Optymalizacja treści na PLATFORMIE ADOBE® FLASH®

Informacje prawne

Informacje prawne można znaleźć na stronie http://help.adobe.com/pl_PL/legalnotices/index.html.

Data ostatniej aktualizacji 11/5/2012

Spis treści

Rozdział 1: Wprowadzenie

Aplikacje Adobe® AIR® i aplikacje programu Adobe® Flash® Player działają na wielu platformach, takich jak komputery stacjonarne, urządzenia przenośne, tablety czy urządzenia telewizyjne. Przykłady kodu i przypadki użycia przedstawione w tym dokumencie prezentują sprawdzone procedury działania przeznaczone dla programistów wdrażających aplikacje tego typu. Uwzględnione zagadnienia:

- Oszczędzanie pamięci
- Minimalizacja użycia procesora
- Optymalizacja wydajności kodu ActionScript 3.0
- Przyspieszanie renderowania
- Optymalizacja interakcji z siecią
- Praca z dźwiękiem i wideo
- Optymalizacja wydajności baz danych SQL
- Testowanie i instalowanie aplikacji

Większość optymalizacji dotyczy aplikacji na wszystkich urządzeniach, zarówno w środowisku wykonawczym AIR, jak i Flash Player. Omówiono również dodatkowe możliwości oraz wyjątki dotyczące pewnych urządzeń.

Niektóre optymalizacje są związane z funkcjami wprowadzonymi w programie Flash Player 10.1 i środowisku AIR 2.5, jednak wiele dotyczy również wcześniejszych wersji środowiska AIR i programu Flash Player.

Fundamentalne zagadnienia wykonywania kodu w środowisku wykonawczym

Kluczem do poznania sposobu na zwiększenie wydajności aplikacji jest zrozumienie, w jaki sposób środowisko wykonawcze platformy Flash wykonuje kod. Środowisko wykonawcze działa w pętli, w której niektóre operacje są wykonywane w każdej klatce. W tym przypadku klatka jest blokiem czasu określonego przez liczbę klatek na sekundę charakterystyczną dla danej aplikacji. Ilość czasu przydzielona dla każdej klatki bezpośrednio przekłada się na liczbę klatek na sekundę. Na przykład: jeśli określono liczbę klatek na sekundę równą 30, wówczas środowisko wykonawcze dba o to, aby każda klatka trwała jedną trzydziestą sekundy.

Początkową liczbę klatek na sekundę należy określić podczas tworzenia aplikacji. Liczbę klatek na sekundę można ustawić za pomocą ustawień w programie Adobe® Flash® Builder™ lub Flash Professional. Początkową liczbę klatek na sekundę można również ustawić w kodzie. W celu ustawienia liczby klatek na sekundę w aplikacji wykorzystującej tylko kod ActionScript należy zastosować znacznik metadanych `[SWF(frameRate="24")]` w głównej klasie dokumentu. W kodzie MXML należy ustawić atrybut `frameRate` w znaczniku Application lub WindowedApplication.

Każda pętla klatki obejmuje dwie fazy podzielone na trzy części: zdarzenia, zdarzenie `enterFrame` oraz rendering.

Pierwsza faza obejmuje dwie części (zdarzenia i zdarzenie `enterFrame`), z których każde może potencjalnie spowodować wywołanie kodu. W pierwszej części pierwszej fazy zdarzenia środowiska wykonawczego docierają i są wywoływane. Te zdarzenia mogą reprezentować zakończenie lub postęp operacji asynchronicznych, takich jak odpowiedź będąca wynikiem ładowania danych przez sieć. Do tych zdarzeń należą także zdarzenia z danych wprowadzonych przez użytkownika. W miarę wywoływania zdarzeń środowisko wykonawcze wykonuje kod w

zarejestrowanych detektorach. Jeśli nie dojdzie do zdarzenia, środowisko wykonawcze będzie oczekiwało na zakończenie tej fazy wykonywania bez wykonania żadnej operacji. Środowisko wykonawcze nigdy nie powoduje zwiększenia liczby klatek na sekundę z powodu braku aktywności. Jeśli w innych częściach cyklu wykonywania będą występowały zdarzenia, wówczas środowisko wykonawcze utworzy kolejkę tych zdarzeń i wywoła je w następnej klatce.

Drugą częścią pierwszej fazy jest zdarzenie `enterFrame`. To zdarzenie odróżnia się od pozostałych, ponieważ jest zawsze wywoływane jeden raz na klatkę.

Po wywołaniu wszystkich zdarzeń rozpoczyna się faza renderowania pętli klatki. W tym momencie środowisko wykonawcze oblicza stan wszystkich elementów widocznych na ekranie i rysuje je na ekranie. Następnie proces powtarza się, dzięki czemu przypomina biegacza na bieżni stadionu.

Uwaga: *W przypadku zdarzeń zawierających właściwość* `updateAfterEvent` *można wymusić natychmiastowe renderowanie, zamiast czekać na fazę renderowania. Należy jednak unikać używania właściwości* `updateAfterEvent`*, jeśli powoduje ona częste problemy z wydajnością.*

Najłatwiej sobie wyobrazić, że każda z tych dwóch faz w pętli klatki trwa tyle samo czasu. W takim przypadku w jednej połowie każdej pętli działają procedury obsługi zdarzeń i działa kod aplikacji, a w drugiej połowie odbywa się renderowanie. Jednak rzeczywistość jest zwykle inna. Czasami wykonanie kodu aplikacji trwa dłużej niż połowa czasu dostępnego w klatce, co powoduje wydłużenie czasu wykonywania i zmniejszenie ilości czasu dostępnego na renderowanie. W innych przypadkach — szczególnie w przypadku złożonych treści wizualnych, takich jak filtry i tryby mieszania, renderowanie trwa dłużej niż pół czasu klatki. Rzeczywista ilość czasu wykonywania poszczególnych faz jest elastyczna, dlatego pętla klatki jest określana jako „elastyczna bieżnia”.

Jeśli łączny czas trwania operacji pętli klatki (wykonanie kodu i renderowanie) jest zbyt długi, wówczas środowisko wykonawcze nie może utrzymać zadanej liczby klatek na sekundę. Klatka zajmuje więcej czasu niż czas przydzielony, dlatego przed wywołaniem następnej klatki dochodzi do opóźnienia. Na przykład, jeśli pętla klatki trwa dłużej niż 1/30 sekundy, środowisko wykonawcze nie będzie w stanie zaktualizować ekranu z szczybkością 30 klatek na sekundę. Gdy liczba klatek na sekundę zmniejsza się, zmniejsza się również zadowolenie użytkownika. W najlepszym przypadku animacja jest odtwarzana w sposób przerywany. W najgorszych przypadkach aplikacja zostanie wstrzymana, a okno staje się puste.

Więcej informacji na temat wykonywania kodu i modelu renderowania w platformie Flash zawierają następujące zasoby:

- Flash Player Mental Model - The Elastic Racetrack (Model Flash Player w wyobraźni — elastyczna bieżnia) (artykuł napisany przez Teda Patricka)
- Asynchronous ActionScript Execution (Asynchroniczne wykonanie kodu ActionScript) (artykuł napisany przez Trevora McCauleya)
- Informacje o optymalizowaniu środowiska Adobe AIR pod względem wykonania kodu, pamięci i renderowania można znaleźć na stronie http://www.adobe.com/go/learn_fp_air_perf_tv_pl (Nagranie wideo prezentacji z konferencji MAX autorstwa Sean Christmann)

Wydajność subiektywna a wydajność rzeczywista

Ostateczny osąd na temat tego, czy aplikacja działa poprawnie, wydają jej użytkownicy. Programiści mogą mierzyć wydajność aplikacji pod względem czasu wykonywania niektórych operacji oraz liczby tworzonych instancji obiektów. Jednak te metryki nie są istotne dla użytkowników końcowych. Niekiedy użytkownicy mierzą wydajność na podstawie innych kryteriów. Na przykład: Czy aplikacja działa szybko i płynnie oraz czy szybko reaguje na dane wejściowe? Czy ujemnie wpływa na wydajność systemu? Każdy programista powinien zadać sobie następujące pytania, które umożliwiają subiektywną ocenę wydajności:

- Czy animacje są wyświetlane gładko, czy z przerwami?
- Czy treść wideo jest wyświetlana gładko, czy z przerwami?
- Czy klipy audio są odtwarzane w sposób ciągły, czy są wstrzymywane i wznawiane?
- Czy podczas długotrwałych operacji okno miga lub znika jego zawartość?
- Czy podczas pisania wpisywane znaki są natychmiast widoczne, czy z opóźnieniem?
- Czy po kliknięciu konkretna operacja jest wykonywana natychmiast, czy z opóźnieniem?
- Czy wentylator procesora działa głośniej podczas pracy aplikacji?
- Na laptopie lub na urządzeniu mobilnym: czy podczas korzystania z aplikacji następuje szybsze zużycie baterii?
- Czy podczas korzystania z tej aplikacji działanie innych aplikacji stało się wolniejsze?

Odróżnienie wydajności subiektywnej i rzeczywistej jest istotne. Sposoby na osiągnięcie najlepszej wydajności subiektywnej nie są zawsze takie same, jak sposoby na osiągnięcie największej szybkości działania aplikacji. Należy się upewnić, że aplikacja nigdy nie wykonuje takiej ilości kodu, przy której środowisko wykonawcze nie mogłoby często aktualizować zawartości ekranu i gromadzić danych wprowadzanych przez użytkownika. W niektórych przypadkach w celu osiągnięcia tej równowagi konieczne jest podzielenie programu na części, pomiędzy którymi środowisko wykonawcze będzie aktualizować ekran. (Konkretne informacje zawiera sekcja „Wydajność renderingu” na stronie 49).

Wskazówki i techniki opisane w niniejszej sekcji są przeznaczone do zwiększania wydajności rzeczywistego wykonania kodu oraz zwiększenia wydajności subiektywnej.

Optymalizacje docelowe

Niektóre udoskonalenia wydajności nie wnoszą widocznych usprawnień dla użytkowników. Ważne jest, aby skupić optymalizację wydajności w obszarach rodzących problemy w danej aplikacji. Niektóre optymalizacje wydajności stanowią ogólne, dobre praktyki, które można stosować zawsze. Inne metody optymalizacji — bez względu na to, czy są użyteczne, czy nie — są zależne od potrzeb aplikacji i przewidzianej grupy użytkowników docelowych. Na przykład aplikacje są zawsze wydajniejsze, jeśli nie są w nich wykorzystywane żadne animacje, wideo, filtry graficzne i efekty. Jednak jedną z przyczyn korzystania z platformy Flash do budowania aplikacji jest możliwość korzystania z funkcji multimedialnych i graficznych, które umożliwiają tworzenie aplikacji bogatych w multimedialne środki wyrazu. Należy więc wziąć pod uwagę, czy żądany poziom tego bogactwa będzie odpowiednio dopasowany do charakterystyk wydajności komputerów i urządzeń, na których będzie uruchamiana dana aplikacja.

Istnieje popularna porada, która głosi: „unikaj optymalizacji zbyt wcześnie”. Niektóre metody optymalizacji wymagają takiego kodu, który będzie trudniejszy w odczycie i mniej elastyczny. Taki kod, po optymalizacji, jest trudniejszy do aktualizowania. W przypadku tego rodzaju optymalizacji przed wyborem metody optymalizacji zawsze lepiej jest poczekać i później zdecydować, czy rzeczywiście konkretna sekcja kodu jest wykonywana wolniej.

W celu zwiększenia wydajności czasami konieczne są kompromisy. W sytuacji idealnej zmniejszenie ilości pamięci zużywanej przez aplikację powoduje również wzrost szybkości wykonywania zadań. Jednak idealne optymalizacje nie zawsze są możliwe. Na przykład: jeśli aplikacja jest wstrzymywana podczas działania, wówczas rozwiązanie obejmuje podział pracy w celu wykonania jej na wielu klatkach. Z uwagi na podział pracy całkowity czas wykonania procesu prawdopodobnie się wydłuży. Istnieje jednak możliwość, że użytkownik nie zauważy dodatkowego czasu, jeśli aplikacja będzie nieprzerwanie reagowała na dane wejściowe i nie zawiesi działania.

Jedną z metod określania tego, co należy optymalizować, oraz tego, czy optymalizacje będą pomocne, są testy wydajności. Kilka technik i wskazówek dotyczących wykonywania testów wydajności przedstawiono w sekcji „Testowanie i instalowanie” na stronie 97.

Więcej informacji o wybieraniu fragmentów aplikacji odpowiednich do optymalizacji zawierają poniższe zasoby:

- Omówienie dostrajania wydajności aplikacji dla środowiska AIR można znaleźć na stronie http://www.adobe.com/go/learn_fp_goldman_tv_pl (Nagranie wideo prezentacji z konferencji MAX autorstwa Olivera Goldmana)
- Omówienie dostrajania wydajności aplikacji dla środowiska AIR można znaleźć na stronie http://www.adobe.com/go/learn_fp_air_perf_devnet_pl (Artykuł w serwisie Adobe Developer Connection autorstwa Olivera Goldmana oparty na prezentacji)

Rozdział 2: Oszczędzanie pamięci

Oszczędzanie pamięci jest zawsze istotne podczas programowania aplikacji, nawet dla komputerów stacjonarnych. Jednak w urządzeniach mobilnych oszczędzanie pamięci jest szczególnie istotne i dlatego tak ważne jest ograniczenie ilości pamięci, jaką wykorzystuje konkretna aplikacja.

Obiekty ekranowe

Należy zawsze wybierać odpowiednie obiekty przeznaczone do wyświetlania.

Język ActionScript 3.0 udostępnia wiele różnych obiektów wyświetlanych. Jedną z najprostszych metod optymalizacji zużycia pamięci jest stosowanie odpowiednich typów obiektów wyświetlanych. Dla prostych kształtów, które nie są interaktywne, należy stosować obiekty Shape. W przypadku obiektów interaktywnych, które nie wymagają osi czasu, należy stosować obiekty Sprite. W animacjach z wykorzystaniem osi czasu należy stosować obiekty MovieClip. Zawsze należy wybierać najbardziej wydajny typ obiektu dla danego zastosowania.

Poniższy kod prezentuje wykorzystanie pamięci dla różnych obiektów wyświetlanych:

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

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

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

Metoda `getSize()` informuje o tym, ile bajtów pamięci wykorzystuje konkretny obiekt. Można się o tym przekonać stosując wiele obiektów MovieClip zamiast pojedynczych obiektów Shape, które powodują niepotrzebne obciążenie pamięci, jeśli funkcje obiektu MovieClip nie są wymagane.

Typy pierwotne

Aby wybrać najbardziej wydajny obiekt dla danego zastosowania i porównać działanie różnych wersji kodu, należy użyć metody `getSize()`.

Wszystkie typy pierwotne, z wyjątkiem typu String, zajmują od 4 do 8 bajtów pamięci. Użycie określonego typu pierwotnego nie spowoduje zmniejszenia tej ilości:

```
// 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
```

Dla typu Number, który reprezentuje wartość 64-bitową, maszyna ActionScript Virtual Machine (AVM) przydziela 8 bajtów, jeśli nie jest określona konkretna wartość typu Number. Wszystkie inne typy pierwotne są zapisywane na 4 bajtach.

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

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

Typ String jest zaimplementowany w inny sposób. Ilość przydzielonej pamięci jest zależna od długości ciągu znaków:

```
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
```

Aby wybrać najbardziej wydajny obiekt dla danego zastosowania i porównać działanie różnych wersji kodu, należy użyć metody `getSize()`.

Ponowne wykorzystywanie obiektów

W miarę możliwości obiekty należy ponownie wykorzystywać zamiast tworzyć je od nowa.

Inną prostą metodą zmniejszenia zużycia pamięci jest ponowne wykorzystywanie obiektów, którego celem jest zapobieganie konieczności ich ponownego tworzenia. Na przykład w pętli nie należy używać następującego kodu:

```
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);
}
```

Odtwarzanie obiektu Rectangle w iteracji każdej pętli powoduje większe zużycie pamięci i spowolnienie, ponieważ w każdej iteracji tworzony jest nowy obiekt. Należy zastosować następujące rozwiązanie:

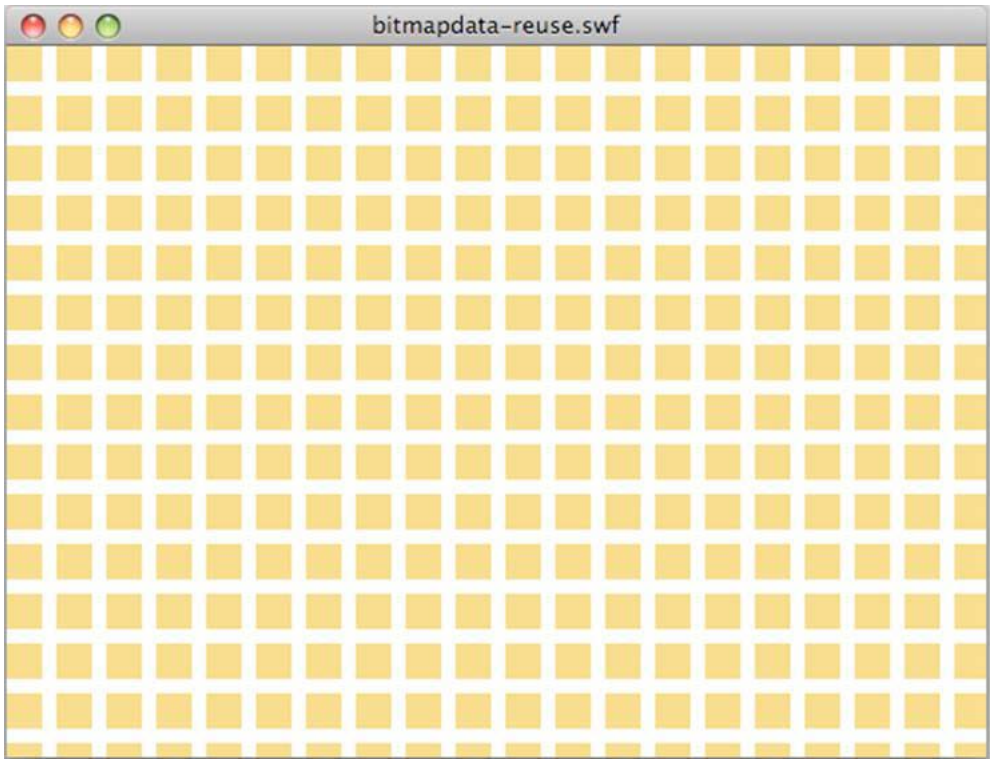
```
const MAX_NUM:int = 18;
const COLOR:uint = 0xCCCCCC;

// 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);
}
```

W poprzednim przykładzie wykorzystano obiekt, który zajmuje stosunkowo niewiele pamięci. Następny przykład prezentuje znaczne ograniczenie zużycia pamięci poprzez ponowne wykorzystanie obiektu BitmapData. Poniższy kod służący tworzenia efektu mozaiki powoduje niepotrzebne zajmowanie pamięci:

```
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);
}
```

Uwaga: *W przypadku wartości dodatnich rzutowanie zaokrąglonej wartości na typ int skraca czas wykonywania kodu w porównaniu z zastosowaniem metody* `Math.floor()`.

Poniższy obraz prezentuje wynik tworzenia mozaiki z bitmapy:



Wynik tworzenia mozaiki z bitmapy:

W zoptymalizowanej wersji tworzona jest jedna instancja BitmapData, do której odwołuje się wiele instancji Bitmap, a wynik jest taki sam:

```
// 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);
}
```

W tym przypadku zmniejszono wykorzystanie pamięci o około 700 kB, co stanowi znaczącą oszczędność w typowym urządzeniu mobilnym. Każdy kontener bitmapy może być modyfikowany bez zmiany oryginalnej instancji obiektu BitmapData — w tym celu należy wykorzystać właściwości 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();
}
```

Poniższy obraz prezentuje wynik transformacji bitmapy:

Wynik transformacji bitmapy:

Więcej tematów Pomocy

„Buforowanie bitmap” na stronie 55

Umieszczanie obiektów w puli

W miarę możliwości należy zawsze umieszczać obiekty w puli.

Inną istotną metodą optymalizacji jest umieszczanie obiektów w puli, co polega na ponownym wykorzystaniu tych samych obiektów w miarę upływu czasu. Należy utworzyć określoną liczbę obiektów podczas inicjowania aplikacji, a następnie zapisać je w puli, która będzie obiektem klasy Array lub Vector. Po zakończeniu pracy z obiektem należy go dezaktywować, dzięki czemu nie będzie on zużywał zasobów procesora, a także usunąć wszystkie odwołania wzajemne. Nie należy jednak ustawiać odwołań na `null`, ponieważ mogłoby to zakwalifikować obiekt do usunięcia w procesie czyszczenia pamięci. Wystarczy umieścić obiekt z powrotem w puli, a następnie pobrać go, gdy potrzebny będzie nowy obiekt.

Ponowne korzystanie z obiektów ogranicza potrzebę tworzenia instancji obiektów, co może być kosztowne. Ponadto ogranicza prawdopodobieństwo uruchomienia procesu czyszczenia pamięci, który może znacznie spowolnić aplikację. Poniższy kod prezentuje technikę umieszczania obiektów w puli.

```
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;
        }
    }
}
```

Klasa SpritePool tworzy pulę nowych obiektów podczas inicjowania aplikacji. Metoda `getSprite()` zwraca instancje tych obiektów, a metoda `disposeSprite()` zwalnia je. Kod umożliwia powiększanie puli, gdy jej pojemność zostanie całkowicie wykorzystana. Możliwe jest także utworzenie puli o stałej wielkości — w takim przypadku całkowite wykorzystanie puli uniemożliwi dodawanie do niej nowych obiektów. W miarę możliwości należy unikać tworzenia nowych obiektów w pętlach. Więcej informacji zawiera sekcja „Zwalnianie pamięci” na stronie 12. W poniższym kodzie wykorzystano klasę SpritePool do pobierania nowych instancji:

```
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 );
    }
}
```

Poniższy kod po kliknięciu przycisku myszy usuwa wszystkie obiekty wyświetlane z listy wyświetlania, a następnie wykorzystuje je ponownie dla innego zadania:

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

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

Uwaga: *Wektor puli zawsze odwołuje się do obiektów Sprite. Jeśli konieczne jest całkowite usunięcie obiektu z pamięci, wymagane jest użycie metody* `dispose()` *klasy SpritePool, która powoduje usunięcie wszystkich pozostałych odwołań.*

Zwalnianie pamięci

Aby uruchomić proces czyszczenia pamięci, należy usunąć wszystkie odwołania do obiektów.

Procesu czyszczenia pamięci nie można uruchomiać bezpośrednio w wersji wydania programu Flash Player. Aby mieć pewność, że obiekt zostanie usunięty przez proces czyszczenia pamięci, należy usunąć wszystkie odwołania do tego obiektu. Należy pamiętać o tym, że stary operator `delete` używany w języku ActionScript 1.0 i 2.0 zachowuje się inaczej w języku ActionScript 3.0. Może służyć wyłącznie do usuwania dynamicznych właściwości w dynamicznym obiekcie.

Uwaga: *Proces czyszczenia pamięci można wywołać bezpośrednio w programie Adobe® AIR® oraz w wersji Flash Player z debugerem.*

Przykład: poniższy kod ustawia odwołanie do obiektu Sprite na `null`:

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

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

Należy pamiętać o tym, że ustawienie obiektu na `null`, nie oznacza jego usunięcia z pamięci. W niektórych przypadkach proces czyszczenia pamięci nie jest uruchamiany, jeśli dostępna ilość pamięci nie zostanie uznana za dostatecznie małą. Czyszczenia pamięci nie można jednoznacznie przewidzieć. Proces przydzielania pamięci uruchamia czyszczenie pamięci, a nie usuwanie obiektów. Uruchomiony proces czyszczenia pamięci znajduje grafy obiektów, które nie zostały jeszcze usunięte. Wykrywa nieaktywne obiekty w grafach, znajdując obiekty, które odwołują się do siebie nawzajem, ale z których nie korzysta już aplikacja. Wykryte w ten sposób nieaktywne obiekty są usuwane.

W przypadku dużych aplikacji ten proces może powodować znaczne obciążenie procesora, wpływać na wydajność i powodować poważne spowolnienie aplikacji. W miarę możliwości należy ograniczać uruchamianie procesu czyszczenia pamięci poprzez ponowne wykorzystywanie obiektów. Ponadto w miarę możliwości należy ustawiać odwołania na null — dzięki temu proces czyszczenia pamięci poświęci mniej czasu wyszukiwaniu obiektów. Proces czyszczenia pamięci należy traktować tylko jako rodzaj zabezpieczenia i zawsze jawnie zarządzać cyklami życia obiektów.

Uwaga: *Przypisanie odwołaniu do obiektu wyświetlanego wartości null nie gwarantuje wstrzymania tego obiektu. Obiekt nadal zużywa cykle procesora, dopóki nie zostanie usunięty przez proces czyszczenia pamięci. Należy zadbać o prawidłowe dezaktywowanie obiektu przed przypisaniem odwołaniu do niego wartości* `null`.

Proces czyszczenia pamięci można uruchamiać za pomocą metody `System.gc()`, która jest dostępna w środowisku Adobe AIR oraz w wersji programu Flash Player z debugerem. Program profilujący dostępny w pakiecie z programem Adobe® Flash® Builder™ umożliwia ręczne uruchamianie procesu czyszczenia pamięci. Uruchomienie procesu czyszczenia pamięci umożliwia sprawdzenie, w jaki sposób reaguje na to aplikacja, oraz czy obiekty są poprawnie usuwane z pamięci.

Uwaga: *Jeśli obiekt był używany jako detektor zdarzenia, inny obiekt może się do niego odwoływać. W takim przypadku detektory zdarzeń należy usunąć za pomocą metody* `removeEventListener()` *przed ustawieniem odwołań na* `null`.

Na szczęście ilość pamięci używanej przez bitmapy można natychmiast zmniejszyć. Na przykład klasa BitmapData zawiera metodę `dispose()`. Poniżej przedstawiono przykład tworzenia instancji klasy BitmapData, która zajmuje 1,8 MB pamięci. Ilość aktualnie wykorzystywanej pamięci wzrasta do 1,8 MB, a właściwość `System.totalMemory` zwraca mniejszą wartość:

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

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

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

Następnie obiekt BitmapData jest ręcznie usuwany (za pomocą metody dispose) z pamięci i ponownie sprawdzane jest aktualne wykorzystanie pamięci:

