствах с ограниченными ресурсами, таких как мобильные устройства.

При выполнении на мобильных устройствах режим использования графического процессора задается автоматически для повышения производительности. Параметру `wmode` уже нельзя задать значение `gpu` для выполнения визуализации за счет графического процессора, однако если задать `wmode` значение `opaque` или `transparent`, аппаратное ускорение будет отключено.

Примечание. *Программная визуализация в приложениях Flash Player для настольных систем по-прежнему выполняется с помощью ЦП. Программная визуализация надежнее по той причине, что на настольных компьютерах драйверы могут существенно отличаться, в результате чего значительно отличаются и результаты визуализации. Кроме того, визуализация может давать разные результаты на настольных компьютерах и мобильных устройствах.*

Визуализация с помощью графического процессора в приложениях AIR для мобильных устройств

Включить аппаратное графическое ускорение в приложении AIR можно путем добавления значения `<renderMode>gpu</renderMode>` в дескриптор приложения. Изменение режима визуализации при выполнении невозможно. На настольных системах параметр `renderMode` игнорируется. В настоящее время ускорение графического процессора не поддерживается.

Ограничения режима визуализации с помощью графического процессора

При использовании режима визуализации с помощью графического процессора в AIR 2.5 действуют следующие ограничения.

- Если графический процессор не может выполнить визуализацию объекта, он не отображается совсем. Альтернативы визуализации с помощью графического процессора нет.
- Не поддерживаются следующие режимы наложения: слой, альфа, стереть, перекрытие, жесткий свет, замена светлым и замена темным.
- Фильтры не поддерживаются.
- PixelBender не поддерживается.

- Многие графические процессоры устанавливают для текстуры максимальный размер 1024 x 1024. В ActionScript это значение определяет максимальный конечный размер при визуализации отображаемого объекта после всех преобразований.
- Компания Adobe не рекомендует использовать режим визуализации с помощью графического процессора в приложениях AIR для воспроизведения видео.
- При визуализации с помощью графического процессора текстовые поля не всегда перемещаются в видимую область, если открыта виртуальная клавиатура. Чтобы текстовое поле было видимым, когда пользователь вводит текст, выполните одно из следующих действий. Поместите текстовое поле в верхнюю половину экрана или перемещайте его в верхнюю половину экрана, когда экран получает фокус.
- Режим визуализации с помощью графического процессора отключен для некоторых устройств, на которых этот режим работает ненадежно. Актуальные сведения можно найти в заметках о выпуске AIR для разработчика.

Оптимальные методы использования режима визуализации с помощью графического процессора

Следующие методы позволят повысить скорость при визуализации с помощью графического процессора.

- Ограничьте число видимых объектов в рабочей области. На визуализацию каждого элемента требуется время, как и на его совмещение с окружающими объектами. Если больше не требуется отображать объект, установите для его свойств `visible` значение `false`. Не следует просто перемещать его за границы рабочей области, скрывать за другими объектами и устанавливать для свойства `alpha` значение 0. Если отображаемый объект больше не требуется, удалите его из рабочей области с помощью команды `removeChild()`.
- Используйте объекты повторно вместо того, чтобы создавать и уничтожать их.
- Используйте растровые изображения, размеры которых приближены, но меньше 2^n x 2^m бит. Размеры не должны быть точно равны степени числа 2, но должны быть приближены к этому значению, не превышая его. Например, изображение с размером 31 x 15 пикселов визуализируется быстрее, чем с размером 33 x 17 пикселов. (31 и 15 чуть меньше степеней числа 2: 32 и 16.)
- По возможности, задайте параметру repeat значение false при вызове метода `Graphic.beginBitmapFill()`.
- Не применяйте перерисовку. Используйте для заднего плана цвет фона. Не накладывайте друг на друга большие фигуры. На отрисовку каждого пиксела требуются системные ресурсы.
- Старайтесь не использовать такие фигуры, как тонкие длинные зубцы, пересекающиеся сами с собой края или много мелких деталей по краям. На визуализацию таких фигур требуется больше времени, чем на визуализацию экранных объектов с ровными краями.
- Ограничьте размер экранных объектов.
- Включите параметры cacheAsBitMap и cacheAsBitmapMatrix, чтобы отображать объекты, графика которых редко обновляется.
- Не используйте API-интерфейс рисования ActionScript (класс Graphics) для создания графики. По возможности вместо этого создавайте такие объекты статически во время разработки.
- Применяйте масштабирование для активов растровых изображений, чтобы установить конечный размер перед выполнением импорта.

Режим визуализации с помощью графического процессора в приложении AIR 2.0.3 для мобильных устройств

Визуализация с помощью графического процессора в приложениях AIR для мобильных устройств, созданных с помощью Packager for iPhone, имеет больше ограничений. Графический процессор эффективно работает только для растровых изображений, фигур с непрерывной заливкой и экранных изображений, для которых задано свойство `cacheAsBitmap`. Если для объекта задано и свойство `cacheAsBitmap`, и свойство `cacheAsBitmapMatrix`, с помощью графического процессора можно эффективно выполнить визуализацию даже для вращаемых и масштабируемых объектов. Графический процессор используется одновременно для других отображаемых объектов, что обычно приводит к снижению производительности визуализации.

Советы по оптимизации производительности визуализации с помощью графического процессора

Визуализация с помощью графического процессора может существенно повысить производительность содержимого SWF, однако метод реализации содержимого играет большую роль. Помните, что настройки, которые раньше хорошо работали при программной визуализации, не дают хороших результатов при визуализации с помощью графического процессора. Ниже представлены советы, которые помогут достичь высокой производительности при визуализации с помощью графического процессора без снижения производительности при аппаратной визуализации.

Примечание. *На мобильных устройствах, поддерживающих аппаратную визуализацию, обращение к содержимому SWF часто происходит через Интернет. Поэтому при создании любого содержимого SWF рекомендуется учитывать эти советы, чтобы достичь наилучшей производительности на всех экранах.*

- Избегайте использования `wmode=transparent` или `wmode=opaque` в качестве внедренных параметров HTML. Производительность в данных режимах может снижаться. Кроме того, они могут приводить к небольшим потерям при синхронизации аудио и видео как при программной, так и при аппаратной визуализации. Также многие платформы не поддерживают визуализацию с помощью графического процессора, если включены данные режимы, что значительно снижает производительность.
- Используйте только обычный режим и режим наложения альфа-канала. Избегайте использования других режимов наложения, особенно режима наложения слоев. Не все режимы наложения воспроизводятся достоверно при визуализации с помощью графического процессора.
- Когда графический процессор отрисовывает векторную графику, перед отрисовкой графика разбивается на сетку, состоящую из небольших треугольников. Данный процесс называется тесселяцией. Тесселяция немного снижает производительность, и чем больше сложность формы, тем сильнее снижается производительность. В целях минимизации влияния на производительность избегайте изменяющихся форм, для которых графический процессор выполняет тесселяцию в каждом кадре.
- Избегайте самопересекающихся кривых, очень тонких изогнутых областей (например, тонких полумесяцев) и сложных деталей по контурам формы. Такие формы являются слишком сложными для того, чтобы графический процессор мог выполнить их разбиение на сетку из треугольников. Чтобы понять причину этого, рассмотрим два вектора: квадрат 500 × 500 и полумесяц 100 × 10. Графический процессор может легко выполнить визуализацию большого квадрата, так как он разбивается всего на два треугольника. Однако для описания кривой полумесяца потребуется множество таких треугольников. Поэтому визуализация данной формы является более сложной, даже если она имеет меньший размер.
- Избегайте больших изменений масштаба, так как при подобных изменениях графическому процессору придется повторно выполнять тесселяцию графики.

- По мере возможности избегайте перерисовки. Перерисовка — это размещение нескольких графических элементов таким образом, чтобы они не перекрывали друг друга. При использовании программной визуализации отрисовка каждого пиксела происходит только один раз. Поэтому при программной визуализации снижения производительности приложения не происходит независимо от количества наложенных друг на друга графических элементов в данной области. При аппаратной визуализации пикселы перерисовываются для каждого элемента независимо от того, перекрывают ли другие элементы данную область. Если имеется наложение двух прямоугольников, при аппаратной визуализации перерисовка области наложения происходит дважды, тогда как при аппаратной визуализации область перерисовывается только один раз.

  Поэтому на настольных компьютерах, которые используют аппаратную визуализацию, снижение производительности в связи с перерисовкой обычно не замечается. Однако большое количество наложенных форм может оказать негативное влияние на производительность устройств, которые используют визуализацию с помощью графического процессора. Наилучшим методом будет удаление этих объектов из списка отображения вместо того, чтобы скрывать их.
- Избегайте использования в качестве фона больших заполненных прямоугольников. Вместо этого задайте цвет фона для объекта Stage.
- По возможности избегайте режима заливки растрового изображения по умолчанию, при котором происходит повтор растрового изображения. Вместо этого для достижения более высокой производительности используйте режим фиксированного растрового изображения.

Асинхронные операции

По возможности используйте вместо синхронных операций их асинхронные аналоги.

Синхронные операции выполняются сразу по команде, и время выполнения ждет завершения их работы. Следовательно, они выполняются в фазе выполнения кода приложения в цикле кадра. Если на выполнение синхронной операции требуется больше времени, то цикл кадра увеличивается, что потенциально приводит к зависанию или не плавному воспроизведению содержимого.

Асинхронные операции не выполняются сразу же. Ваш код и прочий код приложения в текущем потоке выполнения продолжают выполняться. После этого среда выполнения выполнит данную операцию при первой же возможности так, чтобы это не отразилось на качестве визуализации. В некоторых случаях операция выполняется в фоновом режиме и абсолютно не зависит от цикла кадра среды выполнения. По окончании выполнения операции среда выполнения отправляет событие, на которое код может реагировать выполнением дальнейших действий.

Выполнение асинхронных операций планируется так, чтобы они не вызвали проблем с визуализацией. Следовательно, для обеспечения быстродействия приложения лучше использовать асинхронные версии операций. Дополнительные сведения см. в разделе ««Ощущаемая и фактическая производительность» на странице 3».

Но асинхронное выполнение операций влечет некоторые потери. Фактическое время выполнения может увеличиваться при выполнении асинхронных операций, особенно операций, которые выполняются быстро.

В среде выполнения платформы многие из операций всегда выполняются синхронно или наоборот асинхронно независимо от выбора разработчика. Однако в Adobe AIR существует три типа операций, для которых можно выбрать один из двух способов выполнения.

- Операции классов File и FileStream

  Многие операции класса File могут быть выполнены синхронно или асинхронно. Например, для операций копирования или удаления файла или каталога, а также отображения содержимого каталога, существуют как синхронные, так и асинхронные версии. Эти методы имеют суффикс Async, добавленный к названию асинхронной версии. Например, чтобы выполнить асинхронную операцию удаления файла, необходимо вместо метода `File.deleteFile` вызвать метод `File.deleteFileAsync()`.

  При использовании объекта FileStream для считывания или записи файла способ открытия объекта FileStream определяет необходимость выполнения операций в асинхронном режиме. Для выполнения асинхронных операций используйте метод `FileStream.openAsync()`. Запись данных производится в асинхронном режиме. Чтение данных производится поблочно, поэтому одновременно доступна только часть данных. И наоборот, в синхронном режиме объект FileStream сначала завершает чтение всего файла, и лишь затем время выполнения продолжает выполнять код.

- Операции с локальной базой данных SQL

  При работе с локальной базой данных SQL все операции с использованием объекта SQLConnection можно выполнить либо в синхронном, либо в асинхронном режиме. Чтобы выполнить операции в асинхронном режиме, установите подключение к базе данных, используя метод `SQLConnection.openAsync()` вместо метода `SQLConnection.open()`. Асинхронные операции с базой данных выполняются в фоне. Таким образом, механизм базы данных не связан с циклом кадра среды выполнения, поэтому при выполнении операций с базой данных возникновение проблем с визуализацией маловероятно.

  Полезные советы по повышению производительности обмена данными с базой данных SQL см. в разделе ««Производительность базы данных SQL» на странице 94».

- Автономные шейдеры Pixel Bender

  Класс ShaderJob дает возможность пропускать изображение или набор данных через шейдер Pixel Bender и получать доступ к несжатым полученным данным. По умолчанию при вызове метода `ShaderJob.start()` шейдер выполняется асинхронно. Выполнение производится в фоновом режиме и не связано с циклом кадра среды выполнения. Чтобы объект ShaderJob выполнялся синхронно (не рекомендуется), необходимо передать первому параметру метода `start()` значение `true`.

Помимо этих встроенных механизмов для выполнения кода в асинхронном режиме также можно настроить и собственный код на выполнения в синхронном или асинхронном режиме. Если код предназначен для выполнения потенциально длительной операции, можно разбить его на несколько частей. Таким образом, среда выполнения сможет выполнять операции визуализации в промежутках между блоками выполнения кода, что снижает вероятность возникновения проблем с визуализацией.

Ниже перечислено несколько способов разбивки кода. Главная идея всех этих способов заключается в том, чтобы в определенное время выполнить только часть операции. Задача разработчика определить выполняемую кодом операцию и задать временные рамки ее выполнения. Для отслеживания оставшейся работы и выделения отрезков времени для ее дальнейшего выполнения воспользуйтесь механизмом Timer.

Существует несколько проверенных шаблонов разделения кода на функциональные части. Описание этих шаблонов и примеры кода, которые можно использовать в приложениях, находятся в следующих статьях и библиотеках кода.

- Асинхронное выполнение сценариев ActionScript (статья со множеством фоновой информация, а также несколькими примерами реализации, автор Тревор МакКоли (Trevor McCauley))

- Анализ и визуализация большого объема данных в проигрывателе Flash Player (статья с дополнительной фоновой информацией и примерами двух подходов: модель Builder и «зеленые» потоки, автор Джесси Уорден (Jesse Warden))
- «Зеленые» потоки (статья с описанием техники использования «зеленых» потоков с примерами исходного кода, автор Дрю Камминс (Drew Cummins))
- greenthreads (библиотека открытого кода, готового для внедрения в сценарии ActionScript, автор Чарли Хаббард (Charlie Hubbard)). Дополнительные сведения см. в «экспресс-руководстве по «зеленым» потокам».)
- Потоки в ActionScript 3: http://www.adobe.com/go/learn_fp_as3_threads_ru (статья Алекса Харуи (Alex Harui) с примером реализации приема «создания псевдопотоков»)

Прозрачные окна

В приложениях AIR для настольных систем рассмотрите возможность использования непрозрачного прямоугольного окна приложения вместо прозрачного окна.