```
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
```

Mimo że metoda `dispose()` usuwa piksele z pamięci, w celu całkowitego zwolnienia pamięci odwołanie musi być ustawione na `null`. Jeśli obiekt BitmapData nie jest już potrzebny, należy zawsze wywołać metodę `dispose()` i ustawić odwołanie na `null` — dzięki temu pamięć zostanie natychmiast zwolniona.

Uwaga: *W programie Flash Player 10.1 i środowisku AIR 1.5.2 wprowadzono nową metodę o nazwie* `disposeXML()`*, która jest dostępna w klasie System. Ta metoda umożliwia natychmiastowe udostępnienie obiektu XML dla procesu czyszczenia pamięci, ponieważ przekazuje drzewo XML jako parametr.*

Więcej tematów Pomocy

„Wstrzymywanie i aktywowanie obiektów” na stronie 29

Korzystanie z bitmap

Skuteczną metodą oszczędzania pamięci jest stosowanie wektorów zamiast bitmap. Jednak używanie dużej liczby wektorów znacznie zwiększa zapotrzebowanie na zasoby procesora CPU lub procesora graficznego GPU. Stosowanie bitmap jest dobrą metodą optymalizacji renderowania, ponieważ środowisko wykonawcze korzysta z mniejszej mocy obliczeniowej przy rysowaniu pikseli na ekranie niż przy renderowaniu zawartości wektorowej.

Więcej tematów Pomocy

„Ręczne buforowanie bitmap” na stronie 63

Zmniejszanie rozmiaru próbek w bitmapie

W celu zmniejszenia wykorzystania pamięci nieprzezroczyste obrazy 32-bitowe są redukowane do obrazów 16-bitowych, gdy tylko program Flash Player wykryje ekran 16-bitowy. Ten sposób zmniejszania rozmiaru próbek ogranicza wykorzystanie pamięci o połowę, a jednocześnie przyspiesza renderowanie obrazów. Ta funkcja jest dostępna tylko w programie Flash Player 10.1 dla systemu Windows Mobile.

Uwaga: *W wersjach poprzedzających wersję Flash Player 10.1 wszystkie piksele utworzone w pamięci były zapisywane na 32 bitach (4 bajtach). Proste logo o wymiarach 300 x 300 pikseli zajmowało 350 kB pamięci (300*300*4/1024). Dzięki nowemu rozwiązaniu nieprzezroczyste logo zajmuje tylko 175 kB. Jeśli logo jest przezroczyste, rozmiar próbek nie zostanie zmniejszony do 16 bitów i logo zachowa swoją wielkość w pamięci. Ta funkcja ma zastosowanie wyłącznie do osadzonych bitmap lub obrazów ładowanych w środowisku wykonawczym (PNG, GIF, JPG).*

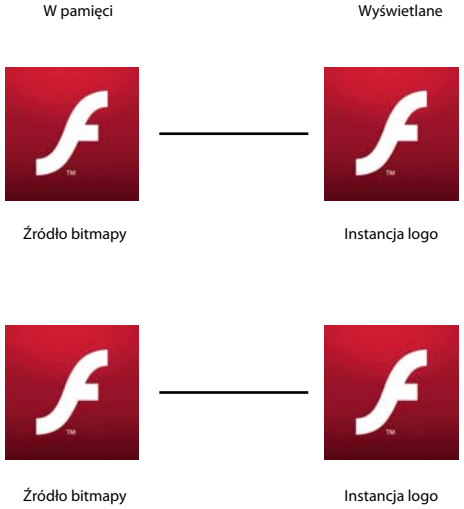
W urządzeniach mobilnych trudno jest odróżnić obraz 16-bitowy od tego samego obrazu zrenderowanego z dokładnością do 32 bitów. W przypadku prostego obrazu zawierającego tylko kilka kolorów różnica jest niewykrywalna. Nawet w przypadku bardziej złożonych obrazów bardzo trudno jest zauważyć jakiekolwiek różnice. Jednak może dochodzić do pewnego pogorszenia jakości kolorów w przypadku powiększania obrazu, a gradient 16-bitowy może wyglądać na mniej wygładzony niż w wersji 32-bitowej.

Pojedyncze odwołanie do obiektu BitmapData

Jak najczęstsze ponowne wykorzystywanie instancji umożliwia zoptymalizowanie użycia klasy BitmapData. W programie Flash Player 10.1 i środowisku AIR 2.5 wprowadzono dla wszystkich platform nową funkcję określaną jako pojedyncze odwołanie do obiektu BitmapData. W przypadku tworzenia instancji BitmapData z osadzonego obrazu dla wszystkich instancji BitmapData używana jest jedna wersja bitmapy. Jeśli później bitmapa zostanie zmodyfikowana, otrzyma w pamięci osobną unikalną bitmapę. Osadzony obraz może pochodzić z biblioteki lub ze znacznika [Embed].

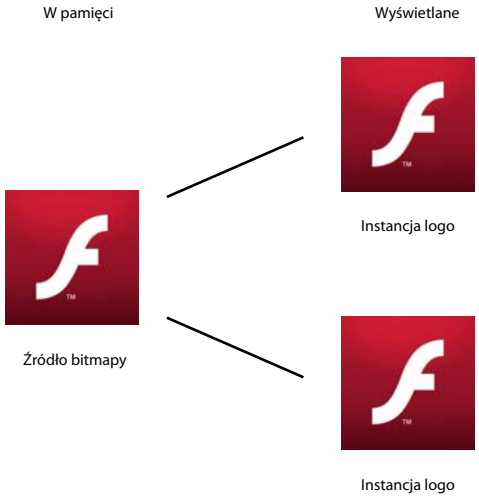
Uwaga: *Nowa funkcja jest przydatna również w przypadku istniejącej zawartości, ponieważ program Flash Player 10.1 i środowisko AIR 2.5 automatycznie ponownie używają bitmap.*

Podczas tworzenia wystąpienia osadzonego obrazu w pamięci tworzona jest powiązana bitmapa. W wersjach oprogramowania wcześniejszych niż Flash Player 10.1 i AIR 2.5 każde wystąpienie otrzymywało osobną bitmapę w pamięci, co przedstawia poniższy diagram.



Bitmapy w pamięci — w wersji oprogramowania wcześniejszej niż Flash Player 10.1 i AIR 2.5

Jeśli w programie Flash Player 10.1 lub środowisku AIR 2.5 tworzonych jest wiele wystąpień tego samego obrazu, wówczas dla wszystkich wystąpień klasy BitmapData jest stosowana jedna wersja bitmapy. Poniższy diagram ilustruje tę koncepcję:

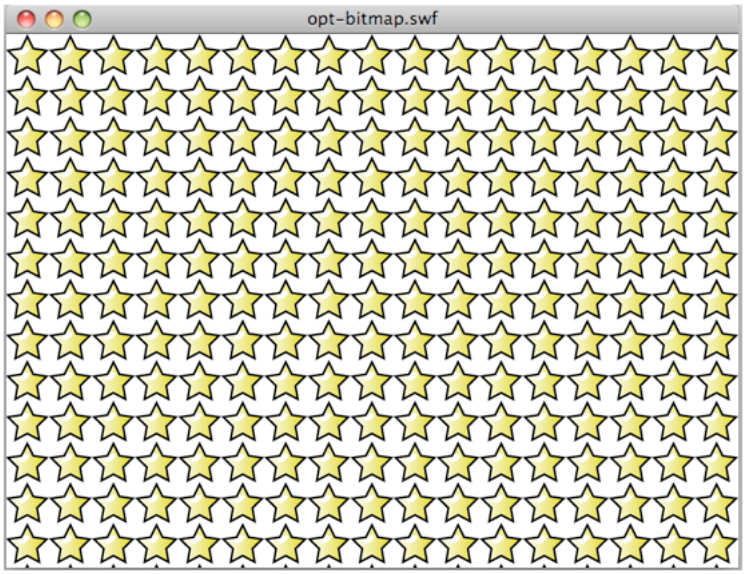


Bitmapy w pamięci w wersji Flash Player 10.1 i AIR 2.5

Taka metoda znacząco zmniejsza ilość pamięci wykorzystywanej przez aplikację z wieloma bitmapami. Poniższy kod tworzy wiele instancji symbolu `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)
    }
}
```

Poniższy obraz prezentuje wynik działania kodu:



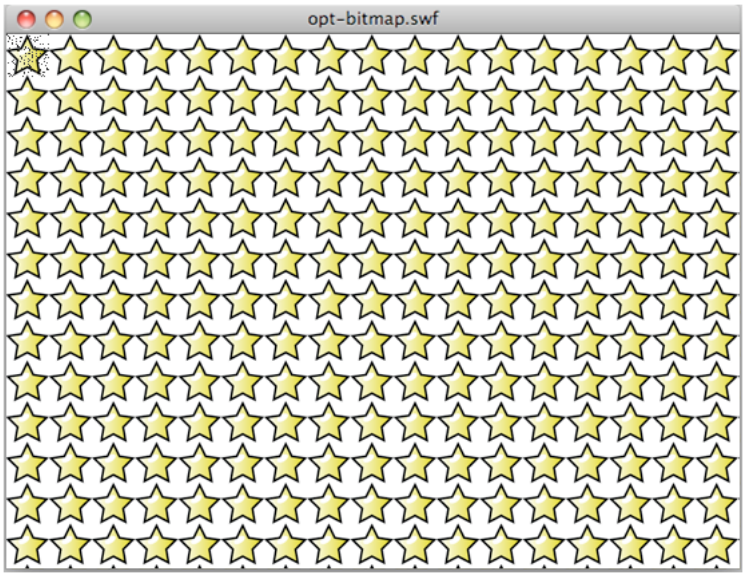
Wynik działania kodu tworzącego wiele instancji symbolu

Na przykład w programie Flash Player 10 powyższa animacja zajmuje około 1008 KB pamięci. W programie Flash Player 10.1 (na komputerze stacjonarnym i na urządzeniu przenośnym) animacja ta zajmuje tylko 4 KB.

Poniższy kod modyfikuje jedną instancję klasy BitmapData:

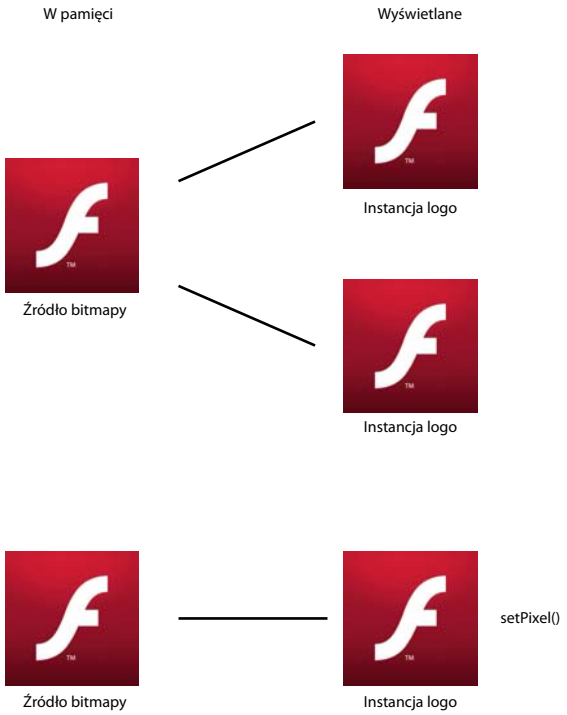
```
const MAX_NUM:int = 18;

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);
```

Poniższy obraz prezentuje wynik modyfikacji jednej instancji `Star`:



Wynik modyfikacji instancji

Środowisko automatycznie przypisuje i tworzy w pamięci bitmapę służącą do obsługi modyfikacji pikseli. Po wywołaniu metody klasy BitmapData, co powoduje modyfikacje pikseli, w pamięci tworzona jest nowa instancja, a żadne inne instancje nie są aktualizowane. Poniższy rysunek ilustruje tę koncepcję:



Wynik modyfikacji jednej bitmapy w pamięci

Zmodyfikowanie jednej gwiazdki powoduje utworzenie nowej kopii w pamięci. Uzyskana w ten sposób animacja zajmuje około 8 KB w pamięci programu Flash Player 10.1 i środowiska AIR 2.5.

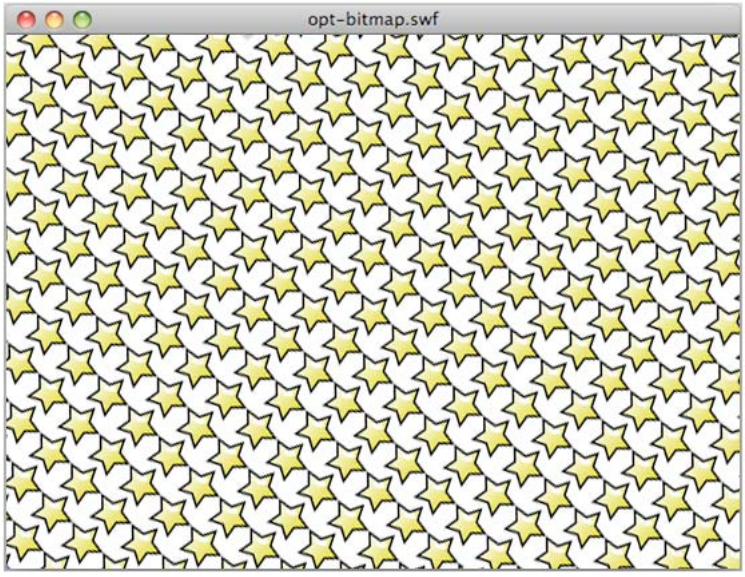
W poprzednim przykładzie każda bitmapa jest dostępna osobno dla transformacji. Metoda `beginBitmapFill()` jest najbardziej odpowiednią metodą do utworzenia mozaiki:

```
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);
```

Ten sposób postępowania zapewnia uzyskanie tego samego wyniku, ale tworzona jest tylko jedna instancja klasy BitmapData. W celu ciągłego obracania gwiazd - zamiast uzyskiwać dostęp do każdej instancji Star - należy użyć obiektu Matrix, który będzie obracany w każdej klatce. Obiekt Matrix należy przekazać do metody `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);
}
```

Gdy stosowana jest ta technika, żadna pętla ActionScript nie jest wymagana do utworzenia tego efektu. Środowisko wykonawcze realizuje wszystkie operacje wewnętrznie. Poniższy obraz prezentuje wynik transformacji gwiazdek:



Wynik obracania gwiazdek

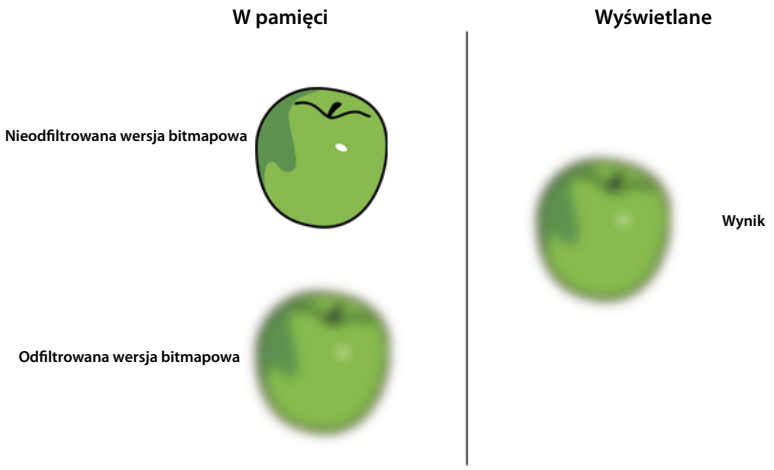
W tym przypadku aktualizacja oryginalnego obiektu źródłowego BitmapData powoduje aktualizację jego wystąpienia na stole montażowym, co może w praktyce okazać się bardzo wydajną techniką programistyczną. Takie podejście nie umożliwia jednak skalowania poszczególnych gwiazdek, jak w poprzednim przykładzie.

Uwaga: *W przypadku stosowania wielu instancji tego samego obrazu sposób rysowania jest zależny od tego, czy jakakolwiek klasa jest powiązana z oryginalną bitmapą w pamięci. Jeśli żadna klasa nie jest powiązana z bitmapą, obrazy są rysowane jako obiekty Shape z wypełnieniami w postaci bitmapy.*

Filtry i dynamiczne usuwanie bitmapy

Należy ograniczyć korzystanie z filtrów, w tym również filtrów przetwarzanych przez moduł Pixel Bender.

Należy w miarę możliwości do minimum ograniczać korzystanie z efektów, takich jak filtry, w tym filtrów przetwarzanych na urządzeniach mobilnych przez mechanizm Pixel Bender. Gdy filtr zostanie zastosowany do obiektu wyświetlanego, środowisko wykonawcze utworzy dwie bitmapy w pamięci. Każda z tych bitmap ma wielkość obiektu wyświetlanego. Pierwsza jest tworzona jako rasteryzowana wersja obiektu wyświetlanego, a następnie jest wykorzystywana w celu utworzenia drugiej bitmapy z zastosowanym filtrem:



Dwie bitmapy w pamięci po zastosowaniu filtru

W przypadku zmodyfikowania właściwości filtru obydwie bitmapy są aktualizowane w pamięci w celu utworzenia bitmapy wynikowej. Ten proces wymaga wykorzystania procesora, a obydwie bitmapy mogą zajmować znaczącą ilość miejsca w pamięci.

W programie Flash Player 10.1 i środowisku AIR 2.5 wprowadzono nowe zachowanie filtrowania dla wszystkich platform. Jeśli filtr nie zostanie zmodyfikowany w ciągu 30 sekund, lub jeśli jest ukryty albo niewidoczny na ekranie, wówczas pamięć zajęta przez niefiltrowaną bitmapę jest zwalniana.

Takie działanie zmniejsza wykorzystanie pamięci przez filtr o połowę na wszystkich platformach. Na przykład: rozważmy obiekt tekstowy z zastosowanym filtrem rozmycia. Tekst w tym przypadku stanowi tylko prostą ozdobę i nie jest modyfikowany. Po 30 sekundach niefiltrowana bitmapa w pamięci jest zwalniana. To samo stanie się, jeśli tekst będzie ukryty przez 30 sekund lub nie będzie widoczny na ekranie. Jeśli jedna z właściwości filtru zostanie zmodyfikowana, niefiltrowana bitmapa w pamięci zostanie odtworzona. Ta funkcja jest nazywana dynamicznym usuwaniem bitmapy. Mimo tych optymalizacji należy zachować ostrożność podczas stosowania filtrów; modyfikacje filtrów nadal wymagają intensywnego użycia procesora CPU lub procesora graficznego GPU.

Sprawdzoną metodą jest używanie bitmap utworzonych za pomocą narzędzia do tworzenia treści, takiego jak Adobe® Photoshop®, w celu emulowania filtrów, gdy tylko jest to możliwe. Należy unikać stosowania dynamicznych bitmap utworzonych w środowisku wykonawczym w kodzie ActionScript. Dzięki zastosowaniu bitmap utworzonych na zewnątrz środowisko wykonawcze może ograniczyć obciążenie procesora lub GPU, szczególnie w przypadkach, gdy w miarę upływu czasu właściwości filtra nie są zmieniane. W razie możliwości w narzędziu do tworzenia zawartości należy przygotować wszystkie efekty wymagane na bitmapie. Następnie bitmapę można wyświetlić w środowisku wykonawczym bez żadnego przetwarzania, co może znacznie skrócić czas wykonywania operacji w programie.

Bezpośredni mipmapping

Mipmapping umożliwia skalowanie dużych obrazów.

Inną nową funkcją dostępną w programie Flash Player 10.1 i środowisku AIR 2.5 na wszystkich platformach jest mipmapping. W programie Flash Player 9 i środowisku AIR 1.0 wprowadzono funkcję mipmappingu, która zwiększała jakość i wydajność bitmap o zmniejszonej skali.

Uwaga: *Funkcja mipmappingu ma zastosowanie tylko do dynamicznie ładowanych obrazów lub osadzonych bitmap. Nie ma natomiast zastosowania do obiektów wyświetlanych, które były filtrowane lub zapisane w pamięci podręcznej. Funkcja mipmapping może być stosowana tylko wówczas, gdy szerokość i wysokość bitmapy są liczbami parzystymi. Jeśli szerokość lub wysokość jest liczbą nieparzystą, mipmapping zostanie przerwany. Na przykład: możliwe jest zastosowanie mipmappingu do obrazu o wymiarach 250 x 250 — wynikiem będzie obraz o wymiarach 125 x 125 i do tego obrazu wynikowego nie można już zastosować mipmappingu. W tym przypadku co najmniej jeden z wymiarów jest liczbą nieparzystą. Mipmapping gwarantuje najlepsze wyniki w przypadku bitmap, których wymiary są potęgami cyfry 2, np.: 256 x 256, 512 x 512, 1024 x 1024 itd.*

Na przykład: wyobraźmy sobie, że załadowano obraz o rozmiarach 1024 x 1024, a programista chce przeskalować ten obraz, aby utworzyć jego miniaturę w galerii. Funkcja mipmappingu renderuje poprawnie przeskalowany obraz, ponieważ wykorzystuje jako tekstury pośrednie wersje bitmapy ze zredukowaną liczbą bitów. W poprzednich wersjach środowisko wykonawcze tworzyło pośrednie, przeskalowane wersje bitmapy w pamięci. Jeśli został wczytany obraz o rozmiarach 1024 x 1024, a następnie został wyświetlony w rozdzielczości 64 x 64, starsze wersje środowiska wykonawczego tworzyły bitmapy o kolejnych rozmiarach zredukowanych o połowę. Na przykład w opisywanym przypadku utworzone zostałyby bitmapy o rozmiarach 512 x 512, 256 x 256, 128 x 128 i 64 x 64.

Program Flash Player 10.1 i środowisko AIR 2.5 realizuje mipmapping bezpośrednio z oryginalnego źródła do wymaganego rozmiaru docelowego. W poprzednim przykładzie utworzone zostałyby tylko: oryginalna bitmapa o wielkości 4 MB (1024 x 1024) i bitmapa o wielkości 16 kB (64 x 64) będąca wynikiem działania funkcji mipmappingu.

Logika mipmappingu działa również z funkcją dynamicznego usuwania bitmapy. Jeśli używana jest tylko bitmapa 64 x 64, wówczas oryginalna bitmapa o wielkości 4 MB zostanie usunięta z pamięci. Jeśli wymagane będzie ponowne przeprowadzenie mipmappingu, oryginalna bitmapa zostanie ponownie załadowana. Ponadto jeśli wymagane są wynikowe bitmapy o różnych wielkościach, zostanie wykorzystany łańcuch bitmap powstających w wyniku mipmappingu. Na przykład: jeśli konieczne jest utworzenie bitmapy 1:8, sprawdzane są bitmapy 1:4, 1:2 i 1:1 w celu określenia, która z nich zostanie załadowana do pamięci jako pierwsza. Jeśli nie zostaną znalezione inne wersje, z zasobu zostanie załadowana oryginalna bitmapa 1:1.

Dekompresor JPEG może realizować funkcję mipmappingu w swoim własnym formacie. Taki bezpośredni mipmapping umożliwia dekompresję dużej bitmapy bezpośrednio do formatu mipmapy bez konieczności ładowania całego zdekompresowanego obrazu. Generowanie mipmapy jest znacznie szybsze, a pamięć wykorzystywana przez duże bitmapy nie jest przydzielana, a następnie zwalniana. Jakość obrazów JPEG jest porównywalna z jakością obrazów uzyskiwanych za pomocą uniwersalnej techniki mipmappingu.

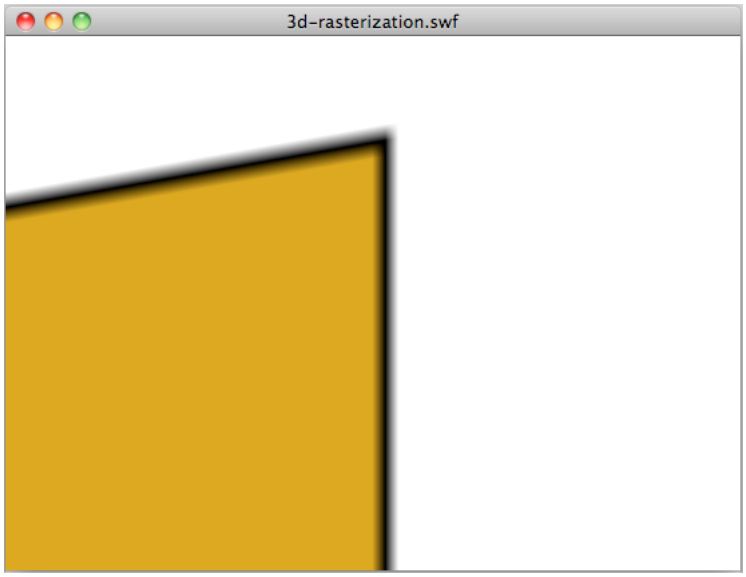
Uwaga: *Z funkcji mipmappingu należy korzystać ostrożnie. Efektem jej stosowania jest poprawa jakości zmniejszanych bitmap, ale funkcja znacznie ogranicza przepustowość, obciąża pamięć i spowalnia działanie systemu. W niektórych przypadkach lepszym rozwiązaniem jest użycie wstępnie zeskalowanej wersji bitmapy z zewnętrznego narzędzia i zaimportowanie jej do aplikacji. Nie należy korzystać z dużych bitmap, jeśli są one przeznaczone wyłącznie do zmniejszenia poprzez skalowanie.*

Korzystanie z efektów 3D


Należy rozważyć możliwość ręcznego tworzenia efektów 3D.

W programie Flash Player 10 i środowisku AIR 1.5 wprowadzono mechanizm 3D, którzy umożliwia stosowanie przekształceń perspektywy do obiektów wyświetlanych. Takie transformacje można stosować za pomocą właściwości `rotationX` i `rotationY` lub za pomocą metody `drawTriangles()` klasy Graphics. W celu uzyskania wrażenia głębi można również korzystać z właściwości `z`. Należy pamiętać o tym, że każdy obiekt wyświetlany z przekształconą perspektywą jest rasteryzowany jako bitmapa i dlatego wymaga większej ilości pamięci.

Poniższy obraz ilustruje wygładzanie będące skutkiem rasteryzacji w przypadku stosowania transformacji perspektywy:



Wygładzanie jako skutek transformacji perspektywy

Wygładzanie jest wynikiem dynamicznego rasteryzowania treści wektorowej, która jest traktowana jak bitmapa. Wygładzenie krawędzi jest stosowane w przypadku używania efektów 3D w wersjach środowiska AIR i programu Flash Player dla komputerów stacjonarnych, a także w środowisku AIR 2.0.1 i AIR 2.5 dla urządzeń przenośnych. Wygładzanie nie jest stosowane w programie Flash Player dla urządzeń przenośnych.

Jeśli efekt 3D zostanie utworzony ręcznie bez korzystania z macierzystego interfejsu API, możliwe jest ograniczenie zużycia pamięci. Nowe funkcje 3D wprowadzone w programie Flash Player 10 i środowisku AIR 1.5 ułatwiają odwzorowywanie tekstur dzięki metodom takim jak `drawTriangles()`, które oferują macierzystą obsługę odwzorowywania tekstur.

Programista musi zdecydować, czy tworzony efekt 3D będzie realizowany wydajniej, gdy zostanie utworzony za pomocą rodzimego interfejsu API, czy też gdy zostanie utworzony ręcznie. Należy rozważyć szybkość wykonywania kodu ActionScript oraz wydajność renderowania, a także ilość zajmowanej pamięci.

Jeśli dla aplikacji dla środowiska AIR 2.0.1 lub AIR 2.5 na urządzeniach przenośnych właściwość `renderMode` zostanie ustawiona na `GPU`, przekształcenia 3D będą obsługiwane przez GPU. W przypadku ustawienia właściwości `renderMode` na `CPU` to procesor, nie GPU, będzie realizować przekształcenia 3D. W aplikacjach dla programu Flash Player 10.1 przekształcenia 3D wykonuje procesor.

Gdy procesor wykonuje przekształcenia 3D, należy pamiętać, że zastosowanie jakiegokolwiek przekształcenia do obiektu wyświetlanego wymaga przechowywania w pamięci dwóch bitmap. Jedna bitmapa zawiera obraz źródłowy, a druga — wersję z przekształconą perspektywą. To sprawia, że przekształcenia 3D działają podobnie do filtrów. W przypadku realizowania przekształceń 3D przez procesor należy więc oszczędnie używać właściwości 3D.

Obiekty tekstowe a pamięć

W przypadku tekstu tylko do odczytu należy korzystać z mechanizmu Adobe® Flash® Text Engine. W przypadku tekstu wprowadzanego należy stosować obiekty TextField.

W programie Flash Player 10 i środowisku AIR 1.5 wprowadzono nowy, zaawansowany mechanizm obsługi tekstu — Adobe Flash Text Engine (FTE) — który pozwala oszczędzać pamięć systemu. Mechanizm FTE jest jednak interfejsem API niskiego poziomu, który wymaga dodatkowej warstwy ActionScript 3.0 dostępnej w pakiecie flash.text.engine.

W przypadku tekstu przeznaczonego tylko do odczytu najlepszym rozwiązaniem jest zastosowanie mechanizmu FTE, który pozwala ograniczyć użycie pamięci i oferuje wydajniejsze renderowanie. W przypadku tekstu wprowadzanego lepsze są obiekty TextField, ponieważ do zrealizowania typowych zadań, jak obsługa wprowadzania i zawijanie tekstu, wymagany jest prostszy kod ActionScript.

Więcej tematów Pomocy

„Renderowanie obiektów tekstowych” na stronie 69

Model zdarzenia i wywołania zwrotne

Zamiast stosować model zdarzenia należy zastanowić się nad zastosowaniem prostych wywołań zwrotnych.

Model zdarzeń programu ActionScript 3.0 jest oparty na koncepcji wywoływania obiektów. Model zdarzeń jest zorientowany obiektowo i zoptymalizowany pod względem ponownego wykorzystania kodu. Metoda `dispatchEvent()` wykonuje pętlę po detektorach z listy i wywołuje procedurę obsługi zdarzeń na każdym zarejestrowanym obiekcie. Jedną z wad modelu zdarzeń jest to, że w cyklu życia aplikacji istnieje prawdopodobieństwo utworzenia wielu obiektów.

Wyobraźmy sobie wywołanie zdarzenia z osi czasu z powiadomieniem o końcu sekwencji animacji. W celu uzyskania powiadomienia można wywołać zdarzenie z konkretnej klatki na osi czasu, co ilustruje poniższy kod:

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

Klasa Document może wykryć to zdarzenie za pośrednictwem następującego wiersza kodu:

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

Mimo, że takie podejście jest poprawne, jednak zastosowanie macierzystego modelu zdarzeń może spowodować działanie wolniejsze niż w przypadku zastosowania tradycyjnej funkcji wywołania zwrotnego. Obiekty zdarzeń muszą być tworzone i przydzielane w pamięci, co powoduje zmniejszenie szybkości działania. Na przykład: podczas wykrywania zdarzenia Event.ENTER_FRAME w każdej klatce tworzony jest nowy obiekt zdarzenia dla procedury obsługi zdarzeń. Szybkość działania może być szczególnie niska dla obiektów wyświetlanych z powodu faz przechwytywania i propagacji, co może być kosztowne, jeśli lista wyświetlania jest złożona.

Rozdział 3: Minimalizacja wykorzystania procesora

Innym istotnym zagadnieniem optymalizacji jest wykorzystanie procesora. Optymalizacja przetwarzania na procesorze zwiększa wydajność, a w rezultacie wydłuża trwałość baterii urządzeń mobilnych.

Udoskonalenia programu Flash Player 10.1 w zakresie wykorzystania procesora

W programie Flash Player 10.1 wprowadzono dwie nowe funkcje, które umożliwiają ograniczenie użycia procesora. Dostępne funkcje obejmują wstrzymywanie i wznawianie odtwarzania zawartości plików SWF zależnie od tego, czy jest widoczna na ekranie, a także ograniczanie liczby wystąpień programu Flash Player na stronę.

Wstrzymywanie, zmniejszanie przepustowości i wznawianie

Uwaga: *Funkcja wstrzymywania, zmniejszania przepustowości i wznawiania nie ma zastosowania do aplikacjach Adobe® AIR®.*

W celu zoptymalizowania obciążenia procesora i zmniejszenia poboru mocy w programie Flash Player 10.1 wprowadzono nową funkcję związaną z nieaktywnymi wystąpieniami. Ta funkcja umożliwia ograniczenie wykorzystania procesora poprzez wstrzymywanie odtwarzania pliku SWF, gdy treść znika z ekranu, a następnie wznowienie — gdy treść pojawia się na ekranie. Dzięki tej funkcji program Flash Player zwalnia maksymalną ilość pamięci, usuwając obiekty, które mogą zostać odtworzone po wznowieniu odtwarzania treści. Treść jest traktowana jako niewidoczna na ekranie, gdy nie jest widoczny żaden jej fragment.

Istnieją dwa scenariusze, które powodują, że treść SWF nie jest widoczna na ekranie:

- Użytkownik przewija stronę i powoduje, że treść SWF przesuwa się poza ekran.

  W tym przypadku, jeśli odtwarzany jest dźwięk lub wideo, będzie ono kontynuowane, ale renderowanie zostanie zatrzymane. Jeśli nie jest odtwarzany żaden dźwięk ani wideo, należy ustawić parametr HTML `hasPriority` na wartość true, aby upewnić się, że nie wstrzymano odtwarzania ani wykonywania kodu ActionScript. Jednak należy pamiętać, że gdy treść zostanie ukryta lub przesunięta poza ekran, renderowanie treści SWF zostanie wstrzymane — bez względu na wartość parametru HTML `hasPriority`.

- W przeglądarce zostaje otwarta inna karta, co powoduje, że treść SWF przesuwa się na drugi plan.

  W takim przypadku — niezależnie od wartości znacznika HTML `hasPriority` — odtwarzanie zawartości SWF jest spowalniane do 2–8 klatek na sekundę (następuje *zmniejszenie przepustowości*). Zostanie zatrzymane odtwarzanie dźwięku i wideo oraz nie będzie przetwarzane żadne renderowanie zawartości, dopóki zawartość SWF nie stanie się ponownie widoczna.

W programie Flash Player 11.2 lub nowszym uruchomionym w przeglądarkach w systemach Windows i Mac można używać w aplikacjach zdarzenia ThrottleEvent. Program Flash Player wywołuje zdarzenie ThrottleEvent, gdy następuje wstrzymanie, zmniejszenie przepustowości lub wznowienie odtwarzania w programie Flash Player.

Jest to zdarzenie ThrottleEvent, co oznacza, że zostanie ono wywołane przez wszystkie obiekty EventDispatcher, których detektor jest zarejestrowany dla tego zdarzenia. Więcej informacji o zdarzeniach broadcast zawiera opis klasy DisplayObject.

Zarządzanie instancjami

Uwaga: *Funkcja zarządzania wystąpieniami nie ma zastosowania do aplikacji Adobe® AIR®.*

Parametr HTML `hasPriority` *umożliwia opóźnienie ładowania plików SWF niewidocznych na ekranie.*

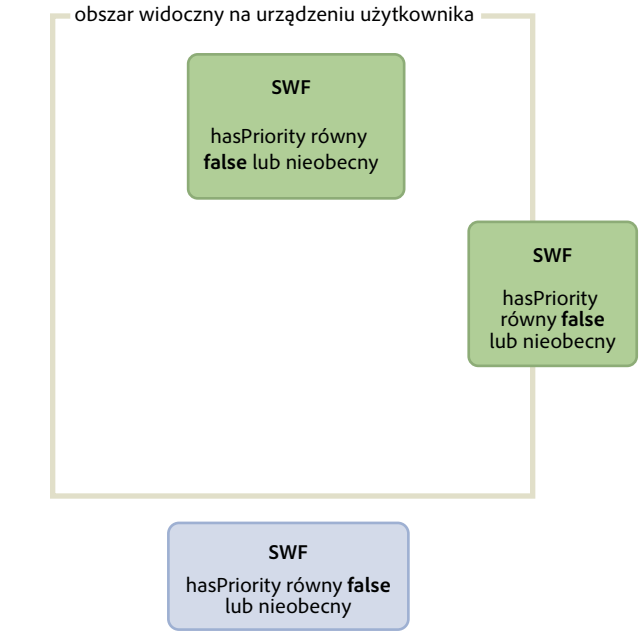
W programie Flash Player 10.1 wprowadzono nowy parametr HTML o nazwie `hasPriority`:

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

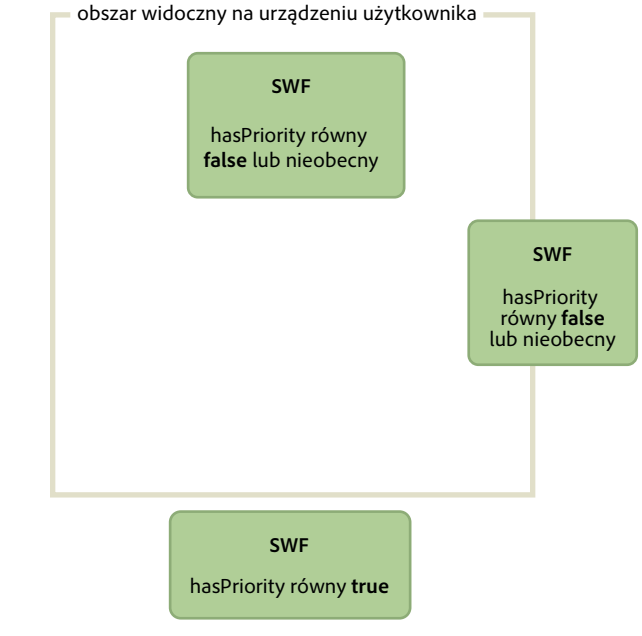
Ten parametr ogranicza liczbę instancji programu Flash Player, jakie są uruchamiane na stronie. Ograniczenie liczby instancji umożliwia minimalizację wykorzystania procesora i baterii. Idea tego parametru jest następująca: do treści SWF należy przypisać określony priorytet, przydzielając w ten sposób pierwszeństwo danej treści wobec pozostałych treści na stronie. Rozważmy prosty przykład: użytkownik przegląda witrynę sieci Web, a strona indeksu zawiera trzy różne pliki SWF. Jeden z nich jest widoczny, drugi jest częściowo widoczny na ekranie, a trzeci nie jest widoczny (konieczne jest przewijanie). Pierwsze dwie animacje są uruchamiane w normalny sposób, ale ostatnia jest opóźniana do momentu, aż stanie się widoczna. Ten scenariusz jest domyślnym sposobem działania, gdy parametr `hasPriority` jest niedostępny lub ustawiony na `false`. Aby mieć pewność, że plik SWF zostanie uruchomiony, nawet jeśli jest niewidoczny na ekranie, należy ustawić parametr `hasPriority` na `true`. Jednak bez względu na wartość parametru `hasPriority`, renderowanie pliku SWF, który nie jest widoczny dla użytkownika, jest zawsze wstrzymane.

Uwaga: *Jeśli zmniejszy się dostępność zasobów procesora, instancje programu Flash Player nie będą uruchamiane automatycznie, nawet jeśli parametr* `hasPriority` *zostanie ustawiony na* `true`*. Jeśli nowe instancje będą tworzone za pomocą kodu JavaScript po załadowaniu strony, będą one ignorowały flagę* `hasPriority`*. Jeśli administrator strony sieci Web nie uwzględni flagi* `hasPriority`*, uruchomiona zostanie każda treść o rozmiarach 1x1 i 0x0 pikseli, co zapobiegnie opóźnianiu plików pomocniczych SWF. Jednak kliknięcie pliku SWF nadal powoduje jego uruchomienie. Takie zachowanie jest określane jako „kliknij, aby odtworzyć”.*

Poniższe diagramy przedstawiają wpływ ustawienia parametru `hasPriority` na różne wartości:



Wpływ różnych wartości parametru hasPriority



Wpływ różnych wartości parametru hasPriority

Tryb uśpienia

W programie Flash Player 10.1 i środowisku AIR 2.5 wprowadzono nową funkcję dostępną na urządzeniach przenośnych, która ogranicza użycie procesora, a w rezultacie wydłuża czas pracy przy zasilaniu akumulatorowym. Ta funkcja steruje podświetleniem ekranu dostępnym w wielu urządzeniach przenośnych. Jeśli na przykład użytkownik korzystający z aplikacji na urządzeniu przenośnym przestanie używać urządzenia, środowisko wykonawcze wykryje przejście podświetlenia w tryb uśpienia. Nastąpi wtedy zmniejszenie szybkości odtwarzania do 4 klatek na sekundę i wstrzymanie renderowania. W przypadku aplikacji AIR tryb uśpienia jest również uaktywniany w momencie, gdy aplikacja przechodzi do działania w tle.

Kod ActionScript będzie nadal wykonywany w trybie uśpienia, podobnie jak przy właściwości `Stage.frameRate` ustawiona na 4 klatki na sekundę. Jednak krok renderowania zostanie pominięty, dlatego użytkownik nie będzie mógł zobaczyć, że odtwarzacz faktycznie odtwarza z szybkością 4 klatek na sekundę. Wybrano 4 klatki na sekundę zamiast zera, ponieważ taka wartość umożliwia zachowanie wszystkich połączeń jako otwartych (NetStream, Socket i NetConnection). Przełączenie na zero spowodowałoby przerwanie wszystkich otwartych połączeń. Wybrano częstotliwość odświeżania równą 250 ms (4 klatki na sekundę), ponieważ wielu producentów urządzeń stosuje taką szybkość odtwarzania jako częstotliwość odświeżania. Użycie tej wartości powoduje utrzymanie szybkości odtwarzania środowiska wykonawczego w zakresie stosowanym na urządzeniu.

Uwaga: *Gdy środowisko wykonawcze działa w trybie uśpienia, właściwość* `Stage.frameRate` *odpowiada szybkości odtwarzania oryginalnego pliku SWF — nie jest to wartość 4 klatek na sekundę.*

Po ponownym włączeniu podświetlenia ekranu renderowanie jest wznawiane. Liczba klatek na sekundę wraca do wartości wyjściowej. Rozważmy odtwarzacz multimediów, w którym użytkownik odtwarza muzykę. Jeśli ekran zostanie przełączony w tryb uśpienia, środowisko wykonawcze będzie reagować odpowiednio do typu odtwarzanej zawartości. Oto lista sytuacji i odpowiadających im działań środowiska wykonawczego:

- Podświetlenie przechodzi w tryb uśpienia, a odtwarzana zawartość jest inna niż audio/wideo: Renderowanie zostaje wstrzymane, a szybkość odtwarzania zostaje ustawiona na 4 klatki na sekundę.
- Podświetlenie przechodzi w tryb uśpienia i jest odtwarzana zawartość audio/wideo: Środowisko wykonawcze wymusza stałe działanie podświetlenia, dzięki czemu użytkownik może kontynuować korzystanie z aplikacji.
- Podświetlenie przechodzi z trybu uśpienia do trybu włączenia: Środowisko wykonawcze przestawia szybkość odtwarzania na zgodną z ustawieniem pierwotnym dla pliku SWF i wznawia renderowanie.
- Działanie programu Flash Player zostało wstrzymane podczas odtwarzania zawartości audio/wideo: Program Flash Player zeruje stan podświetlenia (ustawia zachowanie domyślne w systemie), ponieważ zawartość audio/wideo nie jest już odtwarzana.
- Użytkownik urządzenia przenośnego odbiera połączenie telefoniczne podczas odtwarzania zawartości audio/wideo: Renderowanie zostaje wstrzymane, a szybkość odtwarzania zostaje ustawiona na 4 klatki na sekundę.
- Tryb uśpienia podświetlenia zostaje wyłączony na urządzeniu przenośnym: Środowisko wykonawcze działa normalnie.

Gdy podświetlenie przechodzi w tryb uśpienia, renderowanie zostaje wstrzymane, a szybkość odtwarzania jest redukowana. Takie działanie zmniejsza obciążenie procesora, ale nie można traktować wstrzymania jako prawdziwej pauzy (jak w grze).

Uwaga: *Przy aktywacji lub wyłączeniu trybu uśpienia środowiska wykonawczego nie jest wywoływane żadne zdarzenie ActionScript.*

Wstrzymywanie i aktywowanie obiektów

W celu poprawnego wstrzymywania i aktywowania obiektów należy posłużyć się zdarzeniami `REMOVED_FROM_STAGE` *i* `ADDED_TO_STAGE`*.*

W celu optymalizacji kodu zawsze należy wstrzymywać i aktywować obiekty. Wstrzymywanie i aktywowanie jest istotne dla wszystkich obiektów, ale przede wszystkim dla obiektów wyświetlanych. Nawet jeśli obiekty wyświetlane nie znajdują się na liście wyświetlania i oczekują na usunięcie przez proces czyszczenia pamięci, nadal mogą wymuszać wykonanie kodu intensywnie korzystającego z zasobów procesora. Na przykład nadal mogą korzystać ze zdarzenia Event.ENTER_FRAME. Dlatego poprawne wstrzymywanie i aktywowanie obiektów za pomocą zdarzeń `Event.REMOVED_FROM_STAGE` i `Event.ADDED_TO_STAGE` jest tak ważne. Poniższy przykład prezentuje klip filmowy odtwarzany na stole montażowym. Klip reaguje na czynności wykonywane z klawiatury:

```
// 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;
    }
}
```

Klip filmowy reagujący na naciśnięcia klawiszy

Kliknięcie przycisku Remove powoduje usunięcie klipu filmowego z listy wyświetlania:

```
// 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);
}
```

Po usunięciu z listy odtwarzania klip filmowy nadal wywołuje zdarzenie `Event.ENTER_FRAME`. Klip filmowy nadal działa, ale nie jest renderowany. W celu poprawnego obsłużenia tej sytuacji należy wykryć odpowiednie zdarzenia i usunąć detektory zdarzeń, aby nie dopuścić do wykonania kodu, który znacznie wykorzystuje zasoby procesora:

```
// 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();
}
```

Kliknięcie przycisku Show powoduje restart klipu filmowego, ponowne wykrywanie zdarzeń `Event.ENTER_FRAME` przez klip i sprawia, że klawiatura poprawnie kontroluje klip filmowy.

Uwaga: *Jeśli obiekt wyświetlany zostanie usunięty z listy wyświetlania, wówczas ustawienie go na* `null` *po usunięciu nie zapewni wstrzymania tego obiektu. Jeśli proces czyszczenia pamięci nie zostanie uruchomiony, obiekt nadal będzie zużywał pamięć i zasoby procesora, nawet jeśli nie będzie już wyświetlany. Aby mieć pewność, że obiekt będzie zużywał możliwie jak najmniejszą ilość zasobów procesora, należy zadbać o to, aby przed usunięciem z listy wyświetlania obiekt został całkowicie wstrzymany.*

W oprogramowaniu Flash Player 10 i AIR 1.5 oraz w nowszych wersjach występują również następujące zachowania: Jeśli głowica odtwarzania napotyka pustą klatkę, obiekt wyświetlany jest automatycznie wstrzymywany, nawet jeśli użytkownik nie zaimplementował żadnego działania wstrzymania.

Koncepcja wstrzymania jest również istotna w przypadku wczytywania zdalnej zawartości za pomocą klasy Loader. W przypadku stosowania klasy Loader w programie Flash Player 9 i środowisku AIR 1.0 konieczne było ręczne wstrzymanie zawartości przez wykrycie zdarzenia `Event.UNLOAD` wywołanego przez obiekt LoaderInfo. Konieczne było ręczne wstrzymanie każdego obiektu, co nie było prostym zadaniem. W programie Flash Player 10 i środowisku AIR 1.5 wprowadzono istotną nową metodę klasy Loader o nazwie `unloadAndStop()`. Ta metoda pozwala na usunięcie pliku SWF, automatyczne wstrzymanie każdego obiektu w załadowanym pliku SWF oraz wymuszenie uruchomienia procesu czyszczenia pamięci.

W poniższym kodzie plik SWF jest ładowany, a następnie usuwany za pomocą metody `unload()`, co wymaga większego dodatkowego przetwarzania i ręcznego wstrzymania:

```
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();
}
```

Dobrą praktyką jest użycie metody `unloadAndStop()`, która rodzimie obsługuje wstrzymywanie i wymusza uruchomienie procesu czyszczenia pamięci:

```
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();
}
```

Wywołanie metody `unloadAndStop()` powoduje wykonanie następujących operacji:

- Dźwięki zostają zatrzymane.
- Detektory zarejestrowane w głównej osi czasu pliku SWF zostają usunięte.
- Obiekty czasomierza zostają zatrzymane.
- Urządzenia peryferyjne (takie jak aparat i mikrofon) zostają zwolnione.
- Wszystkie klipy filmowe zostają zatrzymane.
- Wywoływanie zdarzeń `Event.ENTER_FRAME`, `Event.FRAME_CONSTRUCTED`, `Event.EXIT_FRAME`, `Event.ACTIVATE` i `Event.DEACTIVATE` zostaje zatrzymane.

Zdarzenia activate i deactivate

Korzystając ze zdarzeń `Event.ACTIVATE` i `Event.DEACTIVATE`, można wykrywać bezczynność programów działających w tle i odpowiednio optymalizować aplikację.

Dwa zdarzenia (`Event.ACTIVATE` i `Event.DEACTIVATE`) pozwalają dostosować aplikację tak, aby używała możliwie najmniejszej liczby cykli procesora. Zdarzenia te umożliwiają wykrywanie, kiedy środowisko wykonawcze staje się aktywne i nieaktywne. Pozwala to zoptymalizować kod, aby reagował na zmiany kontekstu. Poniższy kod wykrywa oba wymienione zdarzenia i dynamicznie zmniejsza liczbę klatek na sekundę do zera, gdy aplikacja przestaje być aktywna. Animacja może na przykład przestać być aktywna, gdy użytkownik zmieni zakładkę lub przeniesie aplikację do tła.

```
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;
}
```

Gdy aplikacja staje się ponownie aktywna, następuje wyzerowanie szybkości odtwarzania — przywrócenie pierwotnej wartości. Zamiast dynamicznie zmieniać szybkość odtwarzania, można rozważyć inne optymalizacje, takie jak wstrzymywanie i uaktywnianie obiektów.

Zdarzenia activate i deactivate umożliwiają zaimplementowanie mechanizmu podobnego do funkcji wstrzymywania i wznawiania dostępnej niekiedy na urządzeniach przenośnych i netbookach.

Więcej tematów Pomocy

„Liczba klatek na sekundę w aplikacji” na stronie 54

„Wstrzymywanie i aktywowanie obiektów” na stronie 29

Działania wykonywane za pomocą myszy

Jeśli to możliwe, należy rozważyć wyłączenie działań wykonywanych za pomocą myszy.

W przypadku korzystania z obiektu interaktywnego, na przykład klasy MovieClip lub Sprite, środowisko wykonawcze wykonuje kod macierzysty w celu wykrywania i obsługiwania działań wykonywanych za pomocą myszy. Wykrywanie działań wykonywanych za pomocą myszy może znacznie obciążać procesor, szczególnie wówczas, gdy na ekranie znajduje się wiele interaktywnych obiektów, które na siebie nachodzą. Łatwą metodą ograniczenia przetwarzania jest wyłączenie działań wykonywanych za pomocą myszy dotyczących obiektów, które nie wymagają żadnych takich działań. Poniższy kod ilustruje użycie właściwości `mouseEnabled` i `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;
```

Jeśli jest to możliwe, należy rozważyć wyłączenie działań wykonywanych za pomocą myszy. Dzięki temu aplikacja będzie zużywać mniej zasobów procesora, co wydłuży czas działania urządzeń przenośnych przy zasilaniu akumulatorowym.

Czasomierze i zdarzenia ENTER_FRAME

W zależności od tego, czy treść jest animowana, należy wykorzystywać czasomierze lub zdarzenia `ENTER_FRAME`.

W przypadku treści, która nie jest animowana i jest wykonywana przez długi czas, preferowane są czasomierze zamiast zdarzeń `Event.ENTER_FRAME`.

W języku ActionScript 3.0 istnieją dwa sposoby wywoływania funkcji w konkretnych odstępach czasu. Pierwszy sposób polega na używaniu zdarzenia `Event.ENTER_FRAME` wywoływanego przez obiekty ekranowe (wystąpienia klasy DisplayObject). Drugi z nich zakłada wykorzystanie czasomierza. Programiści pracujący w kodzie ActionScript często korzystają ze zdarzenia `ENTER_FRAME`. Zdarzenie `ENTER_FRAME` jest wywoływane w każdej klatce. W rezultacie częstotliwość wywoływania funkcji jest powiązana z bieżącą liczbą klatek na sekundę. Liczbę klatek na sekundę można określić na podstawie właściwości `Stage.frameRate`. Jednak w niektórych przypadkach zastosowanie czasomierza może być lepszym rozwiązaniem niż zastosowanie zdarzenia `ENTER_FRAME`. Na przykład: jeśli nie jest stosowana animacja, ale wymagane jest wywoływanie kodu z określoną częstotliwością, wówczas zastosowanie czasomierza będzie lepszym rozwiązaniem.

Czasomierz może działać w sposób podobny do zdarzenia `ENTER_FRAME`, ale zdarzenie może być wywoływane bez związku z liczbą klatek na sekundę. Ten sposób działania może zapewniać pewne istotne korzyści w zakresie optymalizacji. Rozważmy przykładową aplikację, która jest odtwarzaczem wideo. W tym przypadku nie ma potrzeby stosowania wysokiej liczby klatek na sekundę, ponieważ ruchome są tylko elementy służące do sterowania aplikacją.

Uwaga: *Liczba klatek na sekundę nie wpływa na wideo, ponieważ wideo nie jest osadzone w osi czasu. Zamiast tego wideo jest ładowane dynamicznie poprzez progresywne ładowanie lub przesyłanie strumieniowe.*

W tym przykładzie liczba klatek na sekundę jest ustawiona na niską wartość: 10. Czasomierz aktualizuje elementy sterujące jeden raz na sekundę. Wyższa częstotliwość aktualizacji jest możliwa dzięki metodzie `updateAfterEvent()`, która jest dostępna w obiekcie TimerEvent. Ta metoda wymusza aktualizację ekranu za każdym razem, gdy czasomierz wywołuje zdarzenie (w razie potrzeby). To rozwiązanie przedstawia poniższy kod:

```
// 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();
}
```

Wywołanie metody `updateAfterEvent()` nie powoduje modyfikacji liczby klatek na sekundę. Wymusza na środowisku wykonawczym aktualizację tej zawartości na ekranie, która uległa zmianie. Oś czasu nadal działa z liczbą 10 klatek na sekundę. Należy pamiętać o tym, że czasomierze i zdarzenia `ENTER_FRAME` nie są idealnie dokładne na urządzeniach o niskiej wydajności ani wówczas, gdy procedury obsługi zdarzeń zawierają kod znacznie wykorzystujący zasoby procesora. Częstotliwość aktualizacji czasomierza — podobnie jak liczba klatek na sekundę pliku SWF — może się różnić w niektórych sytuacjach.

Minimalizowanie liczby obiektów Timer i zarejestrowanych procedur obsługi `enterFrame` w aplikacji.

W każdej klatce środowisko wykonawcze wywołuje zdarzenie `enterFrame` dla każdego obiektu wyświetlanego z listy wyświetlanej. Mimo, że możliwe jest rejestrowanie detektorów dla zdarzenia `enterFrame` z wieloma obiektami wyświetlanymi, jednak taka metoda rejestrowania oznacza, że w każdej klatce wykonywana jest większa ilość kodu. Zamiast tego należy rozważyć zastosowanie pojedynczej procedury obsługi `enterFrame`, która wykona cały kod, jaki ma być wykonywany w każdej klatce. Scentralizowanie tego kodu ułatwia zarządzanie całością kodu, który jest często uruchamiany.

I podobnie — jeśli używane są obiekty Timer, istnieje narzut powiązany z tworzeniem i wywoływaniem zdarzeń z wielu obiektów Timer. Jeśli konieczne jest uruchomienie różnych operacji w różnych odstępach czasu, proponowane są następujące rozwiązania:

- Należy użyć minimalnej liczby operacji dotyczących obiektów Timer i grup odpowiednio do częstotliwości ich wykonywania.

  Na przykład: należy użyć jednego obiektu Timer dla często wykonywanych operacji i ustawić wyzwalanie z częstotliwością co 100 milisekund. Należy użyć innego obiektu Timer dla operacji wykonywanych z mniejszą częstotliwością i w tle oraz ustawić wyzwalanie co 2000 milisekund.

- Należy użyć jednego obiektu Timer i wyzwalać operacje przy osiągnięciu określonej wielokrotności wartości właściwości `delay` obiektu Timer.

  Na przykład: załóżmy, że istnieją operacje, które wg oczekiwań będą wykonywane co 100 milisekund, oraz inne operacje, które będą wykonywane co 200 milisekund. W tym celu należy użyć pojedynczego obiektu Timer z wartością `delay` równą 100 milisekund. W procedurze obsługi zdarzenia `timer` należy dodać instrukcję warunkową, która będzie uruchamiała operacje wykonywane co 200 milisekund co drugi raz. Poniższy przykład demonstruje tę technikę:

```
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;
}
```

Zatrzymywanie nieużywanych obiektów Timer.

Jeśli procedura obsługi `timer` obiektu Timer wykonuje operacje tylko w określonych warunkach, a żaden warunek nie jest spełniony, wówczas należy wywołać metodę `stop()` obiektu Timer.