Чтобы использовать непрозрачное окно в качестве начального окна приложения AIR для настольных систем, в XML-файле дескриптора приложения задайте следующее значение:

```
<initialWindow>
    <transparent>false</transparent>
</initialWindow>
```

Для окон, созданных кодом приложения, создайте объект NativeWindowInitOptions и присвойте его свойству `transparent` значение `false` (по умолчанию). Передайте его конструктору NativeWindow в процессе создания объекта NativeWindow:

```
// NativeWindow: flash.display.NativeWindow class

var initOptions:NativeWindowInitOptions = new NativeWindowInitOptions();
initOptions.transparent = false;
var win:NativeWindow = new NativeWindow(initOptions);
```

При работе с компонентом Flex Window перед вызовом метода `open()` объекта Window убедитесь, что свойство transparent компонента имеет значение false (по умолчанию).

```
// Flex window component: spark.components.Window class

var win:Window = new Window();
win.transparent = false;
win.open();
```

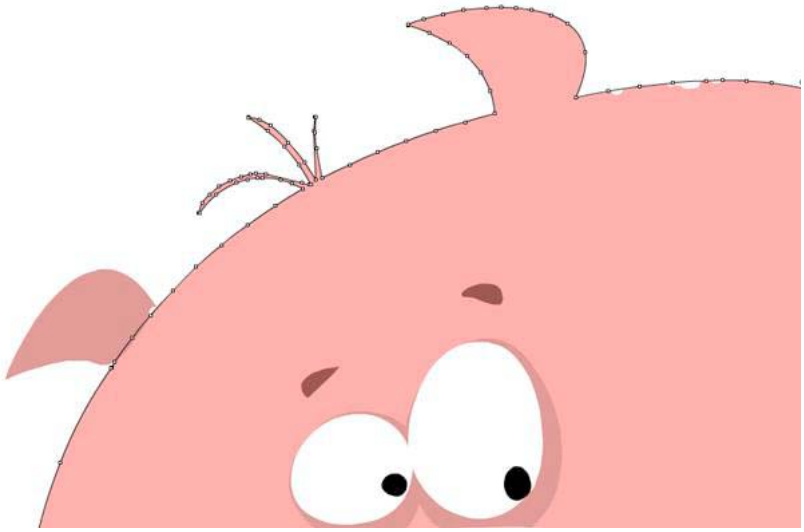
Через прозрачное окно приложения отображается часть рабочего стола пользователя или окна другого приложения, находящегося на заднем плане. Следовательно, для визуализации прозрачного окна среде выполнения требуется больше ресурсов. Для визуализации прямоугольного непрозрачного окна, использующего системный или заказной Chrome, такое количество ресурсов не требуется.

Используйте прозрачные окна только в тех случаях, когда требуется, чтобы содержимое на заднем плане просматривалось через окно приложения или при использовании непрямоугольных окон.

Сглаживание векторных фигур

Сглаживание фигур позволяет повысить производительность.

В отличие от растровых изображений визуализация векторных объектов требует выполнения многочисленных вычислений, особенно если речь идет о градиентах и сложных контурах, содержащих много опорных точек. Задача дизайнера или разработчика — обеспечить достаточную оптимизацию фигур. Представленная ниже фигура иллюстрирует контур до упрощения с многочисленными опорными точками:



Неоптимизированные контуры

Воспользуйтесь инструментом «Сглаживание» во Flash Professional для удаления ненужных опорных точек. Такой же инструмент доступен в Adobe® Illustrator®. Общее число точек и контуров отображается на панели «Информация о документе».

Сглаживание позволяет удалить лишние опорные точки и таким образом уменьшить размер SWF-файла и ускорить его визуализацию. Представленная ниже фигура иллюстрирует те же контуры после сглаживания:

Оптимизированные контуры

Если не переусердствовать с упрощением контуров, такая оптимизация не ухудшает визуальное восприятие. При этом средняя частота кадров в финальной программе может быть значительно повышена за счет упрощения сложных контуров.

Глава 6. Оптимизация сетевого взаимодействия

Усовершенствованные возможности для сетевого взаимодействия

В проигрывателях Flash Player 10.1 и AIR 2.5 представлен набор новых функций, предназначенных для оптимизации работы сети на всех платформах, в том числе кольцевая буферизация и интеллектуальный поиск.

Кольцевая буферизация

При загрузке мультимедийного содержимого в мобильные устройства могут возникать проблемы, которые почти никогда не встречаются в настольных компьютерах. Например, наиболее вероятно возникновение проблем, связанных с нехваткой места на диске или памяти. При загрузке видео настольные версии Flash Player 10.1 и AIR 2.5 загружают и кэшируют весь FLV-файл (или MP 4-файл) на жесткий диск. После этого среда выполнения воспроизводит видео из этого кэшированного файла. Проблема с нехваткой места на диске возникает достаточно редко. Если такое происходит, настольная среда выполнения останавливает воспроизведение видео.

Проблема с нехваткой места на диске в мобильном устройстве возникает гораздо чаще. Если в устройстве недостаточно места на диске, среда выполнения не останавливает воспроизведение, как это делает настольная среда выполнения. Вместо этого среда выполнения повторно использует кэшированный файл, выполняя повторную запись в него с самого начала файла. Пользователь может продолжить просмотр видео. Пользователь не может найти область видео, которая была перезаписана, за исключением начала файла. Кольцевая буферизация не запускается по умолчанию. Ее можно запустить во время воспроизведения, а также в начале воспроизведения, если размер ролика превышает размер места на диске или памяти. Чтобы использовать кольцевую буферизацию, среде выполнения требуется не менее 4 МБ оперативной памяти или 20 МБ места на диске.

Примечание. *Если в устройстве достаточно места на диске, поведение мобильной версии среды выполнения совпадает с поведением версии для настольных систем. Помните, что буфер в оперативной памяти используется в качестве системы восстановления, если в устройстве отсутствует диск или диск заполнен. Во время компиляции можно задать предельный размер кэшированного файла и буфера оперативной памяти. Некоторые MP4-файлы имеют структуру, которая требует загрузки всего файла перед запуском воспроизведения. Среда выполнения обнаруживает такие файлы и предотвращает загрузку, если места на диске недостаточно, и воспроизведение MP4-файла невозможно. Лучше вовсе не запрашивать загрузку таких файлов.*

Разработчик должен помнить, что поиск работает только в пределах кэшированного потока. Иногда не удается выполнить метод `NetStream.seek()`, если смещение находится вне диапазона, и в этом случае отправляется событие `NetStream.Seek.InvalidTime`.

Интеллектуальный поиск

Примечание. *Для использования функции интеллектуального поиска требуется Adobe® Flash® Media Server 3.5.3.*

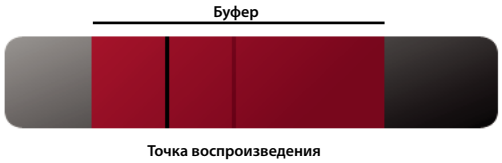
В проигрывателях Flash Player 10.1 и AIR 2.5 представлено новое поведение, называемое интеллектуальным поиском, которое позволяет улучшить взаимодействие с пользователем при воспроизведении потокового видео. Если пользователь выполняет поиск пункта назначения в пределах буфера, среда выполнения повторно использует буфер для выполнения мгновенного поиска. В предыдущих версиях среды выполнения буфер не использовался повторно. Например, если пользователь воспроизводил видео с сервера потоковой передачи и для времени буфера было задано значение 20 секунд (`NetStream.bufferTime`), при переходе на 10 секунд вперед среда выполнения сбрасывала все данные буфера вместо повторного использования уже загруженных 10 секунд видео. Такое поведение вынуждало среду выполнения запрашивать новые данные с сервера гораздо чаще и было причиной плохого качества воспроизведения при медленных подключениях.

На рисунке ниже показано, как заполняется буфер в предыдущих версиях среды выполнения. Свойство `bufferTime` задает число секунд для предварительной загрузки видео, поэтому при сбросе соединения буфер можно использовать без остановки воспроизведения видео:

Поведение буфера до реализации функции интеллектуального поиска

После реализации функции интеллектуального поиска среда выполнения использует буфер для осуществления моментального поиска назад и вперед, когда пользователь проматывает видео. Новое поведение проиллюстрировано ниже.

Поиск вперед с помощью функции интеллектуального поиска



Поиск назад с помощью функции интеллектуального поиска

Функция интеллектуального поиска повторно использует буфер, когда пользователь выполняет перемотку вперед или назад, поэтому воспроизведение выполняется с большей скоростью и более плавно. Одним из преимуществ этого поведения для издателей видео является экономия полосы пропускания. Однако, если поиск выполняется за пределами буфера, поведение является стандартным и среда выполнения запрашивает новые данные с сервера.

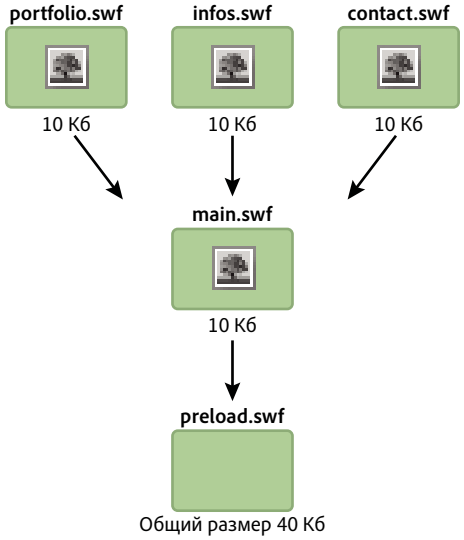
Примечание. *Это поведение не относится к последовательной загрузке видео.*

Чтобы воспользоваться умным поиском, задайте для `NetStream.inBufferSeek` значение `true`.

Внешнее содержимое

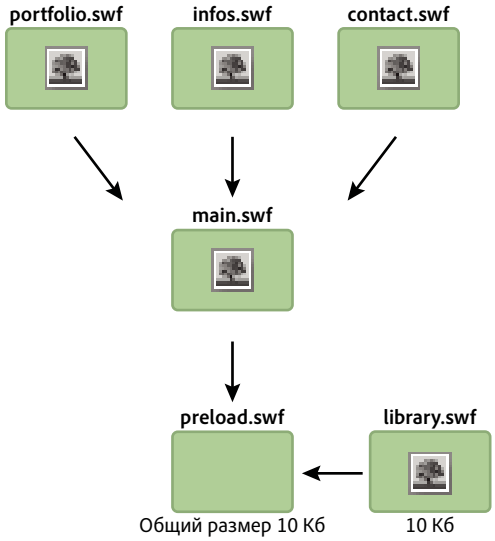
Разделите приложение на несколько SWF-файлов.

Доступ мобильных устройств к сети может быть ограничен. Для обеспечения быстрой загрузки содержимого разделите приложение на несколько SWF-файлов. Попробуйте повторно использовать логику и активы кода во всем приложении. Например, рассмотрите возможность создания приложения, разделенного на несколько SWF-файлов, как показано на следующей схеме.



Приложение, разбитое на несколько SWF-файлов

В этом примере каждый SWF-файл содержит собственную копию одного растрового изображения. Такого дублирования можно избежать с использованием общей библиотеки во время выполнения, как показано на следующей схеме.



Использование общей библиотеки во время выполнения

При использовании этого приема общая библиотека во время выполнения загружается, чтобы растровое изображение было доступно остальным SWF-файлам. Класс ApplicationDomain хранит все загруженные определения классов и делает их доступными во время выполнения с помощью метода `getDefinition()`.

В общей библиотеке во время выполнения может также содержаться вся логика кода. Приложение можно полностью обновить во время выполнения без повторной компиляции. Следующий код загружает общую библиотеку во время выполнения и извлекает определение, содержащееся в SWF-файле, во время выполнения. Такой прием можно использовать со шрифтами, растровыми изображениями, звуками или любым классом ActionScript.

```
// Create a Loader object
var loader:Loader = new Loader();

// Listen to the Event.COMPLETE event
loader.contentLoaderInfo.addEventListener(Event.COMPLETE, loadingComplete );

// Load the SWF file
loader.load(new URLRequest("library.swf") );
var classDefinition:String = "Logo";

function loadingComplete(e:Event ):void
{
    var objectLoaderInfo:LoaderInfo = LoaderInfo ( e.target );

    // Get a reference to the loaded SWF file application domain
    var appDomain:ApplicationDomain = objectLoaderInfo.applicationDomain;

    // Check whether the definition is available
    if ( appDomain.hasDefinition(classDefinition) )
    {
        // Extract definition
        var importLogo:Class = Class ( appDomain.getDefinition(classDefinition) );

        // Instantiate logo
        var instanceLogo:BitmapData = new importLogo(0,0);

        // Add it to the display list
        addChild ( new Bitmap ( instanceLogo ) );
    } else trace ("The class definition " +  classDefinition + " is not available.");
}
```

Получить определение можно более простым способом путем загрузки определений классов в загружающийся домен приложения SWF-файла:

```
// Create a Loader object
var loader:Loader = new Loader();

// Listen to the Event.COMPLETE event
loader.contentLoaderInfo.addEventListener ( Event.COMPLETE, loadingComplete );

// Load the SWF file
loader.load ( new URLRequest ("rsl.swf"), new LoaderContext ( false,
ApplicationDomain.currentDomain) );
var classDefinition:String = "Logo";

function loadingComplete ( e:Event ):void
{
    var objectLoaderInfo:LoaderInfo = LoaderInfo ( e.target );

    // Get a reference to the current SWF file application domain
    var appDomain:ApplicationDomain = ApplicationDomain.currentDomain;

    // Check whether the definition is available
    if (appDomain.hasDefinition( classDefinition ) )
    {
        // Extract definition
        var importLogo:Class = Class ( appDomain.getDefinition(classDefinition) );

        // Instantiate it
        var instanceLogo:BitmapData = new importLogo(0,0);

        // Add it to the display list
        addChild ( new Bitmap ( instanceLogo ) );
    } else trace ("The class definition " +  classDefinition + " is not available.");
}
```

Теперь классы, доступные в загруженном SWF-файле, можно использовать, вызывая метод `getDefinition()` в текущем домене класса. Доступ к классам также можно получить путем вызова метода `getDefinitionByName()`. Этот прием позволяет сэкономить полосу пропускания за счет выполнения загрузки шрифтов и больших активов только один раз. Активы никогда не экспортируются в любые другие SWF-файлы. Единственным ограничением является то, что приложение необходимо протестировать и выполнить с использованием файла loader.swf. Этот файл сначала загружает активы, а затем другие SWF-файлы, из которых состоит приложение.

Ошибки ввода-вывода

Предоставьте обработчики событий и сообщения об ошибках ввода-вывода.