W zdarzeniu `enterFrame` lub procedurach obsługi Timer należy zminimalizować liczbę zmian wyglądu wyświetlanego obiektu, które powodują konieczność ponownego wyświetlania ekranu.

W każdej klatce faza renderowania powoduje ponowne wyświetlenie części stołu montażowego, która uległa zmianie podczas tej klatki. Jeśli ponownie rysowany obszar jest duży lub jest mały, ale zawiera dużą liczbę obiektów albo złożone obiekty wyświetlane, wówczas środowisko wykonawcze wymaga dłuższego czasu na renderowanie. W celu sprawdzenia zakresu wymaganego renderowania należy użyć funkcji „Pokaż odrysowane obszary” w programie Flash Player w wersji z debugerem lub w środowisku AIR.

Więcej informacji na temat zwiększania wydajności często wykonywanych operacji zawiera następujący artykuł:

- Writing well-behaved, efficient, AIR applications (Pisanie poprawnie działających i wydajnych aplikacji AIR) (artykuł i przykładowa aplikacja autorstwa Arno Gourdol)

Więcej tematów Pomocy

„Izolowanie zachowań” na stronie 66

Obciążenia wywołane animacjami

W celu ograniczenia wykorzystania procesora należy ograniczyć stosowanie animacji. Efektem tego ograniczenia jest zmniejszone wykorzystanie procesora, pamięci operacyjnej i zwiększenie trwałości baterii.

Projektanci i programiści tworzący treść Flash na komputery stacjonarne często wykorzystują w aplikacjach wiele animacji ruchu. Podczas tworzenia treści przeznaczonych na urządzenia mobilne o niskiej wydajności należy zminimalizować stosowanie animacji ruchu. Ograniczenie ich stosowania ułatwia wykonywanie treści na urządzeniach o niższym stopniu zaawansowania.

Rozdział 4: Wydajność kodu ActionScript 3.0

Klasa Vector a klasa Array

W miarę możliwości należy korzystać z klasy Vector zamiast z klasy Array.

Klasa Vector umożliwia szybsze odczytywanie i zapisywanie niż klasa Array.

Prosty test wykazuje przewagę klasy Vector nad klasą Array. Poniższy kod stanowi test dla klasy 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
```

Poniższy kod stanowi test dla klasy 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
```

Ten przykład można dodatkowo zoptymalizować, określając długość wektora i ustawiając jego długość jako stałą:

```
// 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
```

Jeśli rozmiar wektora nie zostanie określony z góry, wówczas rozmiar będzie powiększany, gdy zabraknie miejsca na wektor. Przy każdej okazji wzrostu rozmiaru wektora będzie następowało przydzielenie nowego bloku pamięci. Bieżąca treść wektora zostanie skopiowana do nowego bloku pamięci. Dodatkowe przydzielanie i kopiowanie danych powoduje pogorszenie wydajności. Powyższy kod został zoptymalizowany pod względem wydajności poprzez określenie początkowego rozmiaru wektora. Jednak kod nie został zoptymalizowany pod względem łatwości jego modyfikowania. W celu zwiększenia łatwości modyfikowania należy zapisać często używaną wartość w stałej:

```
// 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
```

W miarę możliwości należy korzystać z interfejsów API obiektu Vector, ponieważ mogą one działać szybciej.

Interfejs API rysowania

Interfejs API rysowania umożliwia szybsze wykonywanie kodu.

W programie Flash Player 10 i środowisku AIR 1.5 udostępniono nowy interfejs API, który umożliwia wydajniejsze wykonywanie kodu. Ten interfejs API nie zwiększa wydajności renderingu, ale korzystanie z niego może znacznie skrócić program użytkownika. Mniejsza liczba wierszy kodu ActionScript może przekładać się na zwiększenie wydajności jego wykonywania.

Nowy interfejs API rysowania udostępnia następujące metody:

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

Uwaga: *W niniejszym podręczniku nie opisano metody* `drawTriangles()`*, która jest związana z pracą w 3D. Jednak ta metoda może zwiększać wydajność kodu ActionScript, ponieważ obsługuje rodzime odwzorowania tekstur.*

Poniższy kod jawnie wywołuje odpowiednią metodę dla każdej rysowanej linii:

```
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 );
```

Poniższy kod jest wykonywany szybciej niż poprzedni, ponieważ liczba wykonywanych wierszy jest mniejsza. Im bardziej skomplikowana jest ścieżka, tym większy jest wzrost wydajności po zastosowaniu metody `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 );
```

Metoda `drawGraphicsData()` umożliwia uzyskanie podobnego wzrostu wydajności.

Przechwytywanie i propagowanie zdarzeń

Przechwytywanie i propagowanie zdarzeń umożliwia ograniczenie korzystania z procedur obsługi zdarzeń.

Koncepcja przechwytywania i propagowania zdarzeń została zapoczątkowana w modelu zdarzeń zastosowanym w języku ActionScript 3.0. Wykorzystanie propagacji zdarzenia umożliwia zoptymalizowanie wykonania kodu ActionScript. W celu zwiększenia wydajności można zarejestrować procedurę obsługi zdarzenia w jednym obiekcie zamiast w wielu obiektach.

Przykład: wyobraźmy sobie tworzenie gry, w której użytkownik musi jak najszybciej klikać jabłka, które w ten sposób są niszczone. Gra usuwa z ekranu każde kliknięte jabłko i dodaje punkty do punktacji użytkownika. Czyha pokusa, aby w celu wykrycia zdarzenia `MouseEvent.CLICK` wywoływanego przez każde jabłko napisać następujący kod:

```
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 );
}
```

Kod wywołuje metodę `addEventListener()` dla każdej instancji Apple. Usuwa również każdy detektor po kliknięciu jabłka, stosując w tym celu metodę `removeEventListener()`. Jednak model zdarzenia w języku ActionScript 3.0 udostępnia fazę przechwytywania i propagacji dla niektórych zdarzeń, dzięki czemu można je wykrywać z nadrzędnego obiektu InteractiveObject. W rezultacie - możliwe jest zoptymalizowanie powyższego kodu i zminimalizowanie liczy wywołań metod `addEventListener()` i `removeEventListener()`. Poniższy kod korzysta z fazy przechwytywania w celu wykrywania zdarzeń z obiektu nadrzędnego:

```
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 );
}
```

Kod został uproszczony i znacznie zoptymalizowany. Zawiera również tylko jedno wywołanie metody `addEventListener()` w kontenerze nadrzędnym. Detektory nie są już rejestrowane w instancjach Apple, więc nie muszą być usuwane po kliknięciu jabłka. Moduł obsługi `onAppleClick()` może zostać dodatkowo zoptymalizowany poprzez zatrzymanie propagacji zdarzenia:

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

Faza propagacji może zostać wykorzystana do przechwycenia zdarzenia poprzez przekazanie `false` jako trzeciego parametru do metody `addEventListener()`:

```
// 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 );
```

Domyślną wartością dla parametru fazy przechwytywania jest `false`, więc można go pominąć:

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

Praca z pikselami

Do malowania pikseli należy używać metody `setVector()`.

Podczas malowania pikseli możliwa jest prosta optymalizacja poprzez stosowanie odpowiednich metod klasy BitmapData. Szybszym sposobem rysowania pikseli jest używanie metody `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 );
```

W przypadku korzystania z metod powolnych, takich jak `setPixel()` lub `setPixel32()`, należy użyć metod `lock()` i `unlock()` w celu przyspieszenia wykonywania kodu. W poniższym kodzie metody `lock()` i `unlock()` służą do zwiększania wydajności:

```
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
```

Metoda `lock()` klasy BitmapData blokuje obraz i zapobiega aktualizacji obiektów, które się do niego odwołują, w przypadku zmiany obiektu BitmapData. Na przykład: jeśli obiekt Bitmap odwołuje się do obiektu BitmapData, można zablokować obiekt BitmapData, zmienić go, a następnie odblokować. Obiekt Bitmap nie zostanie zmieniony, dopóki nie nastąpi odblokowanie obiektu BitmapData. W celu podwyższenia wydajności należy stosować tą metodę wraz z metodą `unlock()` przed i po wielokrotnych wywołaniach do metody `setPixel()` i `setPixel32()`. Wywołanie metod `lock()` i `unlock()` zapobiega zbędnym aktualizacjom treści wyświetlanej na ekranie.

Uwaga: *Ta technika nie poprawia wydajności w przypadku przetwarzania pikseli na bitmapie zamiast na liście wyświetlania (podwójne buforowanie). Jeśli obiekt bitmapy nie odwołuje się do buforu bitmapy, wówczas użycie `lock()` i `unlock()` nie zwiększa wydajności. Program Flash Player wykrywa, że nie istnieją żadne odwołania do buforu i dlatego bitmapa nie jest renderowana na ekranie.*

Metody, które iteracyjnie przechodzą przez piksele, takie jak `getPixel()`, `getPixel32()`, `setPixel()` i `setPixel32()`, mogą działać powoli, szczególnie na urządzeniach mobilnych. Jeśli to możliwe, należy korzystać z takich metod, które pobierają wszystkie piksele w jednym wywołaniu. W celu odczytywania pikseli należy korzystać z metody `getVector()`, która działa szybciej niż metoda `getPixels()`. Ponadto w miarę możliwości należy korzystać z interfejsów API, które operują na obiektach Vector, ponieważ takie wywołania prawdopodobnie będą działać szybciej.

Wyrażenia regularne

W celu prostego wyszukiwania i wyodrębniania ciągów znaków — zamiast wyrażeń regularnych — należy stosować metody klasy String, takie jak `indexOf()`, `substr()` lub `substring()`.

Niektóre operacje, które mogą być wykonywane za pomocą wyrażeń regularnych, mogą być również wykonywane za pomocą metod klasy String. Na przykład: w celu określenia, czy ciąg znaków zawiera inny ciąg, można użyć metody `String.indexOf()` lub zastosować wyrażenie regularne. Jeśli jednak dostępna jest metoda klasy String, wówczas ta metoda działa szybciej niż równoważne wyrażenie regularne i nie wymaga tworzenia innego obiektu.

Aby zgrupować elementy bez izolowania treści grupy w wyniku, należy w wyrażeniu regularnym zastosować grupę nieprzechwytującą („`(?:xxxx)`”) zamiast grupy („`(xxxx)`”).

W wyrażeniach regularnych o umiarkowanej złożoności części wyrażenia są często grupowane. Na przykład: w następującym wzorcu wyrażenia regularnego nawiasy tworzą grupę wokół tekstu „ab”. W konsekwencji kwantyfikator „+” ma zastosowanie do grupy zamiast do pojedynczego znaku:

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

Domyślnie zawartość każdej grupy jest „przechwytywana”. Zawartość każdej grupy we wzorcu można uzyskać jako część wyniku wykonania wyrażenia regularnego. Przechwytywanie wyników grupowania trwa dłużej i wymaga większej ilości pamięci, ponieważ wyniki grupowania są zawarte w obiektach. Alternatywnie można zastosować składnię grupy nieprzechwytującej poprzez dołączenie znaku zapytania i dwukropka po nawiasie otwierającym. Taka składnia określa, że znaki zachowują się jak grupa, ale nie są przechwytywane dla wyniku:

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

Zastosowanie składni grupy nieprzechwytującej jest szybsze i obciąża pamięć w mniejszym stopniu niż zastosowanie standardowej składni grupy.

Jeśli wydajność wyrażeń regularnych jest niska, należy rozważyć zastosowanie alternatywnego wzorca wyrażeń regularnych.

W niektórych przypadkach w celu testowania oraz identyfikowania określonego wzorca tekstu można wykorzystać więcej niż jeden wzorzec wyrażenia regularnego. Z różnych przyczyn niektóre wzorce są wykonywane szybciej niż wzorce alternatywne. Jeśli okaże się, że wyrażenie regularne powoduje wolniejsze działanie kodu, należy wziąć pod uwagę inne wzorce wyrażeń regularnych, które pozwolą uzyskać takie same wyniki. Następnie należy przetestować te wzorce w celu określenia, który z nich działa najszybciej.

Różne inne sposoby optymalizacji

Dla obiektu TextField należy stosować metodę `appendText()` zamiast operatora `+=`.

Przy pracy z właściwością `text` klasy TextField należy korzystać z metody `appendText()` zamiast z operatora `+=`. Metoda `appendText()` zapewnia większą wydajność.

Przykład: w poniższym kodzie wykorzystano operator `+=`, a wykonanie całej pętli trwa 1120 ms:

```
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
```

W poniższym przykładzie operator `+=` został zastąpiony metodą `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
```

Wykonanie kodu teraz trwa 847 ms.


W miarę możliwości aktualizacje pól tekstowych należy przeprowadzać poza pętlami.

Poniższy kod można dodatkowo zoptymalizować, stosując prostą technikę. Aktualizacja pola tekstowego w każdej pętli powoduje znaczne wykorzystanie zasobów procesora. Prosta konkatenacja ciągu znaków i przypisanie do niego pola tekstowego poza pętlą znacznie skraca czas wykonywania kodu. Wykonywanie poniższego kodu trwa 2 ms:

```
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
```

W przypadku pracy z tekstem HTML poprzedni kod jest wykonywany tak wolno, że w niektórych przypadkach zwraca wyjątek `Timeout` w programie Flash Player. Na przykład: wyjątek może zostać zgłoszony, jeśli sprzęt bazowy działa zbyt wolno.

Uwaga: *Adobe® AIR® nie zgłasza tego wyjątku.*

```
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 );
```

Przypisanie wartości do ciągu znaków poza pętlą powoduje, że kod jest wykonywany w zaledwie 29 ms:

```
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
```

Uwaga: *W programie Flash Player 10.1 i środowisku AIR 2.5 klasa String została udoskonalona, dzięki czemu ciągi znaków zajmują mniej miejsca w pamięci.*


Jeśli nie jest to konieczne, nie należy stosować operatora nawiasu kwadratowego.

Użycie operatora nawiasu kwadratowego może spowodować spowolnienie wykonywania kodu. Aby uniknąć konieczności użycia tego operatora, można zapisać odwołanie w zmiennej lokalnej. W poniższym przykładzie kodu przedstawiono niewydajne zastosowanie operatora nawiasu kwadratowego:

```
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
```

W poniższej zoptymalizowanej wersji ograniczono użycie tego operatora:

```
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
```

W miarę możliwości należy stosować kod wstawiany bezpośrednio, aby ograniczyć liczbą wywoływań funkcji w kodzie.

Wywoływanie funkcji może być kosztowne. Aby zmniejszyć liczbę wywołań funkcji, należy stosować kod wpisywany bezpośrednio. Stosowanie kodu bezpośredniego jest dobrą metodą optymalizacji wydajności. Należy jednak pamiętać o tym, że ponowne wykorzystanie kodu bezpośredniego może być trudniejsze, zaś ta technika programowania może powodować zwiększenie objętości pliku SWF. Niektóre wywołania funkcji, np. metod klasy Math, mogą być łatwo przenoszone bezpośrednio do kodu. Poniższy kod wykorzystuje metodę `Math.abs()` do obliczania wartości bezwzględnych:

```
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
```

Obliczenie realizowane przez metodę `Math.abs()` można łatwo zrealizować ręcznie i przenieść bezpośrednio do kodu:

```
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
```

Przeniesienie wywołania funkcji bezpośrednio do kodu powoduje ponadczterokrotny wzrost wydajności. Takie rozwiązanie jest użyteczne w wielu sytuacjach, ale należy pamiętać o wpływie, jaki może ono mieć na ewentualne ponowne wykorzystanie kodu i łatwości jego modyfikacji.

Uwaga: *Ilość kodu w znacznym stopniu wpływa na ogólne działanie odtwarzacza. Jeśli aplikacja zawiera znaczne ilości kodu ActionScript, wówczas maszyna wirtualna przeznacza znaczną ilość czasu na sprawdzanie poprawności kodu i kompilowanie JIT. Wyszukiwanie właściwości może być spowolnione z powodu głębszych hierarchii dziedziczenia i ponieważ wewnętrzne pamięci podręczne wykazują tendencję do częstszych awarii. Aby ograniczyć ilość kodu, należy unikać korzystania ze struktury Adobe® Flex®, biblioteki struktury TLF oraz wszelkich dużych bibliotek ActionScript innych firm.*

Należy również unikać określania wartości instrukcji w pętlach.

Innym sposobem optymalizacji może być rezygnacja z wyznaczania wartości instrukcji w każdej pętli. Poniższy kod wykonuje iteracje na tablicy, ale nie jest zoptymalizowany, ponieważ długość tablicy jest obliczana w każdej iteracji:

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

Lepszym rozwiązaniem jest zapisanie wartości i jej ponowne wykorzystanie:

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

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

Dla pętli while należy stosować kolejność odwrotną.

Pętla while wykonana w kolejności odwrotnej działa szybciej niż pętla wykonana do przodu:

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

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

Przedstawiono wskazówki optymalizacji kodu ActionScript oraz wyjaśniono, w jaki sposób jeden wiersz kodu może wpłynąć na wydajność i wykorzystanie pamięci. Istnieje również wiele innych metod optymalizacji kodu ActionScript. Aby uzyskać więcej informacji, należy skorzystać z następującego łącza: http://www.rozengain.com/blog/2007/05/01/some-actionscript-30-optimizations/.

Rozdział 5: Wydajność renderingu

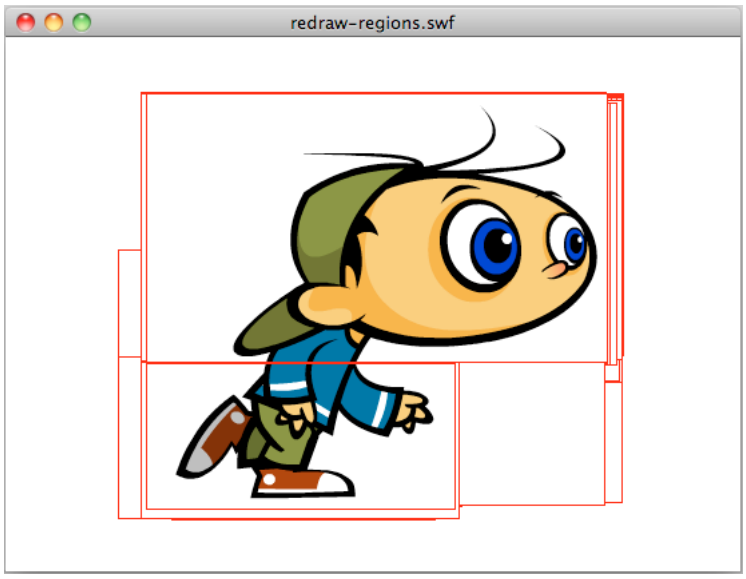
Odrysowanie obszarów

Podczas tworzenia projektów należy zawsze używać opcji odrysowania obszarów.

Aby udoskonalić renderowanie należy pamiętać, aby używać opcji odrysowania obszarów. Używanie tej opcji umożliwia wyświetlanie obszarów renderowanych i przetwarzanych przez program Flash Player. Tę opcję można włączyć, wybierając opcję Pokaż regiony odrysowywania w menu kontekstowym programu Flash Player w wersji z debugerem.

Uwaga: *Opcja Pokaż regiony odrysowywania nie jest dostępna w środowisku Adobe AIR ani w normalnej wersji programu Flash Player. (W środowisku Adobe AIR menu kontekstowe jest dostępne tylko w aplikacjach dla komputerów stacjonarnych, ale nie zawiera wbudowanych ani standardowych pozycji, takich jak polecenie Pokaż regiony odrysowywania).*

Poniższy obraz przedstawia włączoną opcję z prostym animowanym klipem filmowym na osi czasu:



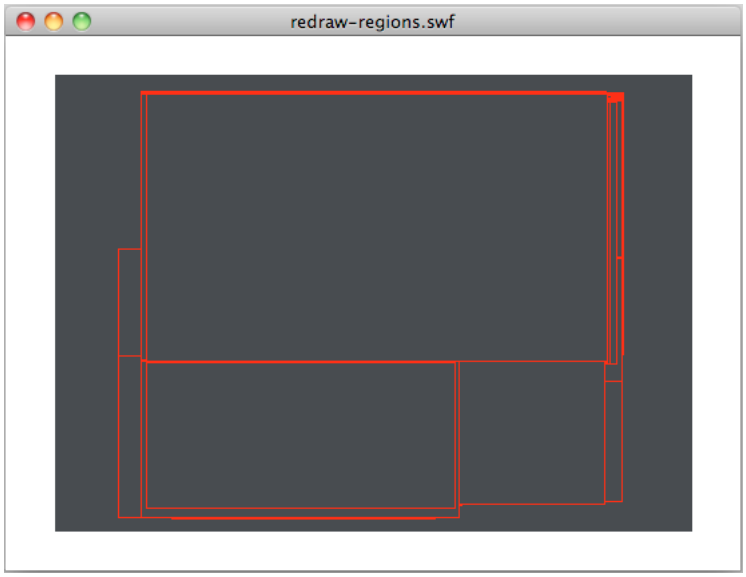
Włączona opcja odrysowania obszarów

Tę opcję można również włączyć programowo za pomocą metody `flash.profiler.showRedrawRegions()`:

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

W aplikacjach Adobe AIR ta metoda oferuje jedyny sposób włączenia opcji regionów odrysowywania.

Regiony odrysowywania ujawniają wykrycie operacji, które można zoptymalizować. Niektóre obiekty wyświetlane nie są widoczne, ale nadal używają cykli procesora, ponieważ są wciąż renderowane. Tę sytuację przedstawiono na poniższym obrazie. Czarny wektor kształtu pokrywa animowany, uruchomiony znak. Obraz pokazuje, że obiekt wyświetlany nie został usunięty z listy wyświetlania i jest nadal renderowany. Powoduje to niepotrzebne użycie zasobów procesora:



Odrysowane obszary

Aby poprawić wydajność, należy ustawić właściwość `visible` ukrytej biegnącej postaci na `false` lub usunąć tę postać z listy wyświetlania. Należy także zatrzymać oś czasu postaci. Wykonanie tych kroków powoduje wstrzymanie obiektu wyświetlanego i wykorzystanie minimalnej mocy procesora.

Należy pamiętać, aby opcji odrysowania obszarów używać w całym cyklu programowania. Użycie tej opcji zapobiega zaskoczeniu użytkownika pod koniec projektu przez niepotrzebne rysowanie i optymalizowanie obszarów, które mogły zostać pominięte.

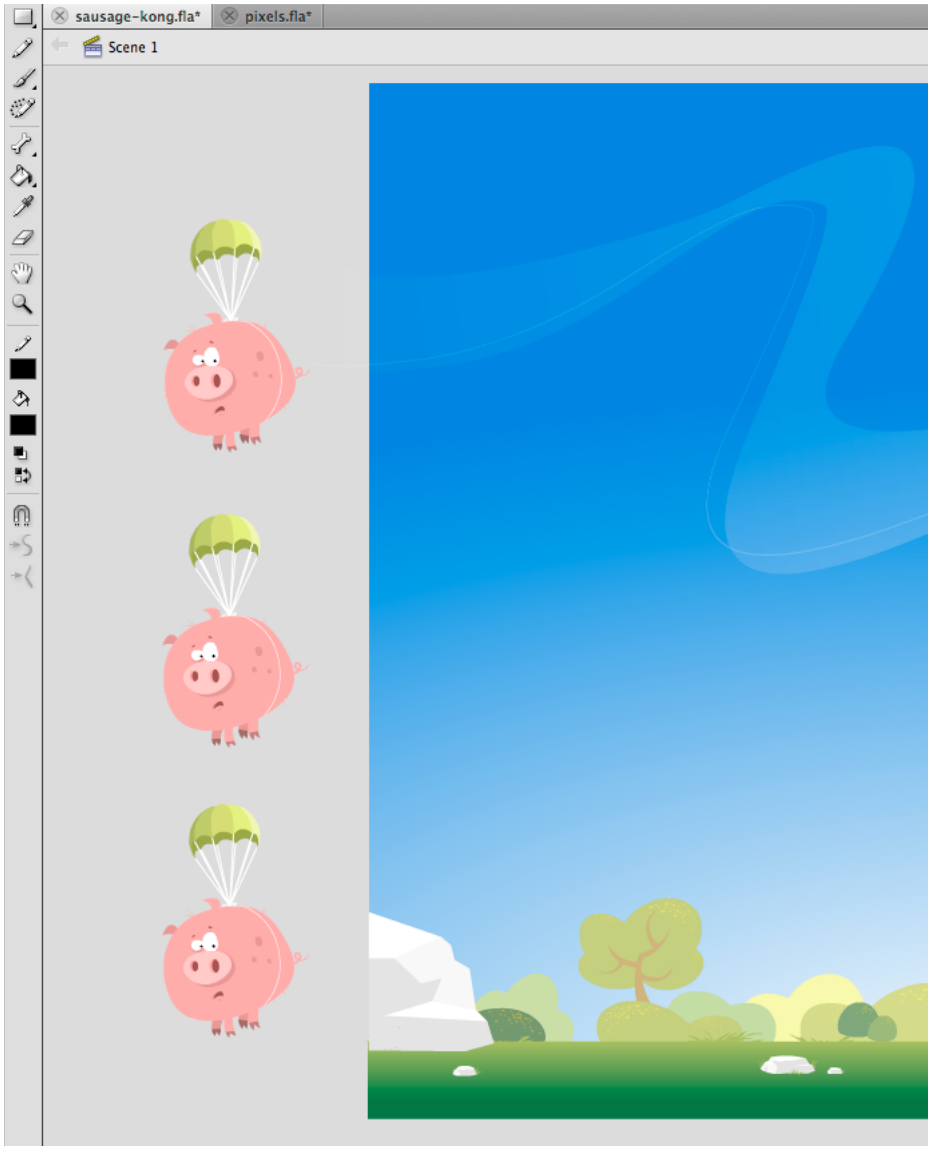
Więcej tematów Pomocy

„Wstrzymywanie i aktywowanie obiektów” na stronie 29

Zawartość poza sceną

Należy unikać umieszczania zawartości poza sceną. Zamiast tego należy umieszczać potrzebne obiekty na liście wyświetlania.

Jeśli jest to możliwe, nie należy umieszczać elementów graficznych poza sceną. Projektanci i programiści często pozostawiają elementy poza sceną, tak aby były dostępne jako zasoby aplikacji. Poniższa ilustracja demonstruje tę typową technikę.



Zawartość poza stołem montażowym

Nawet wtedy, gdy elementy poza sceną nie są renderowane ani wyświetlane na ekranie, nadal istnieją na liście wyświetlania. Środowisko wykonawcze wciąż wykonuje wewnętrzne testy związane z tymi elementami w celu zagwarantowania, że nadal pozostają poza sceną i użytkownik z nich nie korzysta. Z tego powodu należy w miarę możliwości unikać umieszczania obiektów poza sceną i zamiast tego mechanizmu stosować usuwanie z listy wyświetlania.

Jakość odtwarzanych filmów

W celu zwiększenia wydajności renderingu należy stosować odpowiednie ustawienia jakości stołu montażowego.

W przypadku tworzenia treści przeznaczonych na urządzenia mobilne z małymi ekranami, takie jak telefony, jakość obrazu jest mniej istotna niż w przypadku programowania aplikacji na komputery stacjonarne. Wybranie odpowiedniego ustawienia jakości stołu montażowego może spowodować zwiększenie wydajności renderingu.

Istnieją następujące ustawienia dotyczące jakości stołu montażowego:

- `StageQuality.LOW`: Szybkość odtwarzania ma większy priorytet niż wygląd. Nie jest stosowane wygładzanie. To ustawienie nie jest obsługiwane w środowisku Adobe AIR dla komputerów stacjonarnych i urządzeń telewizyjnych.
- `StageQuality.MEDIUM`: Wygładzanie jest stosowane w ograniczonym zakresie. Skalowane bitmapy nie podlegają wygładzaniu. To ustawienie stanowi wartość domyślną w środowisku AIR na urządzeniach przenośnych, ale nie jest obsługiwane w środowisku AIR na komputerach i urządzeniach telewizyjnych.
- `StageQuality.HIGH`: (Jest to domyślne ustawienie w wersji dla komputerów stacjonarnych). Wygląd ma większy priorytet niż szybkość odtwarzania. Zawsze jest stosowane wygładzanie. Jeśli plik SWF nie zawiera animacji, skalowane bitmapy są wygładzane. Jeśli plik SWF zawiera animacje, bitmapy nie są wygładzane.
- `StageQuality.BEST`: Zapewnia najlepszą możliwą jakość. Szybkość odtwarzania nie ma znaczenia. Wszystkie obiekty ekranowe, łącznie ze skalowanymi bitmapami, są wygładzane.

Ustawienie `StageQuality.MEDIUM` często zapewnia wystarczająco dobrą jakość w aplikacjach na urządzeniach przenośnych. Czasami także ustawienie `StageQuality.LOW` może zapewnić wystarczającą jakość. Począwszy od wersji Flash Player 8, wygładzony tekst może być dokładnie renderowany nawet w przypadku ustawienia jakości stołu montażowego na `LOW`.

Uwaga: *Na niektórych urządzeniach przenośnych mimo ustawienia jakości `HIGH` stosowana jest jakość `MEDIUM`, ponieważ zapewnia lepszą wydajność aplikacji programu Flash Player. Jednak ustawienie jakości na HIGH często nie wywołuje zauważalnej różnicy, ponieważ ekrany urządzeń mobilnych zwykle charakteryzują się wyższą rozdzielczością wyrażoną w punktach na cal (dpi). (Rozdzielczość może się różnić w zależności od urządzenia.)*

Na poniższym rysunku jakość odtwarzanego filmu jest ustawiona na `MEDIUM`, a rendering tekstu jest ustawiony na opcję Wygładź dla animacji:

Jakość stołu montażowego Medium i rendering tekstu ustawiony na opcję Wygładź dla animacji

Ustawienie jakości stołu montażowego wpływa na jakość tekstu, ponieważ odpowiednie ustawienie renderingu tekstu nie jest używane.

Środowisko wykonawcze umożliwia ustawienie renderowania tekstu z opcją Wygładź dla czytelności. To ustawienie zapewnia idealną jakość tekstu (wygładzonego) bez względu na to, jakie ustawienie jakości stołu montażowego jest używane:

Jakość stołu montażowego Low i rendering tekstu ustawiony na opcję Wygładź dla czytelności

Tę samą jakość renderingu można uzyskać poprzez ustawienie opcji renderingu tekstu na opcję Tekst bitmapowy (brak wygładzania):

Jakość stołu montażowego Low, a rendering tekstu ustawiony na Tekst bitmapowy (brak wygładzania)

Ostatnie dwa przykłady prezentują, że możliwe jest uzyskanie tekstu wysokiej jakości bez względu na to, które ustawienie stołu montażowego będzie używane. Ta funkcja była dostępna począwszy od programu Flash Player 8 i może być wykorzystywana na urządzeniach mobilnych. Należy pamiętać o tym, że program Flash Player 10.1 automatycznie aktywuje `StageQuality.MEDIUM` na niektórych urządzeniach w celu zwiększenia wydajności.

Mieszanie Alfa

W miarę możliwości należy unikać stosowania właściwości alpha.

W przypadku stosowania właściwości `alpha` należy unikać korzystania z efektów, które wymagają mieszania alfa. Gdy w obiekcie wyświetlanym jest stosowane mieszanie kanału alfa, środowisko wykonawcze musi połączyć wartości ze wszystkich nałożonych obiektów wyświetlanych i koloru tła w celu ustalenia koloru wynikowego. Oznacza to, że mieszanie alfa może bardziej obciążać procesor niż rysowanie nieprzezroczystego koloru. Takie dodatkowe obliczenia mogą znacznie obniżyć wydajność na urządzeniach o małej mocy obliczeniowej. W miarę możliwości należy unikać stosowania właściwości `alpha`.

Więcej tematów Pomocy

„Buforowanie bitmap” na stronie 55

„Renderowanie obiektów tekstowych” na stronie 69

Liczba klatek na sekundę w aplikacji

W ogólnym przypadku w celu osiągnięcia lepszej wydajności należy stosować najniższą możliwą liczbę klatek na sekundę.

Liczba klatek na sekundę w aplikacji określa ilość czasu dostępnego dla każdego „cyklu kodu aplikacji i renderowania”, co zostało opisane w sekcji „Fundamentalne zagadnienia wykonywania kodu w środowisku wykonawczym” na stronie 1. Większa liczba klatek na sekundę tworzy płynniejszą animację. Jeśli jednak nie występują żadne animacje ani zmiany graficzne, często nie ma potrzeby stosowania dużej szybkości odtwarzania. Wyższa szybkość odtwarzania powoduje używanie większej liczby cykli procesora i zużycie dodatkowej energii z akumulatora w porównaniu do niskiej szybkości odtwarzania.

Poniżej przedstawiono ogólne wytyczne dotyczące domyślnej liczby klatek na sekundę dla aplikacji:

- Jeśli używane jest środowisko Flex, należy pozostawić początkową liczbę klatek na sekundę jako wartość domyślną.
- Jeśli aplikacja zawiera animację, wówczas odpowiednia liczba klatek na sekundę wynosi co najmniej 20 klatek na sekundę. Wartości wyższe niż 30 klatek na sekundę zwykle są zbędne.
- Jeśli aplikacja nie zawiera animacji, wówczas wystarczająca liczba klatek będzie prawdopodobnie wynosić 12.

„Najniższa możliwa liczba klatek na sekundę” może się różnić w zależności od bieżącego działania aplikacji. Więcej informacji zawiera następna wskazówka „Dynamiczna zmiana liczby klatek na sekundę w aplikacji”.

Jeśli wideo jest jedyną dynamiczną treścią w aplikacji, również należy stosować niską liczbę klatek na sekundę.

Środowisko wykonawcze odtwarza treść wideo zgodnie z własną liczbą klatek na sekundę bez względu na liczbę klatek aplikacji. Jeśli aplikacja nie zawiera żadnej animacji ani innej szybko zmieniającej się treści wizualnej, wówczas zastosowanie niskiej liczby klatek na sekundę nie zmniejszy zadowolenia użytkownika.

Dynamiczna zmiana liczby klatek na sekundę w aplikacji.

Programista definiuje początkową liczbę klatek na sekundę w ustawieniach projektu lub kompilatora, ale ta wartość nie jest stała dla tej liczby. Liczbę klatek na sekundę można zmienić, ustawiając właściwość `Stage.frameRate` (lub właściwość `WindowedApplication.frameRate` w środowisku Flex).

Liczbę klatek na sekundę należy zmieniać odpowiednio do bieżących potrzeb aplikacji. Na przykład: gdy aplikacja nie wykonuje żadnej animacji, liczbę klatek na sekundę należy zmniejszyć. Przed rozpoczęciem animowanego przejścia należy zwiększyć liczbę klatek na sekundę. Analogicznie, jeśli aplikacja użytkownika jest uruchomiona w tle (po utracie aktywości), można zwykle zmniejszyć liczbę klatek na sekundę. Użytkownik prawdopodobnie pracuje na innej aplikacji lub zadaniu.

Poniżej przedstawiono ogólne wytyczne, które mogą być traktowane jako punkt odniesienia przy określaniu odpowiedniej liczby klatek na sekundę dla różnych typów operacji:

- Jeśli używane jest środowisko Flex, należy pozostawić początkową liczbę klatek na sekundę jako wartość domyślną.
- Podczas odtwarzania animacji liczbę klatek na sekundę należy ustawić na co najmniej 20. Wartości wyższe niż 30 klatek na sekundę zwykle są zbędne.

- Gdy żadna animacja nie jest odtwarzana, wówczas liczba klatek równa 12 będzie prawdopodobnie wystarczająca.
- Załadowane wideo jest odtwarzane z rodzimą liczbą klatek na sekundę bez względu na liczbę klatek w aplikacji. Jeśli wideo jest jedyną poruszającą się treścią w aplikacji, wówczas 12 klatek na sekundę będzie prawdopodobnie liczbą wystarczającą.
- Jeśli aplikacja nie zawiera punktu skupienia służącego do wstawiania, wówczas prawdopodobnie wystarczającą liczbą będzie 5 klatek na sekundę.
- Jeśli aplikacja AIR nie jest widoczna, liczba klatek na sekundę o wartości 2 lub mniejsza jest prawdopodobnie właściwa. Ta wskazówka powinna być brana pod uwagę na przykład wtedy, gdy aplikacja jest zminimalizowana. Dotyczy również komputerów stacjonarnych, na których właściwość `visible` okna macierzystego ma wartość `false`.

W aplikacjach utworzonych w środowisku Flex klasa spark.components. WindowedApplication udostępnia wbudowane funkcje obsługi przeznaczone do dynamicznej zmiany liczby klatek na sekundę w aplikacji. Właściwość `backgroundFrameRate` definiuje liczbę klatek na sekundę aplikacji, gdy aplikacja nie jest aktywna. Wartość domyślna wynosi 1, co powoduje zmianę liczby klatek na sekundę aplikacji utworzonej za pomocą środowiska Spark na 1 klatkę na sekundę. Liczbę klatek na sekundę dla tła można zmienić, ustawiając właściwość `backgroundFrameRate`. Właściwości można przypisać inną wartość, w szczególności wartość -1, aby wyłączyć automatyczne sterowanie liczbą klatek na sekundę.

Więcej informacji na temat dynamicznego zmieniania liczbę klatek na sekundę w aplikacji zawierają następujące artykuły:

- Reducing CPU usage in Adobe AIR (Zmniejszanie wykorzystania procesora w środowisku Adobe AIR) (artykuł i przykładowy kod autorstwa Jonnie'go Hallmana w portalu Adobe Developer Center)
- Writing well-behaved, efficient, AIR applications (Pisanie poprawnie działających i wydajnych aplikacji AIR) (artykuł i przykładowa aplikacja autorstwa Arno Gourdol)

Grant Skinner opracował klasę sterującą liczbą klatek na sekundę. Ta klasa może służyć w aplikacjach do automatycznego zmniejszania liczby klatek na sekundę, gdy aplikacja zostanie przesunięta do tła. Na stronie http://gskinner.com/blog/archives/2009/05/idle_cpu_usage.html jest dostępny artykuł Granta Skinnera „Używanie bezczynnego procesora w środowisku Adobe AIR i programie Flash Player”. Zawiera on dodatkowe informacje oraz pozwala pobrać kod źródłowy dotyczący klasy FramerateThrottler.

Adaptacyjna szybkość odtwarzania

Podczas kompilowania pliku SWF należy ustawić dla filmu odpowiednią liczbę klatek na sekundę. W środowisku z ograniczeniami technicznymi, przy niskiej częstości taktowania procesora, podczas odtwarzania może dochodzić do spadku liczby klatek na sekundę. W celu utrzymania szybkości odtwarzania akceptowalnej dla użytkownika środowisko wykonawcze pomija renderowanie niektórych klatek. Pomijanie renderowania niektórych klatek zapobiega zmniejszeniu szybkości odtwarzania poniżej akceptowalnego poziomu.

Uwaga: *W takiej sytuacji środowisko wykonawcze nie pomija samych klatek, a tylko renderowanie ich zawartości. Kod jest nadal wykonywany, a lista wyświetlania jest aktualizowana, ale aktualizacje nie są odzwierciedlane na ekranie. Nie ma sposobu na określenie progowej liczby klatek na sekundę wskazującej, ile klatek można pominąć, jeśli środowisko wykonawcze nie może utrzymać stabilnej szybkości odtwarzania.*

Buforowanie bitmap

W przypadku złożonych treści wektorowych należy korzystać (zgodnie z potrzebami) z buforowania bitmapy.

Dobrą optymalizację można uzyskać, używając funkcji buforowania bitmap. Ta funkcja buforuje obiekt wektorowy, renderuje go wewnętrznie jako bitmapę, a następnie korzysta z tej bitmapy przy renderowaniu. Rezultatem może być znaczny wzrost wydajności renderingu, jednak niekiedy kosztem miejsca w pamięci. Funkcja buforowania bitmapy jest przeznaczona dla złożonych treści wektorowych, takich jak złożone gradienty i tekst.

Włączenie buforowania bitmapy dla animowanego obiektu, który zawiera złożone grafiki wektorowe (takie jak tekst lub gradienty) powoduje zwiększenie wydajności. Jeśli jednak buforowanie bitmapy zostało włączone dla obiektu wyświetlanego np. klipu filmowego z odtwarzaną osią czasu, wynik będzie przeciwny. Dla każdej klatki środowisko wykonawcze musi zaktualizować buforowaną bitmapę, a następnie ponownie narysować ją na ekranie, co wymaga użycia wielu cykli procesora. Stosowanie funkcji buforowania bitmapy jest korzystne tylko wówczas, gdy buforowana bitmapa może zostać wygenerowana jednorazowo, a następnie może być używana bez konieczności aktualizacji.

Jeśli buforowanie bitmapy zostanie włączone dla obiektu Sprite, wówczas możliwe będzie przesunięcie obiektu bez ponownego generowania buforowanej bitmapy w środowisku wykonawczym. Zmiana właściwości `x` i `y` obiektu nie powoduje ponownego generowania. Jednak każda próba obrócenia obiektu, zmiany jego skali oraz zmiany jego wartości alfa powoduje, że środowisko wykonawcze ponownie generuje buforowaną bitmapę, czego rezultatem jest obniżenie wydajności.

Uwaga: *Ograniczenie to nie dotyczy właściwości* `DisplayObject.cacheAsBitmapMatrix` *dostępnej w środowisku AIR oraz w narzędziu Packager for iPhone. Dzięki właściwości* `cacheAsBitmapMatrix` *można obracać, skalować i pochylać obiekty wyświetlane oraz zmieniać ich wartości alfa, nie wywołując ponownego generowania bitmapy.*

Zbuforowana bitmapa może zajmować znacznie więcej pamięci niż zwykły klip filmowy. Na przykład: zbuforowany klip filmowy o rozdzielczości 250 x 250 pikseli może zajmować nawet 250 kB pamięci, podczas gdy ten sam klip w postaci niezbuforowanej będzie zajmował zaledwie 1 kB pamięci.

Poniższy przykład prezentuje obiekt klasy Sprite, który zawiera obraz jabłka. Do symbolu jabłka dołączana jest następująca klasa:

```
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
```

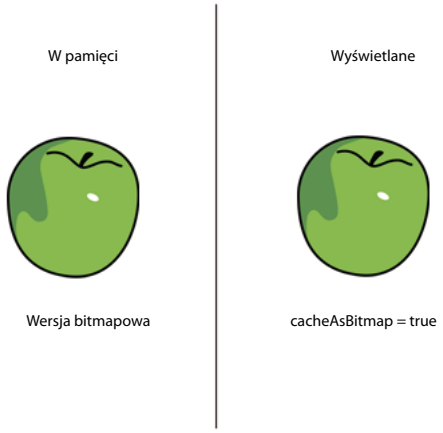
```
        {
            removeEventListener(Event.ENTER_FRAME,handleMovement);
        }

        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();
        }
    }
}
```

W kodzie wykorzystano klasę Sprite zamiast klasy MovieClip, ponieważ oś czasu nie jest wymagana dla każdego jabłka z osobna. W celu zapewnienia najlepszej wydajności należy użyć obiektu o możliwie najmniejszym rozmiarze. Następnie poniższy kod tworzy instancję klasy:

```
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);
        }
    }
}
```

Gdy użytkownik kliknie przycisk myszy, jabłka zostaną utworzone bez buforowania. Gdy użytkownik naciśnie klawisz C (kod 67), wektory jabłek zostaną zbuforowane jako bitmapy i przedstawione na ekranie. Ta technika znacznie zwiększa wydajność renderingu w komputerach stacjonarnych oraz na urządzeniach mobilnych wyposażonych w stosunkowo wolne procesory.

Funkcja buforowania bitmapy zwiększa wydajność renderingu, ale może powodować szybkie przydzielanie dużych ilości pamięci. Po zbuforowaniu obiektu jego powierzchnia zostaje zapisana w pamięci jako przezroczysta bitmapa, co przedstawia poniższy diagram:

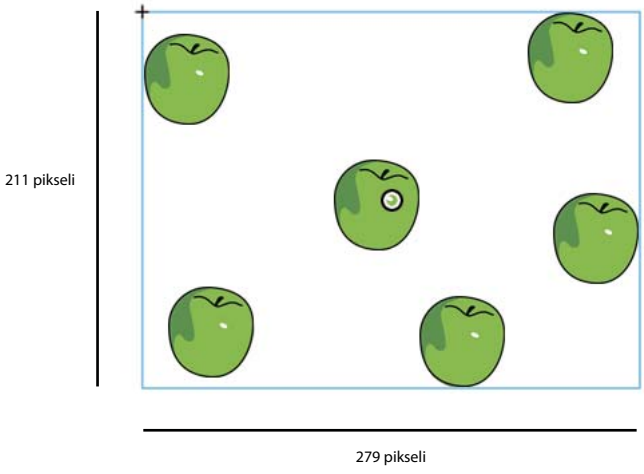


Obiekt i bitmapa jego powierzchni zapisane w pamięci

Program Flash Player 10.1 i środowisko AIR 2.5 optymalizują użycie pamięci, stosując metodę optymalizacji opisaną w sekcji „Filtry i dynamiczne usuwanie bitmapy” na stronie 20. Jeśli buforowany obiekt wyświetlany jest ukryty lub znajduje się poza ekranem, jego bitmapa w pamięci jest zwalniana, gdy nie jest używana przez pewien czas.

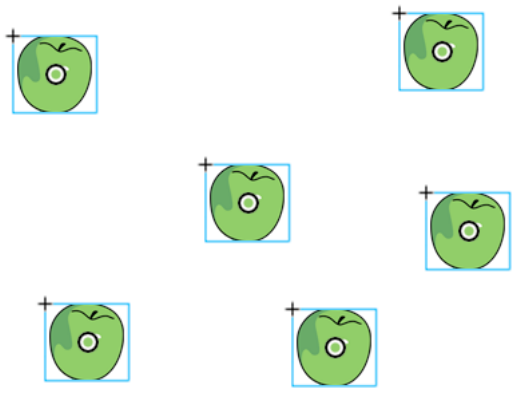
Uwaga: *Jeśli właściwość* `opaqueBackground` *obiektu wyświetlanego jest ustawiona na określony kolor, środowisko wykonawcze traktuje obiekt wyświetlany jako nieprzezroczysty. Jeśli używana jest właściwość* `cacheAsBitmap`*, środowisko wykonawcze tworzy nieprzezroczystą 32-bitową bitmapę w pamięci. Kanał Alfa jest ustawiony na 0xFF, co zwiększa wydajność, ponieważ w celu wyświetlenia bitmapy na ekranie nie jest wymagana przezroczystość. Brak mieszania Alfa powoduje jeszcze szybszy rendering. Jeśli bieżąca głębia barw jest ograniczona do 16 bitów, wówczas bitmapa w pamięci jest zapisana jako obraz 16-bitowy. Użycie właściwości* `opaqueBackground` *nie powoduje jawnej aktywacji buforowania bitmapy.*

W celu zmniejszenia obciążenia pamięci należy użyć właściwości `cacheAsBitmap` i aktywować ją na każdym obiekcie wyświetlanym, a nie na jednym kontenerze. Aktywacja buforowania bitmapy w kontenerze sprawia, że końcowa bitmapa zajmuje znacznie więcej miejsca w pamięci — jest zapisana jako przezroczysta bitmapa o wymiarach 211 x 279 pikseli. Obraz zajmuje około 229 kB:



Aktywacja buforowania bitmapy w kontenerze

Ponadto buforowanie kontenera ogranicza ryzyko aktualizacji całej bitmapy w pamięci, jeśli jakiekolwiek jabłko zacznie się poruszać w klatce. Aktywacja buforowania bitmapy dla poszczególnych instancji powoduje zbuforowanie w pamięci sześciu powierzchni o wielkości 7 kB, co skutkuje wykorzystaniem tylko 42 kB pamięci:



Aktywacja buforowania bitmapy dla instancji

Uzyskiwanie dostępu do poszczególnych instancji jabłka za pośrednictwem listy wyświetlania, a następnie wywoływanie metody `getChildAt()` powoduje zapisywanie odwołań w obiekcie Vector, co umożliwia łatwiejszy dostęp:

```
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);
        }
    }
}
```

Należy pamiętać o tym, że buforowanie bitmapy zwiększa zakres renderingu, jeśli buforowana treść nie będzie obracana, skalowana lub zmieniana w każdej klatce. Jednak nie powoduje ona zwiększenia wydajności renderingu w przypadku transformacji innych niż translacja na osiach x i y. W takich przypadkach program Flash Player aktualizuje zbuforowaną kopię bitmapy dla każdej transformacji, jaka ma miejsce w ramach obiektu wyświetlanego. Aktualizacja zbuforowanej kopii może spowodować powolne działanie oraz znaczące obciążenie procesora i baterii. Również to ograniczenie nie dotyczy właściwości `cacheAsBitmapMatrix` w środowisku AIR oraz w narzędziu Packager for iPhone

Poniższy kod zmienia wartość Alfa w metodzie Movement, co powoduje zmianę krycia jabłka w każdej klatce:

```
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();
}
```

Korzystanie z buforowania bitmapy wywołuje obniżenie wydajności. Każda zmiana wartości alfa zmusza środowisko wykonawcze do zaktualizowania bitmapy buforowanej w pamięci.

Filtry działają w oparciu o bitmapy, które są aktualizowane przy każdym przesunięciu głowicy odtwarzania zbuforowanego klipu filmowego. Dlatego zastosowanie filtru automatycznie powoduje ustawienie właściwości `cacheAsBitmap` na `true`. Poniższy rysunek ilustruje animowany klip filmowy:

Animowany klip filmowy

Filtry nie powinny być stosowane względem treści animowanych, ponieważ mogą one powodować problemy z wydajnością. Na poniższym rysunku widoczny jest efekt dodania filtru Cień:

Animowany klip filmowy z filtrem cienia

Rezultat jest następujący: jeśli oś czasu w klipie filmowym będzie odtwarzana, wówczas konieczne będzie ponowne wygenerowanie bitmapy. Bitmapa musi zostać ponownie wygenerowana także wówczas, gdy treść zostanie zmodyfikowana w dowolny sposób inny niż prosta transformacja x lub y. Dla każdej klatki środowisko wykonawcze musi ponownie narysować bitmapę, co powoduje wzrost obciążenia procesora, obniżenie wydajności i skrócenie czasu działania przy zasilaniu akumulatorowym.

Paul Trani przedstawia przykłady optymalizacji grafiki z bitmapami za pomocą programu Flash Professional i kodu ActionScript w następujących szkoleniach wideo:

- Optymalizacja grafiki
- Optymalizacja grafiki przy użyciu kodu ActionScript

Macierze przekształceń buforowanych bitmap w środowisku AIR

W przypadku używania buforowanych bitmap w aplikacjach AIR dla urządzeń przenośnych należy ustawić właściwość `cacheAsBitmapMatrix`.

W profilu AIR urządzeń przenośnych można przypisać obiekt Matrix do właściwości `cacheAsBitmapMatrix` obiektu wyświetlanego. Po ustawieniu tej właściwości można zastosować do obiektu dowolne przekształcenie dwuwymiarowe, nie powodując ponownego wygenerowania buforowanej bitmapy. Również zmiany właściwości alfa nie powodują ponownego wygenerowania buforowanej bitmapy. Właściwość `cacheAsBitmap` musi mieć wartość `true`, a dla obiektu nie mogą być ustawione żadne właściwości 3D.

Ustawienie właściwości `cacheAsBitmapMatrix` powoduje wygenerowanie buforowanej bitmapy nawet wtedy, gdy obiekt wyświetlany znajduje się poza ekranem (nie jest widoczny) lub ustawiono jego właściwość `visible` na `false`. Wyzerowanie właściwości `cacheAsBitmapMatrix` przy użyciu obiektu macierzy zawierającego inne przekształcenie także powoduje ponowne wygenerowanie buforowanej bitmapy.

Macierz przekształcenia zastosowana do właściwości `cacheAsBitmapMatrix` jest stosowana do obiektu wyświetlanego podczas jego renderowania do bufora bitmap. Oznacza to, że jeśli przekształcenie zawiera skalowanie oznaczające dwukrotne powiększenie, renderowana bitmapa jest dwa razy większa niż renderowany obraz wektorowy. Mechanizm renderujący stosuje do buforowanej bitmapy przekształcenie odwrotne, dzięki czemu obraz wynikowy wygląda tak samo. Buforowaną bitmapę można skalować w celu zmniejszenia rozmiaru i ograniczenia użycia pamięci, może to jednak pogorszyć jakość renderowanego obrazu. Skalowania bitmapy można także użyć w celu powiększenia jej rozmiaru. Niekiedy pozwala to poprawić jakość renderowanego obrazu, wymaga jednak większej ilości pamięci. Zazwyczaj należy używać macierzy jednostkowej (oznaczającej brak przekształcenia) w celu uniknięcia zmian wyglądu, tak jak to przedstawiono na poniższym przykładzie.

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

Po ustawieniu właściwości `cacheAsBitmapMatrix` można skalować, pochylać, obracać i przekształcać obiekt, nie wywołując ponownego generowania bitmapy.

Można również ustawić wartość alfa w zakresie od 0 do 1. W przypadku zmiany wartości alfa za pośrednictwem właściwości `transform.colorTransform` z przekształceniem koloru wartość alfa użyta w przekształceniu musi należeć do zakresu od 0 do 255. Każda inna zmiana przekształcenia koloru spowoduje ponowne wygenerowanie buforowanej bitmapy.

Zawsze należy ustawić właściwość `cacheAsBitmapMatrix` w przypadku ustawienia właściwości `cacheAsBitmap` na `true` w zawartości utworzonej dla urządzeń przenośnych. Należy wziąć pod uwagę następującą sytuację, w której to rozwiązanie może być niekorzystne. Po obróceniu, przeskalowaniu lub pochyleniu obiektu obraz uzyskany przez renderowanie może zawierać artefakty związane ze skalowaniem lub wygładzaniem, których nie będzie w przypadku normalnego renderowania wektorowego.

Ręczne buforowanie bitmap

W celu zrealizowania własnego mechanizmu buforowania bitmapy należy użyć klasy BitmapData.

W poniższym przykładzie przedstawiono pojedynczą rasteryzowaną wersję bitmapy obiektu wyświetlanego oraz odwołania do tego samego obiektu BitmapData. Skalowanie poszczególnych obiektów wyświetlanych nie powoduje aktualizacji ani ponownego rysowania oryginalnego obiektu BitmapData, który jest zapisany w pamięci. Dzięki temu procesor jest mniej obciążony, a aplikacje działają szybciej. Skalowanie obiektu wyświetlanego powoduje rozciąganie bitmapy, która znajduje się w kontenerze.

Oto zaktualizowana klasa `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();
        }
    }
}
```

Wartość alfa nadal jest modyfikowana w każdej klatce. Poniższy kod powoduje przekazanie oryginalnego buforu źródłowego do każdej instancji 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);
}
```

Wykorzystywana jest tylko niewielka część pamięci operacyjnej, ponieważ w pamięci używana jest tylko jedna buforowana bitmapa współużytkowana przez wszystkie instancje klasy `BitmapApple`. Ponadto: oryginalna bitmapa źródłowa nigdy nie jest aktualizowana z wyjątkiem modyfikacji (takich jak zmiana współczynnika alfa, obroty, zmiana skali), które dotyczą instancji klasy `BitmapApple`. Stosowanie tej techniki zapobiega obniżeniu wydajności.

W celu uzyskania wygładzonej końcowej bitmapy należy ustawić właściwość `smoothing` na `true`:

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

    smoothing = true;

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

Odpowiednia zmiana jakości stołu montażowego może powodować wzrost wydajności. Jakość stołu montażowego należy ustawić na `HIGH` przed rasteryzacją, a następnie przełączyć na `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 );
}
```

Zmiana jakości stołu montażowego przed rysowaniem i po narysowaniu wektora na bitmapie może być bardzo wydajną techniką wyświetlania wygładzonej treści na ekranie. Ta technika może być skuteczna bez względu na końcową jakość stołu montażowego. Na przykład: można uzyskać wygładzoną bitmapę z wygładzonym tekstem, nawet wówczas, gdy jakość stołu montażowego jest ustawiona na `LOW`. Ta technika nie jest dostępna w przypadku korzystania z właściwości `cacheAsBitmap`. W takim przypadku ustawienie jakości stołu montażowego na `LOW` powoduje aktualizację jakości wektora, co wywołuje aktualizację powierzchni bitmapy w pamięci i zmianę ostatecznej jakości.