В мобильных устройствах соединение с сетью менее устойчиво, чем в настольных компьютерах с высокоскоростным подключением к Интернету. На доступ к внешнему содержимому в мобильных устройствах накладываются два ограничения: доступность и скорость. Поэтому убедитесь, что активы имеют небольшой размер, и добавьте обработчики для каждого события IO_ERROR для обратной связи с пользователем.

Например, если пользователь просматривает веб-сайт через мобильное устройство и соединение с сетью внезапно прерывается между двумя станциями метро. Динамический актив был загружен при потере соединения. В настольной системе для предотвращения показа ошибки времени выполнения можно использовать пустой прослушиватель событий, поскольку такой сценарий маловероятен. Однако в мобильном устройстве такую ситуацию необходимо обработать не только с помощью простого пустого прослушивателя.

Следующий код не реагирует на ошибку ввода-вывода. Не используйте его в том виде, в каком он приводится.

```
var loader:Loader = new Loader();
loader.contentLoaderInfo.addEventListener( Event.COMPLETE, onComplete );
addChild( loader );
loader.load( new URLRequest ("asset.swf" ) );

function onComplete( e:Event ):void
{
    var loader:Loader = e.currentTarget.loader;
    loader.x = ( stage.stageWidth - e.currentTarget.width ) >> 1;
    loader.y = ( stage.stageHeight - e.currentTarget.height ) >> 1;
}
```

Лучше обработать такую ошибку и вывести для пользователя сообщение об ошибке. Следующий код позволяет правильно обработать ошибку.

```
var loader:Loader = new Loader();
loader.contentLoaderInfo.addEventListener ( Event.COMPLETE, onComplete );
loader.contentLoaderInfo.addEventListener ( IOErrorEvent.IO_ERROR, onIOError );
addChild ( loader );
loader.load ( new URLRequest ("asset.swf" ) );

function onComplete ( e:Event ):void
{
      var loader:Loader = e.currentTarget.loader;
      loader.x = ( stage.stageWidth - e.currentTarget.width ) >> 1;
      loader.y = ( stage.stageHeight - e.currentTarget.height ) >> 1;
}

function onIOError ( e:IOErrorEvent ):void
{
      // Show a message explaining the situation and try to reload the asset.
      // If it fails again, ask the user to retry when the connection will be restored
}
```

Лучшим приемом является вывести для пользователя предложение повторной загрузки содержимого. Это поведение можно реализовать в обработчике `onIOError()`.

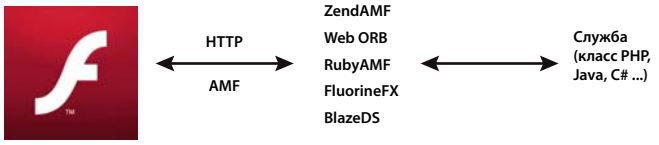
Flash Remoting

Для оптимизации обмена данными между клиентом и сервером используйте технологии Flash Remoting и AMF.

Можно использовать формат XML для загрузки удаленного содержимого в SWF-файлы. Однако формат XML представляет собой открытый текст, который среда выполнения загружает и анализирует. Формат XML лучше подходит для приложений, загружающих ограниченный объем содержимого. При разработке приложения, загружающего большой объем содержимого, рассмотрите возможность использования технологии Flash Remoting и формата Action Message Format (AMF).

Формат AMF — это двоичный формат, используемый для обмена данными между сервером и средой выполнения. При использовании формата AMF уменьшается размер данных и время передачи. AMF является собственным форматом среды выполнения, поэтому при отправке в среду выполнения данных в формате AMF сериализация и десериализация на стороне клиента не требуются. Удаленный шлюз обрабатывает эти задачи. При отправке типа данных ActionScript на сервер удаленный шлюз обрабатывает сериализацию на сервере. Шлюз также отправляет на компьютер пользователя соответствующий тип данных. Этот тип данных связан с классом на сервере, предоставляющем набор методов, которые можно вызвать из среды выполнения. Шлюзы Flash Remoting включают шлюзы ZendAMF, FluorineFX, WebORB и BlazeDS, официальный шлюз Java Flash Remoting с открытым кодом от компании Adobe.

На следующем рисунке показана концепция технологии Flash Remoting.



Flash Remoting

В следующем примере класс NetConnection используется для подключения к шлюзу Flash Remoting.

```
// Create the NetConnection object
var connection:NetConnection = new NetConnection ();

// Connect to a Flash Remoting gateway
connection.connect ("http://www.yourserver.com/remotingservice/gateway.php");

// Asynchronous handlers for incoming data and errors
function success ( incomingData:* ):void
{
    trace( incomingData );
}

function error ( error:* ):void
{
    trace( "Error occured" );
}

// Create an object that handles the mapping to success and error handlers
var serverResult:Responder = new Responder (success, error);

// Call the remote method
connection.call ("org.yourserver.HelloWorld.sayHello", serverResult, "Hello there ?");
```

Подключение к удаленному шлюзу — простой процесс. Однако с помощью технологии Flash Remoting он упрощается еще больше с использованием класса RemoteObject, включенного в пакет Adobe® Flex® SDK.

Примечание. *Внешние SWC-файлы, например файлы из среды Flex, можно использовать в проекте Adobe® Flash® Professional. При использовании SWC-файлов можно работать с классом RemoteObject и его зависимостями без применения остальных компонентов пакета Flex SDK. Опытные разработчики при необходимости могут даже устанавливать связь с удаленным шлюзом непосредственно с использованием примитивного класса Socket.*

Ненужные сетевые операции

Сохраняйте загруженные активы в локальном кэше, избегая их повторной загрузки из сети при необходимости.

Если приложение загружает активы, например мультимедийное содержимое или данные, сохраняйте их в кэше на локальном устройстве. Если активы обновляются не часто, настройте соответствующий интервал обновления кэша. К примеру, можно настроить приложение так, чтобы оно проверяло наличие обновленной версии файла изображения раз в сутки или обновленной версии данных раз в два часа.

В зависимости от типа и природы активов их можно кэшировать несколькими способами.

- Мультимедийные активы, например изображения и видео: сохраняйте файлы в файловой системе, используя классы File и FileStream
- Значения отдельных данных или небольшие наборы данных: сохраняйте значения в качестве локальных общих объектов, используя класс SharedObject
- Более объемные наборы данных: сохраняйте данные в локальной базе данных или сериализуйте данные и сохраняйте их в файл

Для кэширования значений данных можно использовать класс ResourceCache из «проекта AS3CoreLib с открытым исходным кодом».

Глава 7. Работа с мультимедиа

Видео

Сведения по оптимизации производительности видео на мобильных устройствах см. в статье «Оптимизация веб-содержимого для мобильных устройств» на веб-сайте Adobe Developer Connection.

В частности, прочитайте следующие разделы:

- *Воспроизведение видео на мобильных устройствах*
- *Примеры кодов*

В данных разделах представлена следующая информация по разработке видеоплееров для мобильных устройств:

- Инструкции по шифрованию видео
- Передовой опыт
- Профили производительности видеоплееров
- Реализация базового видеоплеера

StageVideo

Класс StageVideo позволяет воспользоваться преимуществами аппаратного ускорения при визуализации видео.

Сведения по использованию объекта StageVideo см. в статье «Использование класса StageVideo для визуализации с аппаратным ускорением» в руководстве разработчика ActionScript 3.0.

Аудио

Начиная с версии Flash Player 9.0.115.0 и AIR 1.0, в среде выполнения можно воспроизводить AAC-файлы (AAC Main, AAC LC и SBR). Использование AAC-файлов вместо MP3-файлов позволяет оптимизировать приложение. При той же скорости потока формат AAC характеризуется лучшим качеством и меньшим размером файлов, чем MP3. Чем меньше файл, тем меньше он влияет на пропускную способность, которая является важным фактором работы на мобильных устройствах без высокоскоростного подключения к Интернету.

Аппаратное декодирование звука

Как и декодирование видео, декодирование звука значительно повышает нагрузку на ЦП, поэтому аппаратное декодирование на устройстве позволяет оптимизировать работу. Проигрыватели Flash Player 10.1 и AIR 2.5 могут обнаруживать и использовать аппаратные аудиодрайверы для более быстрого декодирования AAC-файлов (LC, HE/SBR-профили) или MP3-файлов (PCM не поддерживается). Нагрузка на ЦП заметно снижается, а значит, увеличивается время работы от аккумулятора, а ЦП можно задействовать для других задач.

Примечание. *При использовании формата AAC профиль AAC Main не работает на большинстве устройств из-за недостатка аппаратной поддержки.*

Аппаратное декодирование аудио понятно и пользователю, и разработчику. Когда среде выполнения воспроизводится потоковое аудио, сначала проверяется аппаратное обеспечение, как и при обработке видео. Если аппаратный драйвер доступен и данный формат аудио поддерживается, выполняется аппаратное декодирование аудио. Однако, даже если декодирование входящего потока в формате AAC или MP3 можно выполнить аппаратными средствами, некоторые эффекты приходится декодировать программно. Например, некоторое оборудование не способно декодировать микширование или повторные выборки аудио.

Глава 8. Производительность базы данных SQL

Дизайн приложения и производительность базы данных

Не изменяйте свойство `text` объекта SQLStatement после его выполнения. Вместо этого используйте отдельные экземпляры объекта SQLStatement для каждой инструкции SQL с различными параметрами инструкций для получения требуемых значений.

Перед выполнением любой инструкции SQL среда выполнения подготавливает ее (компилирует), чтобы определить внутренние действия по выполнению инструкции. При вызове метода `SQLStatement.execute()` для экземпляра SQLStatement, который до этого не выполнялся, инструкция автоматически подготавливается перед выполнением. При последующих вызовах метода `execute()`, если свойство `SQLStatement.text` не изменилось, инструкция все еще является подготовленной. Как следствие, она выполняется быстрее.

Чтобы извлечь максимум выгоды от многократного использования инструкции при необходимости изменять значения в интервалах между выполнением инструкции, используйте параметры инструкции для ее настройки. (Параметры инструкции задаются при помощи свойства ассоциативного массива `SQLStatement.parameters`.) В отличие от изменения свойства `text` экземпляра SQLStatement, при изменении значений параметров инструкции среде выполнения не требуется заново подготавливать инструкцию.

Для повторного использования экземпляра SQLStatement приложение должно сохранять ссылку на уже подготовленный экземпляр SQLStatement. Для сохранения ссылки на экземпляр объявите переменную с областью видимости в пределах класса, а не в пределах функции. Для этого рекомендуется структурировать приложение таким образом, чтобы инструкция SQL находилась в оболочке одного класса. Группа инструкций, которые выполняются в комбинации, также может быть заключена в оболочку одного класса. (Эта техника известна как использование шаблона дизайна команды.) За счет определения экземпляров в качестве участвующих переменных класса они продолжают существовать, пока экземпляр класса-оболочки существует в приложении. Можно просто определить переменную, содержащую экземпляр SQLStatement, за пределами функции, чтобы экземпляр оставался в памяти. Например, объявите экземпляр SQLStatement как переменную экземпляра в классе ActionScript или как переменную, не являющуюся функцией, в файле JavaScript. Затем можно задать значения параметров инструкции и вызвать ее метод `execute()` для выполнения запроса.

Для ускорения выполнения операций сравнения и сортировки данных используйте индексы базы данных.

При создании индекса столбца база данных сохраняет копию данных этого столбца. Эта копия хранится отсортированной в числовом или алфавитном порядке. Сортировка позволяет базе данных быстро сопоставлять значения (например, при использовании оператора сравнения) и сортировать результирующие данные с помощью выражения `ORDER BY`.

В базе данных всегда поддерживается актуальность индексов, из-за чего немного замедляется выполнение операций изменения данных (INSERT или UPDATE) в этой таблице. Однако может значительно увеличиться скорость извлечения данных. Из-за такого соотношения производительности не следует индексировать каждый столбец в каждой таблице. Вместо этого используйте стратегию определения собственных индексов. Используйте следующий подход для планирования своей стратегию индексирования:

- Индексируйте столбцы, используемые в таблицах соединения в выражениях WHERE или ORDER BY
- Если столбцы часто используются вместе, индексируйте их в один файл индекса
- При индексации столбца, текстовые данные в котором отсортированы по алфавиту, используйте для него сортировку COLLATE NOCASE


Проводите предварительную компиляцию инструкций SQL во время бездействия приложения.

Первое выполнение инструкции SQL производится медленнее, так как механизм базы данных выполняет подготовку (компиляцию) текста SQL. Так как подготовка и выполнение инструкции может быть трудоемкой операцией, одной из стратегий является предварительная загрузка исходных данных и последующее выполнение других инструкций в фоновом режиме:

1. Сначала загружайте необходимые приложению данные.
2. По завершении начальных операций по запуску приложения или во время «бездействия» приложения выполняйте другие инструкции.

Предположим, приложению совершенно не требуется обращаться к базе данных для отображения начального экрана. В этом случае дождитесь появления начального экрана, а уже потом открывайте подключение к базе данных. Затем создайте экземпляры SQLStatement и выполните любой из них.

Как вариант, при запуске приложения оно может сразу отображать определенные данные, например результат какого-либо запроса. В таком случае можно выполнить экземпляр SQLStatement для этого запроса. После загрузки и отображения начальных данных создайте экземпляры SQLStatement для других операций базы данных и по возможности выполните другие необходимые инструкции позже.

На практике, при повторном использовании экземпляров SQLStatement дополнительное время, необходимое для подготовки инструкции, используется лишь однократно. Возможно, сильного влияния на общую производительность это не окажет.


Объединяйте несколько операций изменения данных SQL в группу.

Предположим, необходимо выполнить большое количество инструкций SQL, предполагающих добавление или изменение данных (инструкции `INSERT` или `UPDATE`). Можно значительно повысить производительность, выполнив все инструкции в пределах явной транзакции. Если не задать явную транзакцию, каждая инструкция начнет выполняться в собственной автоматически созданной транзакции. По завершении выполнения каждой транзакции (каждой инструкции) среда выполнения записывает полученные данные в файл базы данных на диске.

С другой стороны, рассмотрим ситуацию, когда создается явная транзакция и все инструкции выполняются в контексте этой транзакции. Среда выполнения вносит все изменения в память, затем записывает изменения в файл базы данных одновременно при фиксации транзакции. Запись данных на диск является, как правило, самой трудоемкой частью операции. Следовательно, однократная запись на диск может значительно повысить производительность по сравнению с записью при каждом выполнении инструкции SQL.


Производите обработку объемных результатов запроса `SELECT` по частям, используя метод `execute()` класса SQLStatement (с параметром `prefetch`) и метод `next()`.