Izolowanie zachowań

W miarę możliwości należ izolować zdarzenia, takie jak `Event.ENTER_FRAME` do pojedynczej procedury obsługi.

Kod można dodatkowo zoptymalizować poprzez odizolowanie zdarzenia `Event.ENTER_FRAME` klasy `Apple` w pojedynczej procedurze obsługi. Ta technika ogranicza wykorzystanie zasobów procesora. Poniższy przykład prezentuje inne podejście, w którym klasa BitmapApple nie obsługuje ruchu:

```
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;
        }
}
```

Poniższy kod tworzy instancje jabłek i obsługuje ich ruch w ramach jednej procedury obsługi:

```
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;
            }
        }
}
```

Wynikiem jest jedno zdarzenie `Event.ENTER_FRAME` obsługujące ruch, zamiast 200 procedur obsługi przesuwających poszczególne jabłka. Całą animację można łatwo wstrzymać, co może być użyteczne w grze.

Prosta gra może korzystać na przykład z takiej procedury obsługi:

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

Następnym etapem jest zaprogramowanie interakcji jabłek z myszą lub klawiaturą, co wymaga modyfikacji klasy `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);
        }
}
```

Wynikiem są interaktywne instancje klasy `BitmapApple` przypominające tradycyjne obiekty klasy Sprite. Jednak instancje są powiązane z pojedynczą bitmapą, której rozdzielczość nie jest zmieniana w przypadku modyfikacji obiektów wyświetlanych.

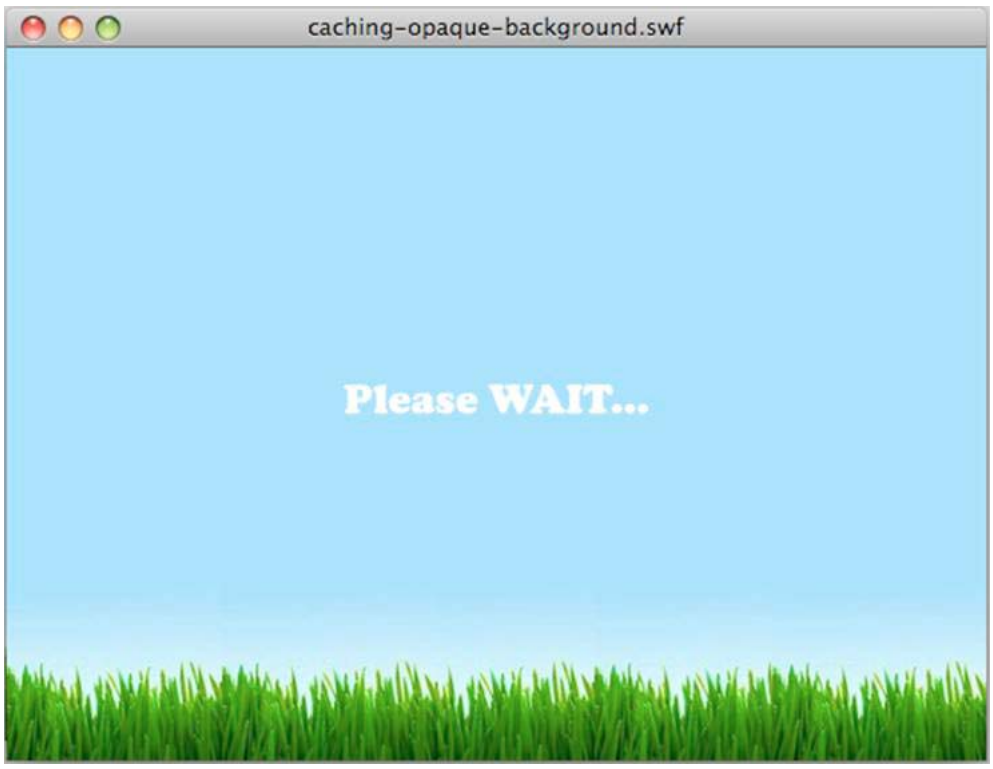
Renderowanie obiektów tekstowych

W celu zwiększenia wydajności renderowania tekstu należy użyć buforowania bitmapy i właściwości `opaqueBackground`.

Mechanizm Flash Text Engine oferuje pewnie znaczące optymalizacje. Jednak do wyświetlenia pojedynczego wiersza tekstu użyć trzeba wielu różnych klas. Dlatego do utworzenia edytowalnego pola tekstowego za pomocą klasy TextLine wymagana jest duża ilość pamięci i wiele wierszy kodu ActionScript. Klasa TextLine najlepiej nadaje się do obsługi statycznego tekstu niedostępnego do edycji — zapewnia wówczas szybsze renderowanie i wymaga mniejszej ilości pamięci.

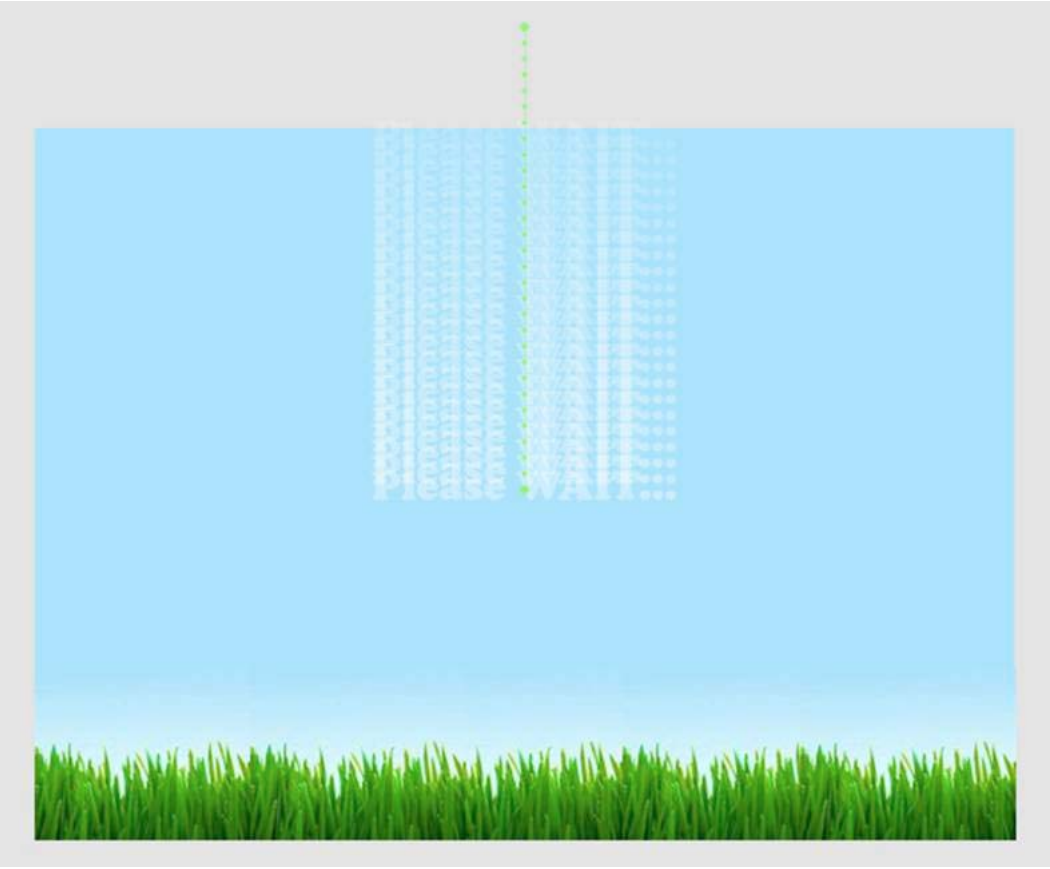
Funkcja buforowania bitmapy umożliwia buforowanie treści wektorowych jako bitmap w celu zwiększenia wydajności renderowania. Ta funkcja jest użyteczna w przypadku złożonych treści wektorowych, a także gdy jest używana z tekstem, którego renderowanie wymaga przetwarzania.

Poniższy przykład przedstawia, w jaki sposób można wykorzystać funkcję buforowania bitmapy i właściwość `opaqueBackground` do przyspieszenia renderingu. Poniższy rysunek przedstawia typowy ekran powitalny, który może zostać wyświetlony, gdy użytkownik oczekuje na załadowanie treści:



Ekran powitalny

Poniższy rysunek ilustruje programową zmianę dynamiki zastosowaną do obiektu TextField. Tekst jest powoli, niejednostajnie przemieszczany od góry do środka sceny:



Dynamiczny niejednostajny ruch tekstu

Poniższy kod generuje efekt dynamicznego niejednostajnego ruchu tekstu. Zmienna `preloader` zawiera bieżący obiekt docelowy, co ogranicza wyszukiwanie właściwości, które może znacznie zmniejszać wydajność:

```
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 );
}
```

Funkcja `Math.abs()` może zostać umieszczona w tym miejscu kodu w celu zmniejszenia liczby wywołań funkcji i dodatkowego zwiększenia wydajności. Dobrą praktyką jest zastosowanie typu int dla właściwości `destX` i `destY`, dzięki czemu dostępne będą wartości stałopozycyjne. Stosowanie typu int umożliwia uzyskanie idealnego przyciągania do pikseli bez konieczności ręcznego zaokrąglania wartości za pomocą powolnych metod, takich jak `Math.ceil()` i `Math.round()`. Ten kod nie zaokrągla współrzędnych do int, ponieważ ciągłe zaokrąglanie wartości powoduje, że obiekt nie porusza się płynnie. Ruchy obiektu mogą być drżące, ponieważ współrzędne są przyciągane do najbliższych zaokrąglonych liczb całkowitych w każdej klatce. Jednak ta technika może być użyteczna w przypadku ustawiania końcowego położenia obiektu wyświetlanego. Nie należy stosować poniższego kodu:

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

Poniższy kod działa znacznie szybciej:

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

Poprzedni kod można dodatkowo zoptymalizować poprzez użycie operatorów bitowych do dzielenia wartości:

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

Funkcja buforowania bitmapy ułatwia środowisku wykonawczemu renderowanie obiektów przez stosowanie dynamicznych bitmap. W bieżącym przykładzie zilustrowano buforowanie klipu filmowego zawierającego obiekt TextField:

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

Dodatkowym sposobem na zwiększenie wydajności jest usunięcie przezroczystości (alfa). Przezroczystość alfa wymaga dodatkowych obliczeń w środowisku wykonawczym podczas rysowania obrazów przezroczystych bitmap, co zademonstrowano w poprzednim przykładowym kodzie. Korzystając z właściwości `opaqueBackground`, można pominąć rysowanie poprzez określenie koloru jako tła.

W przypadku korzystania z właściwości `opaqueBackground` powierzchnia bitmapy utworzona w pamięci nadal wykorzystuje 32 bity. Jednak przesunięcie wartości alfa jest ustawione na 255 i nie jest wykorzystywana przezroczystość. W rezultacie właściwość `opaqueBackground` nie zmniejsza obciążenia pamięci, ale zwiększa wydajność renderowania w przypadku korzystania z buforowania bitmapy. W poniższym kodzie zastosowano wszystkie techniki optymalizacji:

```
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 );
}
```

Zoptymalizowano animację i buforowanie bitmapy poprzez usunięcie przezroczystości. Rozważmy zmianę jakości stołu montażowego na `LOW` i `HIGH` na urządzeniach przenośnych, w różnych stanach animacji i podczas korzystania z funkcji buforowania bitmapy:

```
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 );
    }
}
```

W tym przypadku zmiana jakości stołu montażowego zmusza środowisko wykonawcze do ponownego wygenerowania bitmapy powierzchni obiektu TextField w celu zapewnienia jej zgodności z aktualną jakością obiektu Stage (stołu montażowego). Z tego powodu zmiana jakości stołu montażowego nie jest zalecana, gdy używana jest funkcja buforowania bitmapy.

W tym przypadku możliwe jest także zastosowanie ręcznego buforowania bitmapy. W celu zasymulowania działania właściwości `opaqueBackground` klip filmowy można rysować w nieprzezroczystym obiekcie BitmapData, przez co środowisko wykonawcze nie będzie musiało ponownie generować bitmapy powierzchni.

Ta technika przynosi dobre rezultaty, pod warunkiem że treść nie ulega zmianie w miarę upływu czasu. Jeśli jednak treść pola tekstowego może ulec zmianie, należy rozważyć zastosowanie innej strategii. Wyobraźmy sobie na przykład pole tekstowe, w którym ciągle zmieniana jest wartość procentowa obrazująca postęp ładowania aplikacji. Jeśli to pole tekstowe (lub zawierający je obiekt wyświetlany) zostało zbuforowane jako bitmapa, jego powierzchnia musi zostać wygenerowana przy każdej zmianie zawartości pola. Nie można zastosować ręcznego buforowania bitmapy, ponieważ zawartość obiektu wyświetlanego nieustannie się zmienia. Zmiana tej stałej mogłaby wymusić ręczne wywołanie metody `BitmapData.draw()` w celu zaktualizowania buforowanej bitmapy.

Należy pamiętać o tym, że począwszy od programu Flash Player 8 (i środowiska AIR 1.0) — bez względu na jakość stołu montażowego — pole tekstowe z opcją renderowania Wygładź dla czytelności pozostaje idealnie wygładzone. Takie rozwiązanie zmniejsza zużycie pamięci, ale powoduje większe obciążenie procesora, a rendering przebiega nieznacznie wolniej niż w przypadku funkcji buforowania bitmapy.

W poniższym kodzie zastosowano takie rozwiązanie:

```
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 );
    }
}
```

Stosowanie tej opcji (Wygładź dla czytelności) w odniesieniu do tekstu w ruchu nie jest zalecane. W przypadku skalowania tekstu ta opcja powoduje podejmowanie prób zachowania wyrównania tekstu, co wywołuje efekt przesuwania. Jeśli zawartość obiektu wyświetlanego jest stale zmieniana, a wymagany jest tekst skalowany, można zwiększyć wydajność aplikacji dla urządzeń przenośnych przez ustawienie jakości `LOW`. Po zakończeniu ruchu można z powrotem przywrócić jakość `HIGH`.

Procesor graficzny GPU

Renderowanie przy użyciu GPU w aplikacjach programu Flash Player

Ważną nową funkcją programu Flash Player 10.1 jest to, że program może wykorzystywać procesor graficzny GPU do renderowania treści graficznych na urządzeniach mobilnych. W przeszłości grafiki były renderowane tylko za pomocą procesora CPU. Korzystanie z procesora graficznego optymalizuje renderowanie filtrów, bitmap, wideo i tekstu. Należy pamiętać o tym, że renderowanie za pomocą procesora graficznego nie zawsze jest tak dokładne, jak renderowanie programowe. W przypadku renderingu sprzętowego treść może wyglądać na pofragmentowaną. Ponadto w programie Flash Player 10.1 obowiązuje ograniczenie, które uniemożliwia rendering ekranowych efektów Pixel Bender. Wynikiem renderowania tych efektów z wykorzystaniem przyspieszenia sprzętowego może być czarny kwadrat.

W programie Flash Player 10 dostępna była funkcja przyspieszenia przy użyciu procesora graficznego (GPU, graphics processing unit), ale nie stosowano GPU do obliczeń związanych z grafiką. Używanie GPU było ograniczone tylko do wysyłania całej grafiki na ekran. W programie Flash Player 10.1 obliczenia są realizowane przy użyciu GPU, co może znacznie zwiększać szybkość renderowania. GPU zmniejsza również obciążenie procesora, co może być bardzo użyteczne na urządzeniach o ograniczonych zasobach, takich jak urządzenia przenośne.

Uruchomienie zawartości na urządzeniu przenośnym powoduje automatyczne włączenie trybu GPU w celu zapewnienia najwyższej dostępnej wydajności. Właściwość `wmode` nie musi już mieć ustawionej wartości `gpu` w celu zapewnienia renderowania przez GPU. Ustawienie właściwości `wmode` na `opaque` lub `transparent` powoduje wyłączenie przyspieszenia przy użyciu GPU.

Uwaga: *Program Flash Player dla komputerów stacjonarnych nadal realizuje renderowanie programowe przy użyciu procesora. Renderowanie programowe jest stosowane, ponieważ między sterownikami na komputerach stacjonarnych występują duże różnice, co może powodować dodatkowe uwydatnianie różnic w renderowaniu. Mogą także występować różnice w renderowaniu między komputerami stacjonarnymi a urządzeniami przenośnymi.*

Renderowanie przy użyciu GPU w aplikacjach AIR dla urządzeń przenośnych

W celu włączenia sprzętowego przyspieszania grafiki w aplikacji AIR należy dołączyć do deskryptora tej aplikacji parametr `<renderMode>gpu</renderMode>`. Nie można zmieniać trybów renderowania w czasie wykonywania. Na komputerach stacjonarnych ustawienie `renderMode` jest ignorowane. Obecnie na komputerach nie jest obsługiwane przyspieszanie grafiki przy użyciu GPU.

Ograniczenia trybu renderowania GPU

W przypadku używania trybu renderowania GPU w środowisku AIR 2.5 występują następujące ograniczenia:

- Jeśli GPU nie może renderować obiektu, obiekt w ogóle nie jest wyświetlany. Nie jest dostępna rezerwowa opcja renderowania przy użyciu procesora.
- Nie są obsługiwane tryby mieszania: layer, alpha, erase, overlay, hardlight, lighten, darken.
- Filtry nie są obsługiwane.
- Nie jest obsługiwana wtyczka PixelBender.
- W przypadku wielu jednostek GPU maksymalny rozmiar tekstury wynosi 1024 x 1024. W skryptach ActionScript przekłada się to na maksymalny ostateczny rozmiar renderowanego obiektu wyświetlanego po wszystkich przekształceniach.
- Firma Adobe nie zaleca używania trybu renderowania GPU w aplikacjach AIR odtwarzających materiały wideo.

- W trybie renderowania GPU pola tekstowe czasami nie są przenoszone w widoczne miejsca po otwarciu klawiatury wirtualnej. Aby mieć pewność, że pole tekstowe będzie widoczne, gdy użytkownik będzie wprowadzać tekst, należy wykonać jedną z następujących czynności: Umieść pole tekstowe w górnej połowie ekranu lub przenieś je do tej połowy, gdy stanie się aktywne.
- Tryb renderowania GPU jest wyłączony dla niektórych urządzeń, na których nie działa on stabilnie. Najnowsze informacje na ten temat można uzyskać w przeznaczonych dla programistów uwagach na temat wersji środowiska AIR.

Sprawdzone procedury dotyczące trybu renderowania GPU

Stosując się do następujących wskazówek, można uzyskać wydajniejsze renderowanie przy użyciu GPU:

- Należy ograniczyć liczbę elementów widocznych na stole montażowym. Renderowanie każdego elementu oraz komponowanie go z innymi elementami zajmuje pewien czas. Gdy obiekt wyświetlany ma przestać być widoczny, należy ustawić dla jego właściwości `visible` wartość `false`. Nie należy przenosić obiektu poza stół montażowy, ukrywać go za innym obiektem ani zmieniać jego właściwości `alpha` na 0. Jeśli obiekt wyświetlany nie będzie już w ogóle potrzebny, należy usunąć go ze stołu montażowego przy użyciu metody `removeChild()`.
- Zamiast tworzyć i niszczyć obiekty, należy używać obiektów wielokrotnie.
- Należy tworzyć bitmapy o wymiarach zbliżonych do 2^n na 2^m bitów, ale mniejszych. Wymiary nie muszą być dokładnie potęgami liczby 2, ale powinny być do nich zbliżone i nie powinny być od nich większe. Na przykład obraz o wymiarach 31 na 15 pikseli jest renderowany szybciej niż obraz o wymiarach 33 na 17 pikseli. (31 i 15 to o jeden mniej niż potęgi liczby 2: 32 i 16).
- W miarę możliwości należy, wywołując metodę `Graphic.beginBitmapFill()`, ustawić wartość false dla parametru repeat.
- Nie należy wymuszać zbyt wielu operacji rysowania. Jako tła należy używać koloru tła. Nie należy układać dużych kształtów warstwami — jeden na drugim. Narysowanie każdego piksela wymaga bowiem wykonania pewnych operacji.
- Należy unikać kształtów o długich, cienkich końcach, krawędzi przecinających się i wielu drobnych detali wzdłuż krawędzi. Renderowanie takich kształtów zajmuje znacznie więcej czasu niż wyświetlanie obiektów o równych krawędziach.
- Należy ograniczać rozmiary obiektów wyświetlanych.
- Dla obiektów wyświetlanych z rzadko zmienianą grafiką należy włączyć opcje cacheAsBitMap oraz cacheAsBitmapMatrix.
- Należy unikać interfejsu API rysowania ActionScript (klasy Graphics) jako mechanizmu tworzenia grafiki. Gdy jest to możliwe, należy zamiast tego tworzyć obiekty statycznie — podczas opracowywania zawartości.
- Zasoby bitmapowe należy przeskalować do rozmiarów docelowych przed zaimportowaniem.

Tryb renderowania GPU w środowisku AIR 2.0.3 dla urządzeń przenośnych

Renderowanie GPU podlega większym ograniczeniom w przypadku aplikacji AIR dla urządzeń przenośnych, które utworzono za pomocą narzędzia Packager for iPhone. Przydatność GPU jest ograniczona tylko do bitmap, kształtów wypełnionych i obiektów wyświetlanych z ustawioną właściwością `cacheAsBitmap`. Ponadto w przypadku obiektów, dla których ustawiono właściwości `cacheAsBitmap` i `cacheAsBitmapMatrix`, GPU może wydajnie renderować obiekty podlegające obracaniu i zmianom skali. Przetwarzanie pozostałych obiektów wyświetlanych również jest wykonywane przez GPU, co zazwyczaj skutkuje niską wydajnością renderowania.

Wskazówki dotyczące optymalizacji wydajności renderowania GPU

Renderowanie GPU może znacznie poprawić wydajność zawartości SWF, jest jednak ważne odpowiednie zaprojektowanie takiej zawartości. Ustawienia wcześniej dobrze działające dla renderowania programowego mogą nie być odpowiednie w przypadku renderowania GPU. Poniższe wskazówki pozwolą uzyskać wysoką wydajność renderowania GPU bez obniżania wydajności renderowania programowego.

Uwaga: *Urządzenia przenośne obsługujące renderowanie sprzętowe często uzyskują dostęp do zawartości SWF w Internecie. Z tego powodu warto skorzystać z poniższych porad podczas tworzenia zawartości SWF, tak aby oferowała ona możliwie najlepszą wydajność na dowolnym ekranie.*

- Należy unikać stosowania wartości `wmode=transparent` i `wmode=opaque` w parametrach osadzania HTML. Te tryby mogą skutkować obniżoną wydajnością. Mogą również pogorszyć synchronizację między dźwiękiem a materiałem wideo zarówno w przypadku renderowania programowego, jak i sprzętowego. Wiele platform nie obsługuje renderowania GPU, gdy jest aktywny jeden z tych trybów, co może znacznie obniżyć wydajność.
- Należy stosować tylko tryby normalny i mieszania alfa. Należy unikać używania innych trybów mieszania, szczególnie trybu mieszania warstw. Podczas renderowania GPU niektóre tryby mieszania nie są wiernie reprezentowane.
- Gdy GPU renderuje grafikę wektorową, przed rysowaniem tworzy siatkę złożoną z małych trójkątów. Proces ten jest określany jako tworzenie mozaiki. Tworzenie mozaiki wymaga wykonania pewnych obliczeń, których złożoność rośnie wraz ze złożonością kształtów. Aby ograniczyć wpływ tej operacji na wydajność, należy unikać zmiennych kształtów, które wymagają tworzenia mozaiki przez GPU w każdej klatce.
- Należy unikać stosowania przecinających się krzywych, bardzo cienkich obszarów ograniczonych krzywymi (na przykład cienkiego półksiężyca) oraz skomplikowanych szczegółów na krawędziach kształtów. Tworzenie siatek trójkątów z takich kształtów przez GPU wymaga złożonych obliczeń. Aby wyjaśnić tę kwestię, rozważmy dwa wektory: kwadrat 500 x 500 i półksiężyc 100 x 10. Renderowanie dużego kwadratu nie wymaga wielu obliczeń GPU, ponieważ są to tylko dwa trójkąty. Krzywa półksiężyca wymaga jednak wielu trójkątów do prawidłowego przedstawienia. Renderowanie tego kształtu jest więc znacznie bardziej skomplikowane, mimo iż zawiera on mniej pikseli.
- Należy unikać dużych zmian skali, ponieważ one również mogą wymuszać ponowne tworzenie mozaiki grafiki przez GPU.
- Gdy tylko jest to możliwe, należy unikać nakładania grafiki. Nakładanie grafiki polega na układaniu wielu warstw elementów graficznych w taki sposób, że elementy się nawzajem zasłaniają. W przypadku renderowania programowego każdy piksel jest rysowany tylko raz. Wydajność aplikacji nie zależy więc od liczby elementów graficznych zakrywających się wzajemnie w danym pikselu. Renderowanie sprzętowe powoduje natomiast rysowanie każdego piksela dla każdego elementu, nawet jeśli element jest w danym obszarze zasłonięty. Jeśli dwa prostokąty nakładają się na siebie, podczas renderowania sprzętowego obszar wspólny jest rysowany dwukrotnie (mimo iż byłby rysowany jeden raz w przypadku renderowania programowego).

  Nakładanie grafiki nie obniża wydajności na komputerach, gdyż jest w tym przypadku stosowane renderowanie programowe. Nakładające się kształty mogą natomiast pogorszyć wydajność na urządzeniach, na których jest używane renderowanie GPU. Sprawdzoną procedurą jest usuwanie obiektów z listy wyświetlania (zamiast ich ukrywania).
- Należy unikać stosowania dużego, wypełnionego prostokąta w charakterze tła. Zamiast tego należy ustawić kolor tła obiektu Stage.
- Należy unikać domyślnego trybu wypełniania bitmapą, który polega na powtarzaniu obrazu bitmapy we wszystkich możliwych miejscach. Zamiast tego należy stosować tryb ściskania, który oferuje lepszą wydajność.

Operacje asynchroniczne

W miarę możliwości należy stosować asynchroniczne wersje operacji zamiast synchronicznych.

Operacje synchroniczne są wykonywane, gdy tylko kod poleci ich wykonanie, a kod oczekuje na ich zakończenie. W konsekwencji takie operacje działają w fazie wykonywania kodu aplikacji w pętli klatki. Jeśli operacja synchroniczna jest wykonywana zbyt długo, wówczas operacja wydłuża czas pętli klatki, co może spowodować zatrzymanie wyświetlania lub wyświetlanie z przerwami.

Gdy kod jest wykonywany jako operacja asynchroniczna, nie musi być wykonywany natychmiast. W przypadku operacji asynchronicznych trwa wykonywanie właściwego kodu oraz innego kodu aplikacji w bieżącym wątku wykonania. Następnie środowisko wykonawcze jak najszybciej wykonuje operację, podejmując próby uniknięcia problemów z renderowaniem. W niektórych przypadkach wykonywanie odbywa się w tle i kod nie jest w ogóle wykonywany w ramach pętli klatki w środowisku wykonawczym. Gdy operacja zostanie ukończona, środowisko wykonawcze wywoła zdarzenie, a kod może wykryć to zdarzenie w celu wykonania dodatkowych operacji.

Operacje asynchroniczne są zaplanowane i dzielone w celu uniknięcia problemów z renderowaniem. W konsekwencji aplikacje, w których stosowane są asynchroniczne wersje operacji, działają dużo szybciej. Więcej informacji zawiera sekcja „Wydajność subiektywna a wydajność rzeczywista” na stronie 3.

Istnieje jednak narzut ujęty w operacjach uruchomionych asynchronicznie. Rzeczywisty czas wykonania może być dłuższy dla operacji asynchronicznych, szczególnie dotyczący operacji, których wykonanie zajmuje niewiele czasu.

W środowisku wykonawczym wiele operacji jest wykonywanych w sposób synchroniczny lub asynchroniczny i nie ma możliwości wybrania sposobu ich wykonywania. Jednak w środowisku Adobe AIR istnieją trzy typy operacji, które mogą być wykonywane synchronicznie lub asynchronicznie:

- Operacje klasy File i FileStream

  Wiele operacji klasy File może być wykonywanych synchronicznie lub asynchronicznie. Na przykład: wszystkie metody kopiowania lub usuwania plików i katalogów oraz metody wyświetlania zawartości katalogów mają wersje asynchroniczne. Te metody mają przyrostek „Async” dodany do nazwy wersji asynchronicznej. Na przykład: w celu asynchronicznego usuwania pliku należy wywołać metodę `File.deleteFileAsync()` zamiast metody `File.deleteFile()`.

  W przypadku stosowania obiektu FileStream do odczytywania z pliku lub zapisywania do pliku sposób otwierania obiektu FileStream określa, czy operacje zapisu i odczytu będą wykonywane w sposób asynchroniczny. Dla wszystkich operacji asynchronicznych należy stosować metodę `FileStream.openAsync()`. Zapis danych jest wykonywany metodą asynchroniczną. Odczyt danych odbywa się porcjami, dlatego w danym momencie dostępna jest tylko jedna porcja danych. I odwrotnie — w trybie synchronicznym obiekt FileStream odczytuje cały plik zanim przystąpi do wykonywania kodu.

- Operacje na lokalnej bazie danych SQL

  Podczas pracy z lokalną bazą danych SQL wszystkie operacje wykonywane przez obiekt SQLConnection są wykonywane w trybie synchronicznym lub asynchronicznym. W celu określenia, że operacje będą wykonywane asynchronicznie, należy otworzyć połączenie z bazą danych, korzystając z metody `SQLConnection.openAsync()` zamiast metody `SQLConnection.open()`. Jeśli operacje bazy danych działają asynchronicznie, wówczas są wykonywane w tle. Mechanizm bazy danych w ogóle nie działa w pętli klatki środowiska wykonawczego, dlatego prawdopodobieństwo, że operacje bazodanowe będą powodowały problemy z renderowaniem, jest dużo niższe.

  Dodatkowe strategie umożliwiające zwiększenie wydajności korzystania z lokalnej bazy danych SQL zawiera sekcja „Wydajność bazy danych SQL” na stronie 91.

- Autonomiczne moduły cieniujące Pixel Bender

Klasa ShaderJob umożliwia uruchomienie obrazu lub zestawu danych za pośrednictwem modułu cieniującego Pixel Bender oraz uzyskanie dostępu do nieprzetworzonych danych wynikowych. Domyślnie po wywołaniu metody `ShaderJob.start()` moduł cieniujący działa w trybie asynchronicznym. To wykonanie odbywa się w tle bez wykorzystania pętli klatki środowiska wykonawczego. W celu wymuszenia synchronicznego działania obiektu ShaderJob (co nie jest zalecane) należy przekazać `true` do pierwszego parametru metody `start()`.

Dostępne są zatem wbudowane mechanizmy umożliwiające asynchroniczne wykonywanie kodu, a oprócz tego programista może zapisywać kod w taki sposób, który zapewni jego asynchroniczne wykonywanie. W przypadku pisania kodu przeznaczonego do wykonywania potencjalnie długotrwałego zadania strukturę kodu można zdefiniować w taki sposób, aby kod był wykonywany w częściach. Jeśli kod jest podzielony na części, wówczas środowisko wykonawcze może wykonywać operacje renderowania pomiędzy blokami wykonywanego kodu, dzięki czemu występowanie problemów z renderowaniem będzie mniej prawdopodobne.

Poniżej przedstawiono kilka technik dzielenia kodu. Głównym celem tych technik jest taki zapis kodu, który w dowolnym czasie będzie gwarantował wykonanie tylko części operacji. Programista może śledzić, jakie operacje wykonuje kod oraz kiedy przestaje działać. Za pomocą mechanizmu, takiego jak obiekt Timer, programista może wielokrotnie sprawdzać, jaka część pracy została do wykonania, oraz wykonywać w porcjach dodatkowe operacje do zakończenia pracy.

Istnieje kilka predefiniowanych wzorców tworzenia struktury kodu w taki sposób, który zapewni podział operacji. Poniższe artykuły i biblioteki kodu opisują te wzorce i zawierają kod, który ułatwia implementowanie tych wzorców w aplikacjach:

- Asynchronous ActionScript Execution (Asynchroniczne wykonanie kodu ActionScript) (artykuł autorstwa Trevora McCauleya zawierający informacje szczegółowe oraz kilka przykładów implementacji)
- Parsing & Rendering Lots of Data in Flash Player (Analizowanie i renderowanie dużych ilości danych w programie Flash Player) (artykuł autorstwa Jesse'ego Wardena zawierający szczegółowe informacje oraz prezentujący dwa podejścia: „wzorzec budowania” i „zielone wątki”)
- Green Threads (Zielone wątki) (artykuł autorstwa Drew Cumminsa opisujący technikę „zielone wątki” z przykładowym kodem źródłowym)
- greenthreads (biblioteka kodu Open Source utworzona przez Charliego Hubbarda, zawiera informacje o implementacji „zielonych wątków” w kodzie ActionScript. Więcej informacji zawiera Podręcznik szybki start dotyczący zielonych wątków).
- Wątki w języku ActionScript 3 — http://www.adobe.com/go/learn_fp_as3_threads_pl (artykuł został napisany przez Alexa Harui i zawiera m.in. przykład implementacji techniki „pseudowątków”)

Przezroczyste okna

W aplikacjach AIR dla komputerów stacjonarnych zamiast stosowania przezroczystego okna należy rozważyć użycie nieprzezroczystego, prostokątnego okna aplikacji.

W celu użycia okna nieprzezroczystego dla początkowego okna aplikacji AIR dla komputerów stacjonarnych należy ustawić następującą wartość w pliku XML deskryptora aplikacji:

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

W przypadku okien tworzonych przez kod aplikacji należy utworzyć obiekt NativeWindowInitOptions z właściwością `transparent` ustawioną na `false` (domyślnie). Ten obiekt należy przekazać do konstruktora NativeWindow podczas tworzenia obiektu NativeWindow:

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

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

Dla składnika Window Flex należy się upewnić, że właściwość transparent jest ustawiona na false (wartość domyślna), a dopiero wówczas można wywołać metodę `open()` obiektu Window.

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

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

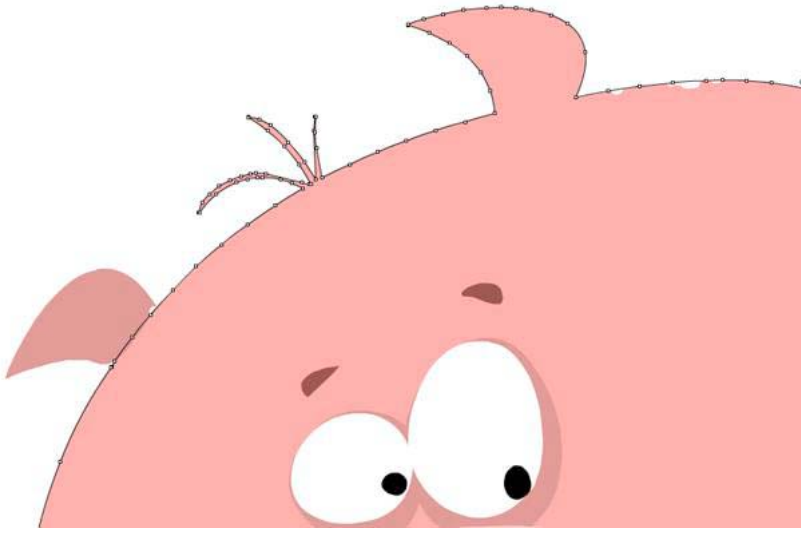
Przez przezroczyste okno można dostrzec części pulpitu lub okna innych aplikacji znajdujące się za oknem aktywnej aplikacji. W konsekwencji środowisko wykonawcze zużywa więcej zasobów w celu renderowania okna przezroczystego. W przypadku nieprzezroczystego okna prostokątnego — bez względu na to, czy stosowana jest karnacja systemowa, czy niestandardowa — obciążenie związane z renderowaniem jest znacznie niższe.

Okna przezroczyste powinny być stosowane tylko wówczas, gdy istotne jest, aby okno aplikacji było nieprostokątne lub aby przez okno aplikacji widoczna była treść znajdująca się w tle.

Wygładzanie kształtów wektorowych

Kształty należy w miarę możliwości wygładzać, aby uzyskać lepszą wydajność renderowania.

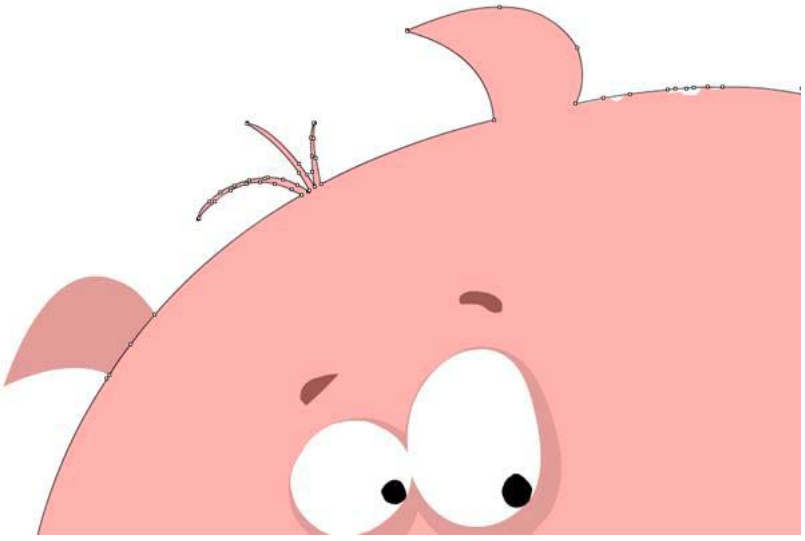
Renderowanie treści wektorowych, w przeciwieństwie do bitmapowych, wymaga wielu obliczeń, zwłaszcza przy generowaniu gradientów i skomplikowanych ścieżek zawierających wiele punktów sterujących. Projektant i/lub programista powinien zadbać o stosowną optymalizację kształtów. Na poniższej ilustracji przedstawiono ścieżki nieuproszczone zawierające wiele punktów sterujących:



Ścieżki niezoptymalizowane

Korzystając z narzędzia Wygładzanie w programie Flash Professional, można usunąć zbędne punkty sterujące. Równoważne narzędzie jest dostępne w programie Adobe® Illustrator®, a łączną liczbę punktów i ścieżek można odczytać w panelu Informacje o dokumencie.

Wygładzanie powoduje usunięcie dodatkowych punktów sterujących, a w konsekwencji zmniejszenie ostatecznej objętości pliku SWF i poprawę wydajności renderowania. Na następnej ilustracji przedstawiono te same ścieżki po wygładzeniu:



Ścieżki zoptymalizowane

Jeśli tylko uproszczenie ścieżek nie będzie zbyt daleko idące, taka optymalizacja nie zmienia efektu wizualnego widocznego dla użytkownika. Za to uproszczenie skomplikowanych ścieżek umożliwia znaczące zwiększenie częstości wyświetlania klatek gotowej aplikacji.

Rozdział 6: Optymalizacja interakcji z siecią

Ulepszenia poprawiające interakcje z siecią

W programie Flash Player 10.1 i środowisku AIR 2.5 wprowadzono zestaw nowych funkcji przeznaczonych do optymalizacji pracy w sieci na wszystkich platformach. Nowe funkcje obsługują na przykład buforowanie cykliczne oraz inteligentne przewijanie.

Buforowanie cykliczne

Podczas ładowania treści multimedialnych w urządzeniach mobilnych można napotkać problemy, które prawie nigdy nie zdarzają się w komputerach stacjonarnych. Na przykład: może zabraknąć miejsca na dysku lub pamięci operacyjnej. Podczas wczytywania wideo w wersjach programu Flash Player 10.1 lub środowiska AIR 2.5 dla komputerów stacjonarnych cały plik FLV (lub MP4) jest pobierany i buforowany na dysku twardym. Następnie środowisko wykonawcze odtwarza wideo z pliku bufora. Do całkowitego wyczerpania miejsca na dysku dochodzi bardzo rzadko. W takiej sytuacji środowisko wykonawcze na komputerze stacjonarnym przerywa odtwarzanie wideo.

Na urządzeniu przenośnym może szybciej zabraknąć miejsca na dysku. Jeśli na urządzeniu zabraknie miejsca na dysku, środowisko wykonawcze nie zatrzyma odtwarzania, jak w przypadku środowiska na komputerze stacjonarnym. Zamiast tego środowisko wykonawcze ponownie użyje pliku bufora, kontynuując zapisywanie od początku tego pliku. Użytkownik może kontynuować oglądanie wideo. Użytkownik nie może wyszukiwać w obszarze wideo, który został nadpisany, może jedynie przejść na początek pliku. Domyślnie buforowanie cykliczne nie jest uruchamiane. Może zostać uruchomione podczas odtwarzania, a także po rozpoczęciu odtwarzania, jeśli wielkość filmu przekracza ilość miejsca na dysku lub jego objętość jest większa od dostępnej ilości pamięci RAM. Środowisko wykonawcze wymaga co najmniej 4 MB pamięci RAM lub 20 MB miejsca na dysku — tylko wówczas możliwe jest buforowanie cykliczne.

Uwaga: *Jeśli urządzenie oferuje wystarczającą ilość miejsca na dysku, środowisko wykonawcze dla urządzeń przenośnych działa tak samo jak w wersji dla komputerów stacjonarnych. Należy pamiętać o tym, że bufor w pamięci RAM jest używany jako zabezpieczenie na wypadek, gdy urządzenie nie jest wyposażone w dysk lub dysk został zapełniony. Limit wielkości pliku pamięci podręcznej oraz buforu RAM może zostać ustawiony w czasie kompilacji. Niektóre pliki MP4 mają strukturę, która wymaga pobrania całego pliku zanim możliwe będzie jego odtworzenie. Środowisko wykonawcze wykrywa takie pliki i uniemożliwia ich pobieranie, jeśli stwierdzi brak wystarczającej ilości miejsca na dysku. Nie ma wtedy możliwości odtworzenia pliku MP4. Najlepiej, gdy te pliki nie będą w ogóle pobierane.*

Programista powinien pamiętać o tym, że wyszukiwanie działa tylko w części strumienia zapisanej w pamięci podręcznej. Działanie metody `NetStream.seek()` czasami kończy się niepowodzeniem, jeśli przesunięcie jest poza zakresem, a w takim przypadku wywoływane jest zdarzenie `NetStream.Seek.InvalidTime`.

Inteligentne wyszukiwanie

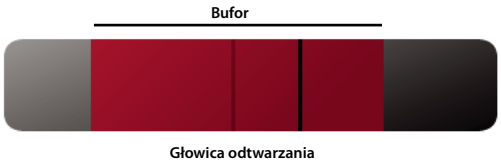
Uwaga: *Funkcja inteligentnego wyszukiwania wymaga programu Adobe® Flash® Media Server 3.5.3.*

Program Flash Player 10.1 i środowisko AIR 2.5 udostępniają nowe zachowanie, określane jako inteligentne przewijanie, które zwiększa komfort odtwarzania wideo przesyłanego strumieniowo. Jeśli użytkownik przewija do miejsca zawartego w buforze, środowisko wykonawcze korzysta z zawartości bufora, umożliwiając natychmiastowe przewinięcie. We wcześniejszych wersjach środowiska wykonawczego bufor nie był ponownie używany. Jeśli na przykład użytkownik odtwarzał wideo z serwera przesyłania strumieniowego, czas bufora (`NetStream.bufferTime`) został ustawiony na 20 sekund i użytkownik podjął próbę przejścia o 10 sekund do przodu, środowisko wykonawcze usuwało wszystkie dane z bufora, zamiast ponownie użyć wczytanego 10-sekundowego fragmentu. Takie działanie powodowało, że środowisko wykonawcze zbyt często żądało nowych danych z serwera, co wywoływało obniżenie wydajności odtwarzania i spowolnienie przesyłania.

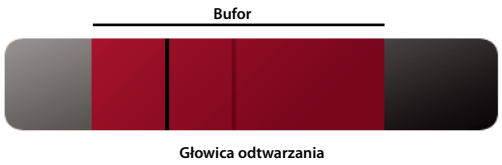
Poniższy rysunek przedstawia sposób działania bufora we wcześniejszych wersjach środowiska wykonawczego. Właściwość `bufferTime` określa liczbę sekund wideo do wstępnego wczytania, dlatego jeśli połączenie zostanie utracone, możliwe będzie korzystanie z bufora bez zatrzymywania wideo.

Zachowanie bufora przed wprowadzeniem funkcji inteligentnego przewijania

Dzięki funkcji inteligentnego przewijania środowisko wykonawcze korzysta z bufora w celu umożliwienia natychmiastowego cofania i przewijania do przodu, gdy użytkownik przewija wideo. Poniższy rysunek przedstawia nowe zachowanie.



Przewijanie do przodu przy użyciu funkcji inteligentnego przewijania



Cofanie przy użyciu funkcji inteligentnego przewijania

Funkcja inteligentnego wyszukiwania ponownie wykorzystuje bufor, gdy użytkownik przechodzi naprzód i wstecz, dzięki czemu odtwarzanie przebiega szybciej i bardziej płynnie. Jedną z zalet takiego działania jest zmniejszenie obciążeń łącz internetowych serwisów publikujących materiały wideo. Jeśli przewijanie sięga poza bufor, wykonywane są standardowe operacje, a środowisko wykonawcze żąda nowych danych z serwera.

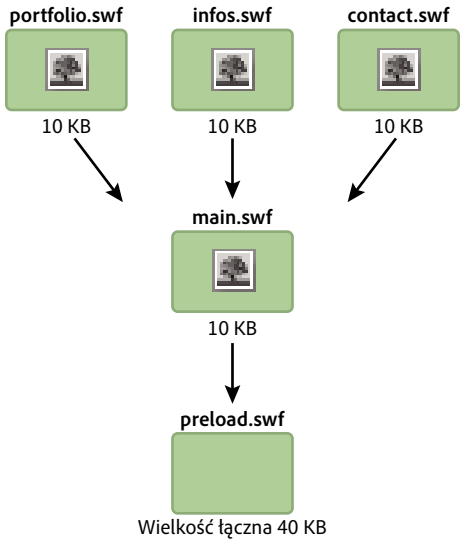
Uwaga: *To zachowanie nie dotyczy progresywnego pobierania wideo.*

Aby włączyć inteligentne przewijanie, należy przypisać do właściwości `NetStream.inBufferSeek` wartość `true`.

Treść zewnętrzna

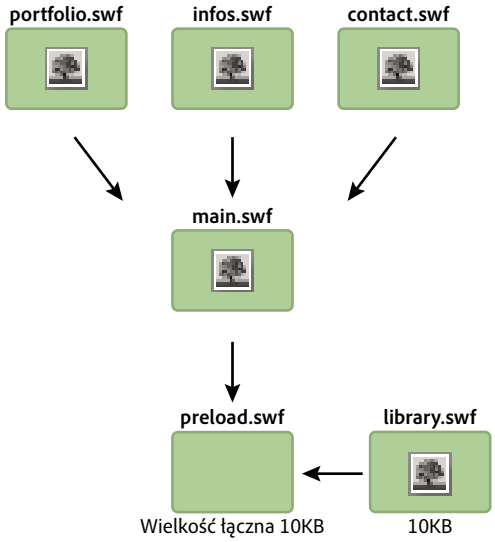
Aplikację należy podzielić na wiele plików SWF.

Urządzenia mobilne mogą mieć ograniczony dostęp do sieci. W celu szybkiego załadowania treści należy podzielić aplikację na wiele plików SWF. Należy dążyć do wielokrotnego używania fragmentów kodu i zasobów w całej aplikacji. Na przykład: rozważmy aplikację, która została podzielona na wiele plików SWF, co przedstawia poniższy diagram.



Aplikacja podzielona na wiele plików SWF

W tym przykładzie każdy plik SWF zawiera własną kopię tej samej bitmapy. Takiemu powielaniu można zapobiegać poprzez korzystanie z biblioteki RSL (Runtime Shared Library), co prezentuje poniższy diagram:



Użycie wspólnej biblioteki wykonawczej (RSL)

Gdy używana jest ta technika, biblioteka RSL zostaje załadowana w celu udostępnienia bitmapy innym plikom SWF. Klasa ApplicationDomain zawiera wszystkie definicje klas, które zostały załadowane, i udostępnia je w środowisku wykonawczym za pośrednictwem metody `getDefinition()`.

Biblioteka RSL może również zawierać całą logikę realizowaną przez kod. Cała aplikacja może zostać zaktualizowana w środowisku wykonawczym bez konieczności rekompilacji. Poniższy kod ładuje bibliotekę RSL i wyodrębnia definicję zawartą w pliku SWF w środowisku wykonawczym. Ta technika może być stosowana z czcionkami, bitmapami, dźwiękami lub dowolnymi klasami 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.");
}
```

Uzyskanie definicji może być łatwiejsze poprzez załadowanie definicji klasy w domenie aplikacji ładowanego pliku SWF:

```
// 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.");
}
```

Klasy dostępne w załadowanym pliku SWF mogą być używane poprzez wywołanie metody `getDefinition()` w bieżącej domenie aplikacji. Dostęp do klas można również uzyskać poprzez wywołanie metody `getDefinitionByName()`. Ta technika zmniejsza obciążenie łącz, ponieważ czcionki i duże zasoby są ładowane tylko raz. Zasoby nie są nigdy eksportowane w żadnych innych plikach SWF. Jedynym ograniczeniem jest to, że aplikacja musi zostać przetestowana i być uruchamiana za pośrednictwem pliku loader.swf. Ten plik najpierw ładuje zasoby, a następnie ładuje różne pliki SWF, które składają się na aplikację.

Błędy wejścia/wyjścia

Konieczne jest przygotowanie procedur obsługi zdarzeń i komunikatów o błędach wejścia/wyjścia.

W urządzeniu mobilnym dostęp do Internetu może być bardziej ograniczony niż w komputerze stacjonarnym podłączonym do szybkiego łącza internetowego. W przypadku uzyskiwania dostępu do treści zewnętrznej na urządzeniach mobilnych obowiązują dwa ograniczenia: dostępność i szybkość transferu. Dlatego należy dopilnować, aby zasoby były niewielkie oraz należy dodać procedury obsługi dla każdego zdarzenia IO_ERROR w celu ewentualnego poinformowania użytkownika o problemie.

Załóżmy na przykład, że użytkownik przegląda witrynę internetową na urządzeniu przenośnym i nagle traci połączenie sieciowe podczas przejazdu między dwiema stacjami metra. W momencie utraty połączenia akurat ładowany był zasób dynamiczny. W komputerze stacjonarnym można użyć pustego detektora zdarzeń, aby zapobiec pojawianiu się błędów środowiska wykonawczego, ponieważ prawdopodobieństwo realizacji takiego scenariusza jest znikome. Jednak w przypadku urządzenia mobilnego należy zastosować rozwiązanie bardziej rozbudowane niż prosty pusty detektor.

Poniższy kod nie reaguje na błędy wejścia/wyjścia. Przedstawiono nieprawidłowy kod, z którego nie należy korzystać:

```
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;
}
```

Lepszą metodą jest obsłużenie takiego błędu i wyświetlenie komunikatu o błędzie. Poniższy kod poprawnie obsługuje błąd:

```
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
}
```

Dobrą praktyką jest zaoferowanie użytkownikowi możliwości ponownego załadowania treści. Jest to możliwe w procedurze obsługi `onIOError()`.

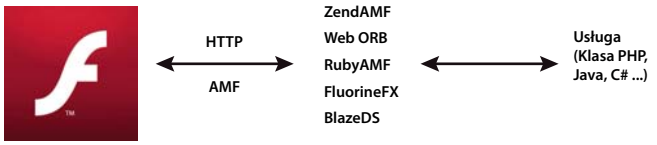
Flash Remoting

Funkcje Flash Remoting i format AMF zapewniają zoptymalizowane przesyłanie danych między klientem i serwerem.

Treść zdalną można ładować zdalnie do plików SWF w formacie XML. Dane XML są jednak zwykłym tekstem, który środowisko wykonawcze wczytuje i analizuje. XML działa najlepiej w aplikacjach, które ładują ograniczoną ilość treści. W przypadku tworzenia aplikacji, która ładuje znaczną ilość treści, należy rozważyć zastosowanie technologii Flash Remoting i formatu AMF (Action Message Format).

AMF jest formatem binarnym używanym do udostępniania danych między serwerem i środowiskiem wykonawczym. Stosowanie formatu AMF zmniejsza ilość danych i przyspiesza przesyłanie. AMF jest macierzystym formatem środowiska wykonawczego, dlatego wysyłanie danych AMF do środowiska wykonawczego umożliwia uniknięcie operacji szeregowania i deszeregowania po stronie klienta, które wymagają dużej ilości pamięci. Te zadania realizuje brama dostępu zdalnego. W przypadku wysłania danych typu ActionScript na serwer brama dostępu zdalnego realizuje serializowanie po stronie serwera. Ponadto brama wysyła do użytkownika dane odpowiedniego typu. Ten typ danych to klasa tworzona na serwerze, która udostępnia zestaw metod, jakie można wywołać w środowisku wykonawczym. Do bram Flash Remoting należą: ZendAMF, FluorineFX, WebORB i BlazeDS — oficjalna brama Java Flash Remoting firmy Adobe należąca do kategorii Open Source.