Предположим, производится выполнение инструкции SQL для получения объемного результата. Приложению необходимо обработать каждую строку данных в цикле. Например, оно форматирует данные или создает из них объекты. Обработка этих данных может занять много времени, что может привести к появлению проблем с рендерингом, например зависанию экрана. Одним из решений является разделение задания на блоки, как описано в разделе ««Асинхронные операции» на странице 79». API-интерфейс базы данных SQL позволяет легко разбивать задачи по обработке данных на части.

Метод `execute()` класса SQLStatement имеет дополнительный параметр `prefetch` (первый параметр). Значение этого параметра определяет максимальное количество строк, возвращаемых базой данных до завершения выполнения операции:

```
dbStatement.addEventListener(SQLEvent.RESULT, resultHandler);
dbStatement.execute(100); // 100 rows maximum returned in the first set
```

После получения первого набора данных можно использовать метод `next()` для возобновления выполнения инструкции и получения следующего набора строк. Подобно методу `execute()`, метод `next()` также использует параметр `prefetch` для определения максимального числа возвращаемых строк:

```
// This method is called when the execute() or next() method completes
function resultHandler(event:SQLEvent):void
{
    var result:SQLResult = dbStatement.getResult();
    if (result != null)
    {
        var numRows:int = result.data.length;
        for (var i:int = 0; i < numRows; i++)
        {
            // Process the result data
        }

        if (!result.complete)
        {
            dbStatement.next(100);
        }
    }
}
```

Вызывайте метод `next()` до тех пор, пока не загрузятся все данные. В предыдущем примере кода показано как можно определить, что загружены все данные. Проверьте свойство `complete` объекта SQLResult, который создается каждый раз при завершении работы метода `execute()` или `next()`.

Примечание. *Используйте параметр `prefetch` и метод `next()` для разделения обработки результирующих данных. Не используйте этот параметр и метод для получения только части результатов запроса. Если необходимо извлечь только подмножество строк в наборе результатов инструкции, используйте выражение `LIMIT` инструкции `SELECT`. Если набор результатов выполнения достаточно большой, можно использовать параметр `prefetch` и метод `next()` для разделения процесса обработки результатов на части.*

Используйте несколько асинхронных объектов SQLConnection с одной базой данных для выполнения нескольких инструкций одновременно.

При подключении объекта SQLConnection к базе данных с использованием метода `openAsync()` он выполняется в фоновом режиме, а не в основном потоке выполнения среды выполнения. К тому же объект SQLConnection выполняется в фоновом режиме в собственном потоке. При использовании нескольких объектов SQLConnection можно эффективно выполнить несколько инструкций SQL одновременно.

У такого подхода существуют и недостатки. Основным недостатком является то, что для создания дополнительных объектов SQLStatement требуется дополнительная память. Одновременное выполнение нескольких операций также приводит к повышению нагрузки на ЦП, особенно если в системе используется один процессор (или процессор с одним ядром). Принимая во внимание эти недостатки, не рекомендуется использовать этот подход при разработке приложений для мобильных устройств.

Еще одним недостатком является то, что объект SQLStatement связан с одним объектом SQLConnection, поэтому потенциальную выгоду от повторного использования объектов SQLStatement извлечь не удастся. Следовательно, объект SQLStatement невозможно использовать повторно, если он связан с уже используемым объектом SQLConnection.

При использовании нескольких объектов SQLConnection, подключенных к одной базе данных, необходимо помнить, что для выполнения инструкций каждый из них использует собственную транзакцию. Учитывайте эти отдельные транзакции в любом коде, который изменяет данные, например добавляет, изменяет или удаляет данные.

Пол Робертсон (Paul Robertson) создал библиотеку с открытым исходным кодом, которая позволяет извлечь максимум выгоды от использования нескольких объектов SQLConnection, минимизируя потенциальные недостатки этого подхода. Эта библиотека использует пул объектов SQLConnection и управляет связанными объектами SQLStatement. Это гарантирует повторное использование объектов SQLStatement, а также доступность объектов SQLConnection для одновременного выполнения различных инструкций. Для получения дополнительных сведений и загрузки библиотеки посетите веб-сайт http://probertson.com/projects/air-sqlite/.

Оптимизация файла базы данных

Избегайте изменений схемы базы данных.

По возможности не изменяйте схему (структуру таблиц) базы данных после добавления данных в таблицы. Как правило, в структуре файла базы данных определения таблиц находятся в начале файла. При открытии подключения к базе данных среда выполнения загружает эти определения. При добавлении данных в таблицы базы данных эти данные вставляются в файл после определения таблицы. Однако при изменении схемы новые данные определения таблиц смешиваются с данными таблиц в файле базы данных. Например, добавление столбца в таблицу или добавление новой таблицы может привести к смешиванию типов данных. Если все данные определения таблиц не находятся в начале файла базы данных, установка соединения с базой данных выполняется дольше. Соединение устанавливается дольше, поскольку среда выполнения дольше читает данные определения таблиц в различных частях файла.

После изменения схемы базы данных используйте метод `SQLConnection.compact()` для проведения оптимизации базы данных.

Если необходимо изменить схему, можно вызвать метод `SQLConnection.compact()` после внесения изменений. При этом изменяется структура файла базы данных так, чтобы все данные определения таблицы находились в одном месте, в начале файла. Однако операция `compact()` может занять много времени, особенно по мере увеличения объема файла базы данных.

Излишняя обработка базы данных во время выполнения

Используйте в инструкции SQL полное имя таблицы (включая название базы данных).

Всегда указывайте в инструкции название базы данных вместе с названием таблицы. (Используйте обозначение «main», если это главная база данных.) Например, в следующем коде название базы данных `main` указано явным образом:

```
SELECT employeeId
FROM main.employees
```

Явное указание имени базы данных позволяет среде выполнения не выполнять проверку каждой подключенной базы данных на наличие соответствующей таблицы. Это также позволяет избежать выбора средой выполнения неправильной базы данных. Следуйте этому правилу, даже если SQLConnection подключен только к одной базе данных, так как SQLConnection также подключается к временной базе данных, доступной посредством инструкций SQL.

Задавайте имена столбцов в инструкциях SQL `INSERT` и `SELECT` явным образом.

Ниже приводятся примеры использования явных имен столбцов:

```
INSERT INTO main.employees (firstName, lastName, salary)
VALUES ("Bob", "Jones", 2000)

SELECT employeeId, lastName, firstName, salary
FROM main.employees
```

Сравните предыдущие примеры со следующими. Не используйте код в следующем стиле:

```
-- bad because column names aren't specified
INSERT INTO main.employees
VALUES ("Bob", "Jones", 2000)

-- bad because it uses a wildcard
SELECT *
FROM main.employees
```

Если имена столбцов не указаны явным образом, среде выполнения потребуется выполнить дополнительные операции для их уточнения. Если в инструкции `SELECT` используется подстановочный символ вместо явного определения столбцов, среда выполнения извлекает дополнительные данные. На обработку лишних данных затрачивается дополнительное время, и создаются ненужные экземпляры объектов.

Не объединяйте одну и ту же таблицу в инструкции несколько раз, если не стоит задача сравнить таблицу саму с собой.

Поскольку инструкция SQL сильно разрастается, таблица базы данных может быть непреднамеренно присоединена к запросу несколько раз. Часто такого же результата можно достичь, используя таблицу только один раз. Присоединение одной таблицы несколько раз с большей вероятностью происходит при использовании одного или нескольких представлений в запросе. Например, таблица может быть присоединена к запросу, а также представлению, включающему данные из этой таблицы. Результатом двух операций будут несколько соединений.