Poniższy rysunek ilustruje koncepcję Flash Remoting:



Flash Remoting

W poniższym przykładzie wykorzystano klasę NetConnection w celu nawiązania połączenia z bramą 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 ?");
```

Nawiązywanie połączenia z bramą dostępu zdalnego odbywa się w sposób bezpośredni. Jednak korzystanie z rozwiązania Flash Remoting można jeszcze bardziej ułatwić poprzez korzystanie z klasy RemoteObject zawartej w pakiecie Adobe® Flex® SDK.

Uwaga: *Zewnętrzne pliki SWC, np. pochodzące ze środowiska Flex, mogą być używane w projektach Adobe® Flash® Professional. Używanie plików SWC umożliwia korzystanie z klasy RemoteObject oraz jej zależności bez korzystania z pozostałych części pakietu Flex SDK. W razie potrzeby zaawansowani programiści mogą nawet komunikować się za pomocą bramy zdalnego dostępu za pośrednictwem surowej klasy Socket.*

Nadmierne operacje sieciowe

Pobrane zasoby powinny być umieszczane w buforze, a nie pobierane za każdym razem z sieci, gdy tylko są potrzebne.

Jeśli aplikacja wczytuje zasoby, takie jak multimedia lub dane, należy umieszczać je w buforze, zapisując je na urządzeniu lokalnym. W przypadku zasobów, które nie są zmieniane często, należy rozważyć aktualizowanie pamięci podręcznej w wybranych odstępach czasu. Na przykład: aplikacja może raz dziennie sprawdzać dostępność nowej wersji pliku obrazu lub sprawdzać dostępność świeżych danych co dwie godziny.

Zasoby można umieszczać w buforze na kilka sposobów, w zależności od typu i charakteru zasobu:

- Zasoby multimedialne, takie jak obrazy i wideo: pliki należy zapisać w systemie plików za pomocą klas File i FileStream
- Osobne wartości lub niewielkie zestawy danych: wartości należy zapisywać jako lokalne obiekty współużytkowane korzystając z klasy SharedObject
- Większe zestawy danych: dane należy zapisywać w lokalnej bazie danych lub szeregować je i zapisywać w pliku

Na potrzeby przechwytywania wartości danych projekt open source AS3CoreLib udostępnia klasę ResourceCache, która wykonuje operacje wczytywania i buforowania.

Rozdział 7: Praca z multimediami

Wideo

Informacje na temat optymalizowania wydajności odtwarzania wideo na urządzeniach przenośnych zawiera artykuł Optymalizacja zawartości internetowej na potrzeby urządzeń przenośnych dostępny w witrynie internetowej Adobe Developer Connection.

Należy zapoznać się przede wszystkim z następującymi sekcjami:

- *Odtwarzanie wideo na urządzeniach przenośnych*
- *Przykłady kodu*

Sekcje te zawierają informacje o programowaniu odtwarzaczy wideo dla urządzeń przenośnych, takie jak:

- Wytyczne dotyczące kodowania wideo
- Sprawdzone procedury
- Sposoby profilowania wydajności odtwarzacza wideo
- Dokumentacja implementacji odtwarzacza wideo

StageVideo

Klasa StageVideo umożliwia wyświetlanie wideo przy użyciu przyspieszania sprzętowego.

Informacje na temat używanie obiektu StageVideo zawiera sekcja Wyświetlanie z przyspieszaniem sprzętowym przy użyciu klasy StageVideo w dokumencie ActionScript 3.0 — Podręcznik programistów.

Dźwięk

Środowiska wykonawcze Flash Player 9.0.115.0 i AIR 1.0 oraz nowsze pozwalają odtwarzać pliki AAC (AAC Main, AAC LC oraz SBR). Prosta optymalizacja polega na stosowaniu plików AAC zamiast plików MP3. Format AAC oferuje lepszą jakość i mniejsze rozmiary plików niż format MP3 przy podobnej szybkości transmisji bitów. Zmniejszenie rozmiaru pliku ogranicza wykorzystanie przepustowości, co jest istotne w urządzeniach mobilnych, które nie oferują szybkich połączeń z Internetem.

Sprzętowe dekodowanie dźwięku

Dekodowanie dźwięku — podobnie jak dekodowanie wideo — zużywa znaczną część zasobów procesora i może zostać optymalizowane poprzez wykorzystanie dostępnych zasobów sprzętowych urządzenia. Program Flash Player 10.1 i środowisko AIR 2.5 może wykrywać i stosować sprzętowe sterowniki audio w celu zwiększenia wydajności podczas dekodowania plików AAC (profile LC, HE/SBR) lub plików MP3. (Metoda PCM nie jest obsługiwana). Powoduje to znaczne ograniczenie wykorzystania procesora, co wydłuża żywotność baterii i udostępnia procesor dla innych operacji.

Uwaga: *W przypadku korzystania z formatu AAC na urządzeniach nie jest obsługiwany profil AAC Main, ponieważ nie jest obsługiwany sprzętowo na większości urządzeń.*

Przebieg sprzętowego dekodowania audio jest oczywisty dla użytkowników i programistów. Gdy środowisko wykonawcze rozpoczyna odtwarzanie strumieni audio, najpierw sprawdza sprzęt, podobnie jak w przypadku wideo. Jeśli dostępny jest sterownik sprzętowy, a format audio jest obsługiwany, wówczas ma miejsce sprzętowe dekodowanie audio. Jednak w niektórych przypadkach sprzęt nie może przetwarzać wszystkich efektów, nawet jeśli może obsługiwać dekodowanie przychodzących strumieni AAC lub MP3. Czasami ograniczenia sprzętowe mogą na przykład powodować, że nie można przetwarzać miksowania i zmian próbkowania audio.

Rozdział 8: Wydajność bazy danych SQL

Projektowanie aplikacji w celu zwiększenia wydajności bazy danych

Po wykonaniu kodu nie należy zmieniać właściwości `text` *obiektu SQLStatement. Zamiast tego należy użyć jednej instancji SQLStatement dla każdej instrukcji SQL i użyć parametrów instrukcji w celu udostępnienia innych wartości.*

Przed wykonaniem instrukcji SQL środowisko wykonawcze przygotowuje (kompiluje) instrukcję w celu określenia kroków, jakie powinny zostać wykonane wewnętrznie w celu wykonania instrukcji. Po wywołaniu metody `SQLStatement.execute()` dla instancji klasy SQLStatement, która nie została wykonana poprzednio, instrukcja jest automatycznie przygotowywana zanim zostanie wykonana. Przy kolejnych wywołaniach metody `execute()` instrukcja jest nadal przygotowywana, pod warunkiem że właściwość `SQLStatement.text` nie została zmieniona. Dzięki temu jest wykonywana szybciej.

W celu maksymalnego wykorzystania instrukcji wykonywanych wielokrotnie, gdy wartości muszą zostać zmienione pomiędzy wykonywanymi instrukcjami, należy zastosować parametry instrukcji w celu dostosowania instrukcji. (Parametry instrukcji są określane za pomocą właściwości `SQLStatement.parameters` tablicy asocjacyjnej). W przypadku zmiany wartości parametrów instrukcji (w odróżnieniu od zmiany właściwości `text` instancji klasy SQLStatement) środowisko wykonawcze nie musi ponownie przygotowywać instrukcji.

Jeśli instancja klasy SQLStatement jest używana ponownie, aplikacja powinna zachować odwołanie do instancji SQLStatement, gdy zostanie ona przygotowana. W celu zachowania odwołania do instancji należy zadeklarować zmienną jako zmienną o zasięgu klasy, a nie jako zmienną o zasięgu funkcji. Dobrym sposobem na ustawienie instancji SQLStatement jako zmiennej o zasięgu klasy jest takie skonstruowanie aplikacji, aby instrukcja SQL była opakowana w pojedynczą klasę. Grupę instrukcji wykonywanych łącznie również można opakować w pojedynczą klasę. (Ta technika jest znana jako stosowanie wzorca projektowego Command). Zdefiniowanie instancji jako zmiennych należących do klasy powoduje, że będą one istniały tak długo, jak instancja klasy opakowującej będzie istniała w aplikacji. Wystarczy również zdefiniować zmienną zawierającą instancję SQLStatement na zewnątrz funkcji, dzięki czemu instancja zostanie zachowana w pamięci. Przykład: należy zadeklarować instancję SQLStatement jako zmienną klasy ActionScript lub jako zmienną (która nie jest funkcją) w pliku JavaScript. Następnie można ustawić wartości parametrów instrukcji i wywołać metodę `execute()` w celu rzeczywistego uruchomienia zapytania.

W celu zwiększenia szybkości porównywania i sortowania danych należy zastosować indeksy bazy danych.

Podczas tworzenia indeksu dla kolumny w bazie danych jest zapisywana kopia danych tej kolumny. Kopia jest posortowana w porządku liczbowym lub alfabetycznym. Dzięki sortowaniu baza danych może szybko dopasować wartości (np. podczas korzystania z operatora równości) i sortować dane wyników za pomocą klauzuli `ORDER BY`.

Aktualność indeksów bazy danych jest stale zachowana, przez co operacje zmiany danych (INSERT lub UPDATE) w tabeli działają nieznacznie wolniej. Jednak wzrost szybkości odczytu danych może być znaczący. Z powodu tego kompromisu nie należy po prostu indeksować każdej kolumny każdej tabeli. W zamian należy użyć strategii do definiowania indeksów. Poniższe wytyczne ułatwiają zaplanowanie strategii indeksowania:

- Indeksowaniem należy obejmować kolumny, które są używane w łączeniu tabel, w klauzulach WHERE oraz w klauzulach ORDER BY
- Jeśli kolumny są często wykorzystywane razem, należy utworzyć dla nich jeden wspólny indeks

- W przypadku kolumny zawierającej dane posortowane w porządku alfabetycznym należy określić porównanie COLLATE NOCASE dla indeksu


Należy rozważyć wstępne kompilowanie instrukcji SQL w trakcie bezczynności aplikacji.

Przy pierwszym uruchomieniu instrukcja SQL działa wolniej, ponieważ tekst SQL jest przygotowany (skompilowany) przez mechanizm bazy danych. Przygotowanie i wykonanie instrukcji może być wymagające, dlatego dobrym sposobem postępowania jest wstępne załadowanie danych, a następnie wykonanie innych instrukcji w tle.

1. Najpierw należy załadować dane, których aplikacja potrzebuje na samym początku.
2. Inne instrukcje należy wykonać po zakończeniu wstępnych operacji początkowych w aplikacji lub po wystąpieniu kolejnego momentu „bezczynności” w aplikacji.

Na przykład: załóżmy, że w celu wyświetlenia ekranu początkowego aplikacja nie musi uzyskiwać dostępu do bazy danych. W takim przypadku przed otwarciem połączenia z bazą danych należy poczekać na wyświetlenie ekranu. Na koniec należy utworzyć instancje SQLStatement i wykonać dowolne z nich.

Inna możliwość: załóżmy, że uruchomiona aplikacja natychmiast wyświetla jakieś dane, np. wynik określonego zapytania. W takim przypadku należy uruchomić instancję SQLStatement dla tego zapytania. Po początkowym załadowaniu i wyświetleniu danych należy utworzyć instancje klasy SQLStatement dla innych operacji bazy danych, a jeśli to możliwe — wykonać inne instrukcje, które będą wymagane później.

W rzeczywistości w przypadku ponownego wykorzystania instancji SQLStatement czas wymagany do przygotowania instrukcji to wyłącznie koszt jednorazowy. Prawdopodobnie nie wywiera dużego wpływu na ogólną wydajność.


W transakcji należy zgrupować wiele operacji zmiany danych SQL.

Załóżmy, że wykonywanych jest wiele instrukcji SQL, które obejmują dodawanie lub zmianę danych (instrukcje `INSERT` lub `UPDATE`). Wykonanie wszystkich instrukcji w jawnej transakcji powoduje znaczny wzrost wydajności. Jeśli transakcja nie zostanie rozpoczęta jawnie, każda instrukcja będzie działała we własnej automatycznie tworzonej transakcji. Po zakończeniu wykonywania każdej transakcji (każdej instrukcji) środowisko wykonawcze zapisuje dane wynikowe do pliku bazy danych na dysku.

Rozważmy jednak, co się stanie po jawnym utworzeniu transakcji i wykonaniu instrukcji w kontekście transakcji. Środowisko wykonawcze wykona wszystkie zmiany w pamięci, a następnie zapisze jednorazowo wszystkie zmiany w pliku bazy danych po wykonaniu transakcji. Zapisywanie danych na dysku zwykle zajmuje najwięcej czasu w operacji. W konsekwencji jednorazowe zapisanie na dysk zamiast zapisywania przy każdej instrukcji SQL może spowodować znaczny wzrost wydajności.


Wiele wyników zapytania `SELECT` należy przetwarzać w częściach, korzystając z metody `execute()` (z parametrem `prefetch`) oraz metody `next()` klasy SQLStatement.

Załóżmy, że wykonywana jest instrukcja SQL, która odczytuje duży zestaw wyników. Następnie aplikacja przetwarza każdy wiersz danych w pętli. Na przykład: formatuje dane lub tworzy z nich obiekty. Przetwarzanie tych danych może trwać długo, co może powodować problemy z renderowaniem, np. wstrzymanie ekranu lub brak reakcji na działania użytkownika. Zgodnie z opisem w sekcji „Operacje asynchroniczne” na stronie 77 odpowiednim rozwiązaniem jest podział pracy na porcje. Interfejs API bazy danych SQL sprawia, że podział przetwarzania danych jest łatwy do przeprowadzenia.

Metoda `execute()` klasy SQLStatement udostępnia opcjonalny parametr `prefetch` (pierwszy parametr). W przypadku wprowadzenia wartości ten parametr określa maksymalną liczbę wierszy wyników, jakie zwraca baza danych po zakończeniu wykonywania:

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

Gdy zostanie zwrócony pierwszy zestaw danych wynikowych, można wywołać metodę `next()`, aby kontynuować wykonywanie instrukcji i pobrać inny zestaw wierszy wyników. Metoda `next()` — podobnie jak metoda `execute()` — akceptuje parametr `prefetch` w celu określenia maksymalnej liczby zwracanych wierszy:

```
// 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);
        }
    }
}
```

Metodę `next()` można wywoływać tak długo, aż zostaną załadowane wszystkie dane. Poprzedni listing przedstawia, że możliwe jest określenie, czy wszystkie dane zostały załadowane. W tym celu należy sprawdzić stan właściwości `complete` obiektu SQLResult, który jest tworzony za każdym razem, gdy zakończone zostanie działanie metody `execute()` lub `next()`.

Uwaga: *Aby podzielić przetwarzanie danych wynikowych, należy użyć parametru `prefetch` oraz metody `next()`. Nie należy używać tego parametru ani metody do ograniczania wyników zapytania do części zestawu jego wyników. Jeśli konieczne jest pobranie jedynie podzestawu wierszy w zestawie wynikowym instrukcji, należy użyć klauzuli `LIMIT` instrukcji `SELECT`. Jeśli zestaw wyników jest duży, nadal można skorzystać z parametru `prefetch` i metody `next()` w celu podzielenia przetwarzania wyników.*

Należy rozważyć zastosowanie wielu asynchronicznych obiektów SQLConnection z pojedynczą bazą danych w celu jednoczesnego wykonania wielu instrukcji.

Gdy obiekt SQLConnection zostanie połączony z bazą danych za pomocą metody `openAsync()`, wówczas działa w tle, a nie w głównym wątku wykonania w środowisku wykonawczym. Ponadto każdy obiekt SQLConnection działa we własnym wątku w tle. Korzystając z wielu obiektów SQLConnection można skutecznie uruchamiać wiele instrukcji SQL jednocześnie.

Istnieją także potencjalne wady takiego podejścia. Najważniejszą z nich jest to, że każdy dodatkowy obiekt SQLStatement wymaga dodatkowej pamięci. Ponadto jednoczesne wykonania również zwiększają ilość pracy, jaką wykonuje procesor, szczególnie na komputerach wyposażonych w jeden procesor lub procesor jednordzeniowy. Z powodu tych problemów takie podejście nie jest zalecane w przypadku urządzeń mobilnych.

Dodatkowym problemem jest to, że potencjalna korzyść z ponownego użycia obiektów SQLStatement może zostać utracona, ponieważ obiekt SQLStatement jest połączony z jednym obiektem SQLConnection. W konsekwencji obiekt SQLStatement nie może zostać wykorzystany ponownie, jeśli powiązany z nim obiekt SQLConnection jest już używany.

Jeśli wybrano korzystanie z wielu obiektów SQLConnection połączonych z jedną bazą danych, należy pamiętać o tym, że każdy z tych obiektów wykonuje jego instrukcje we własnej transakcji. Należy pamiętać, aby wziąć pod uwagę te oddzielne transakcje w każdym kodzie, który zmienia dane, np. przy dodawaniu, modyfikowaniu lub usuwaniu danych.

Paul Robertson przygotował bibliotekę kodu Open Source, który ułatwia wykorzystanie zalet stosowania wielu obiektów SQLConnection przy jednoczesnym ograniczeniu potencjalnych wad takiego podejścia. Biblioteka korzysta z puli obiektów SQLConnection i zarządza powiązanymi obiektami SQLStatement. W ten sposób zapewnia, że obiekty SQLStatement zostaną ponownie wykorzystane, oraz zapewnia dostęp do wielu obiektów SQLConnection, dzięki czemu możliwe jest jednoczesne wykonywanie wielu instrukcji. Więcej informacji na ten temat oraz opcje pobrania biblioteki można znaleźć na stronie http://probertson.com/projects/air-sqlite/.

Optymalizacja pliku bazy danych


Należy unikać zmian schematu bazy danych.

Jeśli to możliwe, należy unikać zmian schematu (struktur tabel) bazy danych po dodaniu danych do tabel bazy danych. Normalnie struktura pliku bazy danych zawiera definicje tabeli na początku pliku. Po otwarciu połączenia z bazą danych środowisko wykonawcze ładuje te definicje. Dane dodane do tabel bazy danych są dodawane do pliku za definicjami tabel. Jeśli jednak zostaną dokonane zmiany w schemacie, nowe dane definicji tabeli zostaną zmieszane z danymi tabeli w pliku bazy danych. Na przykład dodanie kolumny do tabeli lub dodanie nowej tabeli może spowodować pomieszanie typów danych. Jeśli nie wszystkie dane definicji tabeli znajdują się na początku pliku bazy danych, otwarcie połączenia z bazą danych zajmuje wówczas więcej czasu. Otwarcie połączenia jest wolniejsze, ponieważ wydłuża się czas potrzebny środowisku wykonawczemu na odczyt danych definicji tabeli z różnych części pliku.


Za pomocą metody `SQLConnection.compact()` można zoptymalizować bazę danych po zmianie schematu.

Jeśli konieczne jest dokonanie zmian schematu, po wykonaniu zmian można wywołać metodę `SQLConnection.compact()`. Ta metoda powoduje zmianę struktury pliku bazy danych, dzięki czemu dane definicji tabeli zostają umieszczone razem na początku pliku. Jednak operacja `compact()` może być wykonywana przez długi czas, szczególnie w przypadku dużych plików baz danych.

Niepotrzebne przetwarzanie bazy danych w czasie wykonywania programu


W instrukcjach SQL należy zawsze stosować pełną nazwę tabeli (łącznie z nazwą bazy danych).

W instrukcji należy zawsze określić jawnie nazwę bazy danych oraz nazwy poszczególnych tabel. (W przypadku głównej bazy danych należy stosować słowo „main”). Przykład: poniższy kod zawiera jawną nazwę bazy danych `main`:

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

Jawne określanie nazw baz danych powoduje, że środowisko wykonawcze nie musi sprawdzać każdej podłączonej bazy danych w celu odnalezienia określonej tabeli. Uniemożliwia również wybór błędnej bazy danych w środowisku wykonawczym. Należy przestrzegać tej reguły, nawet jeśli klasa SQLConnection jest połączona tylko z jedną bazą danych. W tle klasa SQLConnection jest również połączona z tymczasową bazą danych, która jest dostępna za pośrednictwem instrukcji SQL.


W instrukcjach SQL `INSERT` i `SELECT` należy stosować jawne nazwy kolumn.

Poniższe przykłady prezentują stosowanie jawnych nazw kolumn:

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

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

Należy porównać poprzednie przykłady z kolejnymi. Należy unikać takiego stylu kodowania:

```
-- 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
```

W przypadku braku jawnych nazw kolumn środowisko wykonawcze musi wykonywać dodatkowe operacje w celu określenia nazw kolumn. Jeśli instrukcja `SELECT` używa symbolu wieloznacznego zamiast jawnych kolumn, środowisko wykonawcze pobierze wówczas dodatkowe dane. Te dodatkowe dane wymagają przetworzenia i tworzą dodatkowe instancje obiektów, które nie są potrzebne.

Należy unikać łączenia tej samej tabeli wiele razy w instrukcji, chyba że konieczne jest porównanie tabeli z nią samą.

Ponieważ instrukcje SQL są coraz dłuższe, wielokrotne złączenie tabeli bazy danych w zapytaniu może nastąpić nieumyślnie. Często ten sam wynik można uzyskać, używając tabeli tylko raz. Wielokrotne złączenie tej samej tabeli może powstać, jeśli w zapytaniu jest używany co najmniej jeden widok. Na przykład można złączyć tabelę z zapytaniem oraz użyć widoku, który zawiera dane z tabeli. Obie operacje spowodują co najmniej jedno złączenie.

Składnia SQL zwiększająca efektywność wykonania kodu

Za pomocą instrukcji `JOIN` (w klauzuli `FROM`) można uwzględnić tabelę w zapytaniu zamiast w podrzędnym zapytaniu w klauzuli `WHERE`. Ta wskazówka ma zastosowanie, nawet jeśli dane tabeli są potrzebne do filtrowania, a nie do uzyskania zestawu wyników.

Połączenie wielu tabel w klauzuli `FROM` działa lepiej niż użycie podrzędnego zapytania w klauzuli `WHERE`.

Należy unikać instrukcji SQL, które nie korzystają z indeksów. Te instrukcje obejmują użycie funkcji agregacji w podzapytaniu, instrukcji `UNION` w podzapytaniu lub klauzuli `ORDER BY` z instrukcją `UNION`.

Indeks może znacznie zwiększyć szybkość przetwarzania zapytania `SELECT`. Jednak niektóre fragmenty składni SQL uniemożliwiają bazie danych korzystanie z indeksów i zmuszają ją do wykonywania operacji wyszukiwania i sortowania na rzeczywistych danych.

Należy unikać stosowania operatora `LIKE`, szczególnie z wiodącym znakiem wieloznacznym np.: `LIKE('%XXXX%')`.

Operacja `LIKE` obsługuje stosowanie wyszukiwań z wykorzystaniem znaków wieloznacznych, dlatego działa wolniej niż porównania z założeniem dokładnej zgodności. W szczególności, jeśli ciąg wyszukiwania rozpoczyna się symbolem wieloznacznym, baza danych nie będzie mogła w ogóle użyć indeksów w wyszukiwaniu. W zamian baza danych będzie musiała użyć wyszukiwania pełnotekstowego w każdym wierszu tabeli.

Należy unikać stosowania operatora `IN`. Jeśli możliwe wartości są z góry znane, wówczas operacja `IN` może zostać zapisana za pomocą operatorów `AND` lub `OR`, które zapewniają szybsze wykonanie.

Druga z poniższych dwóch instrukcji jest wykonywana szybciej. Dzieje się tak, ponieważ wykorzystuje ona proste wyrażenia równości połączone z operatorem `OR` zamiast instrukcji `IN()` lub `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
```

W celu zwiększenia wydajności należy rozważyć stosowanie alternatywnych form instrukcji SQL.

Poprzednie przykłady demonstrują, że sposób zapisania instrukcji również może wpłynąć na wydajność bazy danych. Często istnieje wiele sposobów pisania instrukcji SQL `SELECT` w celu pobrania określonego zestawu wyników. W niektórych przypadkach instrukcja zapisana w jeden określony sposób będzie działać znacznie szybciej niż zapisana inaczej. Dodatkowe informacje na temat instrukcji SQL i ich wydajności zawierają dedykowane materiały dotyczące języka SQL.

Wydajność instrukcji SQL

W celu wybrania szybciej działających instrukcji SQL należy je bezpośrednio porównywać.

Najlepszym sposobem porównania wydajności wielu wersji instrukcji SQL jest przetestowanie ich bezpośrednio z bazą danych i danymi.

Poniższe narzędzia programistyczne udostępniają czasy wykonania instrukcji SQL. Za pomocą tych narzędzi można porównać szybkość wykonania alternatywnych wersji instrukcji:

- Run! (narzędzia do tworzenia i testowania AIR SQL, autor: Paul Robertson)
- Lita (SQLite Administration Tool, autor: David Deraedt)

Rozdział 9: Testowanie i instalowanie

Testowanie

Istnieje pełna gama narzędzi dostępnych dla aplikacji przeznaczonych do testowania. Dostępne są klasy Stats i PerformanceTest opracowane przez społeczność użytkowników Flash. W przypadku aplikacji do testowania można także korzystać z programu profilującego Adobe® Flash® Builder™ i z narzędzia FlexPMD.

Klasa Stats

Klasa Stats — opracowana przez użytkownika mr. doob ze społeczności Flash — umożliwia utworzenie profilu kodu w środowisku wykonawczym za pomocą wersji wydania środowiska wykonawczego bez używania narzędzi zewnętrznych. Klasę Stats można pobrać ze strony https://github.com/mrdoob/Hi-ReS-Stats.

Klasa Stats umożliwia śledzenie następujących parametrów:

- Liczba klatek zrenderowanych na sekundę (im wyższa jest ta liczba, tym lepiej).
- Liczba milisekund wykorzystanych na renderowanie klatki (im niższa liczba, tym lepiej).
- Ilość pamięci, z jakiej korzysta kod. Jeśli ilość pamięci wzrasta przy każdej klatce, wówczas możliwe jest, że w aplikacji istnieje przeciek pamięci. Konieczne jest sprawdzenie każdego ewentualnego przecieku pamięci.
- Maksymalna ilość pamięci, z jakiej korzystała aplikacja.

Po pobraniu klasa Stats może być wykorzystywana z następującym kompaktowym kodem:

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

Obiekt Stats można włączyć, stosując kompilację warunkową w środowisku Adobe® Flash® Professional oraz w programie Flash Builder.

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

Przełączając wartość stałej `DEBUG` można włączać i wyłączać kompilowanie obiektu Stats. To samo podejście może być stosowane w celu zamiany dowolnego kodu, którego kompilacja nie jest wymagana w aplikacji.

Klasa PerformanceTest

Aby umożliwić tworzenie profilu wykonania kodu ActionScript, Grant Skinner przygotował narzędzie, które można zintegrować z przepływem pracy testowania modułów. Do klasy PerformanceTest można przekazać niestandardową klasę, a PerformanceTest przeprowadzi serię testów kodu. Klasa PerformanceTest umożliwia łatwe testowanie różnych rozwiązań. Klasę PerformanceTest można pobrać pod następującym adresem: http://www.gskinner.com/blog/archives/2009/04/as3_performance.html.

Program profilujący w programie Flash Builder

Program Flash Builder jest dostarczany z programem profilującym, który umożliwia testowanie kodu z wysokim poziomem szczegółowości.

Uwaga: *Dostęp do programu profilującego można uzyskać za pomocą programu Flash Player w wersji z debugerem — w przeciwnym wypadku pojawi się komunikat o błędzie.*

Program profilujący może być również używany z zawartością utworzoną w programie Adobe Flash Professional. W tym celu należy wczytać skompilowany plik SWF z projektu ActionScript lub Flex do programu Flash Builder, a następnie uruchomić program profilujący dla tego pliku. Informacje o programie profilującym zawiera sekcja Profilowanie aplikacji Flex w dokumencie Używanie pakietu Flash Builder 4.

FlexPMD

Dział Adobe Technical Services wydał narzędzie o nazwie FlexPMD, które umożliwia kontrolowanie jakości kodu ActionScript 3.0. FlexPMD to narzędzie ActionScript, które przypomina narzędzie JavaPMD. FlexPMD zwiększa jakość kodu, ponieważ kontroluje katalog źródłowy kodu ActionScript 3.0 lub Flex. Wykrywa mało wydajne fragmenty kodu, np. niewykorzystany kod, kod nadmiernie złożony, kod zbyt długi oraz niepoprawne użycie cyklu życia składników Flex.

FlexPMD jest projektem open source firmy Adobe dostępnym pod adresem http://opensource.adobe.com/wiki/display/flexpmd/FlexPMD. Jest również dostępna wtyczka do środowiska Eclipse, którą można pobrać pod adresem http://opensource.adobe.com/wiki/display/flexpmd/FlexPMD+Eclipse+plugin.

FlexPMD ułatwia kontrolowanie kodu i zapewnia, że jest on przejrzysty i zoptymalizowany. Jednak najważniejszą zaletą narzędzia FlexPMD jest jego rozszerzalność. Programiści mogą tworzyć niestandardowe zestawy reguł w celu kontrolowania dowolnego kodu. Na przykład: można utworzyć zestaw reguł, które będą wykrywać nadmierne stosowanie filtrów lub dowolne inne mało wydajne praktyki kodowania.

Instalowanie

Podczas eksportowania końcowej wersji aplikacji w programie Flash Builder należy się upewnić, że eksportowana jest wersja wydania. Eksportowanie wersji wydania usuwa informacje dotyczące debugowania zawarte w pliku SWF. Usunięcie informacji o debugowaniu powoduje zmniejszenie wielkości pliku SWF i umożliwia szybsze działanie aplikacji.

W celu eksportowania wersji wydania projektu należy użyć panelu Projekt w programie Flash Builder oraz opcji eksportowania kompilacji wydania.

Uwaga: *W przypadku kompilowania projektu w środowisku Flash Professional nie ma możliwości wyboru między wersją wydania i wersją zawierającą informacje o debugowaniu. Skompilowany plik SWF jest domyślnie wersją wydania.*