Эффективный синтаксис SQL

Для включения таблицы в запрос используйте `JOIN` (в выражении `FROM`) вместо вложенного запроса в выражении `WHERE`. Этот способ работает эффективно даже тогда, когда необходимо получить из таблицы данные только для фильтрации, а не для создания набора результатов.

Объединение таблиц в выражении `FROM` выполняется быстрее, чем при использовании вложенного запроса в выражении `WHERE`.

Избегайте использования инструкций SQL, которые неэффективно используют индексы. Эти инструкции включают использование функций агрегирования в подзапросе, инструкцию `UNION` в подзапросе или выражение `ORDER BY` с инструкцией `UNION`.

Использование индекса может существенно повысить скорость обработки запроса `SELECT`. Однако некоторые вида синтаксиса SQL не позволяют базе данных использовать индексы, заставляя ее использовать в операциях поиска и сортировки фактические данные.

Попробуйте исключить оператор `LIKE`, особенно если в начале использован подстановочный символ, например `LIKE('%XXXX%')`.

Так как оператор `LIKE` поддерживает поиск с использованием подстановочных символов, операция выполняется медленнее, чем при использовании явных имен. В частности, если вы начали искать строку с использованием подстановочного символа, использование индексов в базе данных во время поиска будет невозможно. В базе данных должен выполняться поиск во всем тексте каждой строки таблицы.

Постарайтесь не использовать оператор `IN`. Если возможные значения известны заранее, можно ускорить выполнение операции, заменив оператор `IN` оператором `AND` или `OR`.

Вторая из следующих двух инструкций выполняется быстрее. Она выполняется быстрее из-за использования простых выражений проверки равенства, объединенных с оператором `OR`, а не инструкций `IN()` или `NOT IN()`:

```
-- Slower
SELECT lastName, firstName, salary
FROM main.employees
WHERE salary IN (2000, 2500)

-- Faster
SELECT lastName, firstName, salary
FROM main.employees
WHERE salary = 2000
    OR salary = 2500
```

Для повышения производительности попробуйте использовать альтернативные формы выражения SQL.

Из примеров выше видно, что форма инструкции SQL также может влиять на производительность базы данных. Существует несколько способов написания инструкции SQL `SELECT` для извлечения определенного набора результатов. В некоторых случаях выбор правильного подхода позволяет значительно ускорить выполнение операции. Узнать о других способах повышения производительности базы данных можно из дополнительных ресурсов по языку SQL.

Производительность инструкции SQL

Проведите непосредственное сравнение инструкций SQL, чтобы определить, какая из них обрабатывается быстрее.

Лучше всего проверить производительность различных версий инструкций непосредственно на базе данных и самих данных.

Ниже представлены инструменты разработчика, которые помогут определить время выполнения инструкций SQL. Воспользуйтесь ими для сравнения скорости обработки различных версий инструкций:

- Run! (средство проверки и разработки запросов SQL в среде AIR, автор Пол Робертсон (Paul Robertson))
- Lita (инструмент администрирования SQLite, автор Дэвид Дэрэ (David Deraedt))

Глава 9. Тестирование и развертывание

Тестирование

Для тестирования предлагается целый ряд инструментов. Среди них — классы Stats и PerformanceTest, разработанные участниками сообщества Flash. Можно также использовать профилировщик в Adobe® Flash® Builder™ и инструмент FlexPMD.

Класс Stats

Для профилирования кода во время выполнения в окончательной версии среды выполнения без внешних инструментов используйте класс Stats, разработанный участником сообщества Flash с псевдонимом mr.doob. Класс Stats можно загрузить на странице: https://github.com/mrdoob/Hi-ReS-Stats.

Класс Stats позволяет проверить следующие характеристики.

- Количество визуализируемых кадров в секунду (чем больше, тем лучше).
- За сколько миллисекунд визуализируется кадр (чем меньше, тем лучше).
- Объем памяти, используемый кодом. Если от кадра к кадру он возрастает, возможно, в приложении есть утечка памяти. Ее необходимо найти и устранить.
- Максимальный объем памяти, используемый приложением.

Загруженный класс Stats можно использовать со следующим небольшим фрагментом кода.

```
import net.hires.debug.*;
addChild( new Stats() );
```

Использование условной компиляции в Adobe® Flash® Professional или Flash Builder активирует объект Stats.

```
CONFIG::DEBUG
{
    import net.hires.debug.*;
    addChild( new Stats() );
}
```

Изменяя значение константы `DEBUG`, можно включать и отключать компиляцию объекта Stats. Таким же образом можно заменять логику кода, чтобы исключить ее при компиляции приложения.

Класс PerformanceTest

Для профилирования выполнения кода ActionScript Грэнт Скиннер (Grant Skinner) разработал инструмент, который можно задействовать в технологическом процессе тестирования. Пользовательский класс передается классу PerformanceTest, который производит с вашим кодом ряд тестов. Класс PerformanceTest позволяет без труда протестировать разные подходы. Класс PerformanceTest может быть загружен на странице http://www.gskinner.com/blog/archives/2009/04/as3_performance.html.

Профилировщик Flash Builder

Flash Builder поставляется с профилировщиком, позволяющим детально проверить код.

Примечание. *Используйте для доступа к профилировщику отладочную версию Flash Player, иначе произойдет ошибка.*

Профилировщик также можно использовать с содержимым, созданным в Adobe Flash Professional. Для этого загрузите скомпилированный SWF-файл из проекта ActionScript или Flex во Flash Builder и запустите профилировщик. Дополнительные сведения о профилировщике см. в разделе «Профилирование приложений Flex» руководства «Использование Flash Builder 4».

FlexPMD

Техническая служба Adobe разработала инструмент FlexPMD, позволяющий оценить качество кода ActionScript 3.0. FlexPMD — это инструмент, схожий с JavaPMD, но предназначенный для ActionScript. FlexPMD оценивает непосредственно исходный код ActionScript 3.0 или Flex, помогая повысить его качество. Инструмент обнаруживает фрагменты неправильного, неоправданно сложного и неоправданно длинного кода, а также случаи некорректного использования жизненного цикла компонентов Flex.

FlexPMD — это проект с открытым исходным кодом от компании Adobe, доступный на странице http://opensource.adobe.com/wiki/display/flexpmd/FlexPMD. Подключаемый модуль Eclipse также доступен на странице http://opensource.adobe.com/wiki/display/flexpmd/FlexPMD+Eclipse+plugin.

FlexPMD помогает оценить код и сделать его максимально чистым и оптимизированным. Сила FlexPMD — в его расширяемости. Разработчик может создать собственный набор правил для оценки любого кода. Например, можно создать правила для обнаружения чрезмерного использования фильтров или других неудачных фрагментов кода, которые требуется исправить.

Развертывание

При экспорте приложения во Flash Builder обязательно удостоверьтесь, что это окончательная версия. При экспорте окончательной версии отладочная информация удаляется из SWF-файла. При удалении отладочной информации размер SWF-файла уменьшается, и приложение работает быстрее.

Используйте для экспорта окончательной версии панель «Проект» во Flash Builder и функцию экспорта окончательной сборки.

Примечание. *При компиляции проекта во Flash Professional нельзя выбрать между окончательной и отладочной версий. Скомпилированный SWF-файл по умолчанию считается окончательной версией.*