ВИКОРИСТАННЯ СУЧАСНИХ ТЕХНОЛОГІЙ ПО ОПТИМІЗАЦІЇ ВИХІДНОГО КОДУ ПРИ ПІДГОТОВЦІ ФАХІВЦІВ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

УДК 519.6

БОСКІН Олег Осипович
старший викладач кафедри інформаційних технологій Херсонського Національного Технічного Університету.
Наукові інтереси: комп'ютеризовані системи навчання.
e-mail: bbbosss@i.ua

ЧЕБАНЕНКО Олександр Володимирович
аспірант кафедри інформаційних технологій Херсонського Національного Технічного Університету.
Наукові інтереси: інтелектуальний аналіз даних.
e-mail: alexandrchebanenko@gmail.com

ВОРОБЙОВ Олександр Олегович
аспірант кафедри інформаційних технологій Херсонського Національного Технічного Університету.
Наукові інтереси: 3D-моделювання.
e-mail: Ileks93@mail.ru

ВСТУП

При розробці складних програмних проектів, пов'язаних з ігровою анімацією, динамікою, 3D, обробкою даних математичних моделей, виникає необхідність оптимізувати графіку і робочий код, який реалізує ту чи іншу функціональність. Хоч остання версія Flash-плеєра працює набагато швидше за попередні, не завжди це може допомогти при погано написаній програмі. Уповільнення роботи можуть викликати багато факторів: графіка, велика кількість об'єктів, складна анімація, велика частота кадрів тощо. Все це треба враховувати при підготовці фахівців розробки програмних проектів, так як це дуже важлива проблема для флеш-додатків, бо вони працюють в умовах обмежених ресурсів віртуальної машини, яка значно уповільнює роботу з системою.

МЕТА ДОСЛІДЖЕННЯ

Метою дослідження є використання сучасних технологій оптимізації коду, як особливо важливої частини флеш-розробки, при підготовці фахівців з інженерії програмного забезпечення. У флеш-технології як і у будь-якій іншій є ряд сильних сторін і недоліків. І завдання полягає якраз по-максимуму обрати найкращу, намагаючись обходити проблемні місця. Звичайно розробка оптимального, простого і логічно правильного проекту, завдання творче, але при цьому важливо дотримуватися певних рекомендацій і методів для специфіки даного середовища, щоб «творчість» не виявилася марною.

ОСНОВНА ЧАСТИНА

Представимо навчальний процес складною системою, що має безліч елементів (об'єктів) та відносин (зв'язків) між ними. У загальному вигляді складна система складається з множини викладачів $\{p_1, p_2, ..., p_{n1}\} \in P$, множини тих, що навчаються (студентів) $\{s_1, s_2, ..., s_{n2}\} \in S$, множини навчального матеріалу $\{k_1, k_2, ..., k_{n3}\} \in K$, множини методичного матеріалу $\{m_1, m_2, ..., m_{n4}\} \in M$.

Кожному викладачу характерні індивідуальні властивості які представлено у вигляді набору атрибутів

$p_i = \langle p_i^1, p_i^2, p_i^3, p_i^4, ..., p_i^j \rangle$, де p_i^1 - посада і- го викладача, p_i^2 - вчена ступінь, p_i^3 - вчене звання, p_i^4 - педагогічний стаж і-го викладача. Набір атрибутів викладача буде залежати від рівня абстракції і в залежності від типу відносин, де вони будуть розглянуті. Аналогічним чином представлено набір атрибутів студента $s_i = \langle s_i^1, s_i^2, s_i^3, s_i^4, ..., s_i^j \rangle$, де s_i^1 - спеціальність студента, s_i^2 - напрям підготовки, s_i^3 - курс і т.і. в залежності від рівня абстракції і типу відносин.

Всі відносини, що виникають в процесі науково-педагогічної діяльності у ВУЗі можуть бути представлені у вигляді відносин між множинами P,S,K,M. Та все ж основну увагу необхідно приділяти використанню передового досвіду сучасних розробників програмного забезпечення.

Необхідно при вивчені мов програмування та при розробці програмних проектів студентами спеціальності «Інженерія програмного забезпечення» показувати важливість використання сучасних технологій оптимізації програмного коду. У загальному випадку оптимізацію треба проводити комплексно як для графіки так і для програмного коду, так як ці речі часто між собою тісно пов'язані, наприклад: програмна анімація, інтерактивний графічний інтерфейс користувача і т.п. Головне завдання оптимізації флеш-додатків – максимально зменшувати розмір swf файлу і збільшувати швидкість роботи коду [5].

Робота з векторною графікою є сильною стороною флеш-технології, завдяки якій він власне і отримав широке поширення. Зберігання та робота з векторними кривими дає великі переваги для всіляких варіантів анімації, але важливо розуміти і слабкі сторони, які здатні вплинути на хорошу роботу.

Необхідно на практичних заняттях показати студентам, що графічний елемент, який використовується багато разів, краще оформляти у вигляді бібліотечного символу «Graphic». Це дає виграш за розміром кінцевого swf-файлу, зрозумілу структурованість, а також зручність модифікації – при редагуванні графічного символу в бібліотеці він автоматично буде змінений у всіх місцях куди був поміщений. Вельми оптимально конструювати нові графічні об'єкти, використовуючи певний набір символьних бібліотечних або ж по можливості якомога більше групувати елементи.

Звертається увага студентів на те, що якщо в сайті присутні графічні елементи зі складними межами, то бажано їх спрощувати (оптимізувати) наскільки це можливо. У векторному поданні будь-яка крива представляється як набір точок-координат і заданим рівнянням дуги кола між кожною парою точок. Отже, чим у нас більше задано точок, тим точніше можна описати криву. Але практично немає необхідності ускладнювати опис кривої, якщо в цьому немає особливої потреби, так як це забирає зайву пам'ять і швидкість відтворення. Отже, бажано мінімізувати такий опис кривої, наприклад за допомогою меню: Modify> Shape> Optimize (Ctrl + Alt + Shift + C). До аналогічних результатів також призводять операції згладжування «Smooth», «Straighten» [4].

Треба намагатися зменшувати кількість різноманітних типів ліній (пунктир, штрихова і т.д.). Сама базова і оптимальна по займаній пам'яті і відтворення є суцільна лінія, тому бажано застосовувати її якомога частіше. Слід також зазначити, що досить корисно використовувати замість «Brush Tool» – «Pencil Tool», так як перший малює не просту лінію а полігон, залитий всередині вибраним кольором.

Безумовно, одна з найсильніших сторін даної технології є векторне подання даних, яке дає великі можливості для графіки та анімації при порівняно невеликих обсягах, тому основний упір завжди треба робити на векторну графіку і по можливості намагатися обходитися без растра. Останній рекомендується використовувати тільки як фоновий або статичний. Але не завжди застосування векторної графіки може себе виправдовувати. Наприклад, відрисовка складного, з безліччю кривих і градієнтів, векторного фону або анімація векторного об'єкта може займати відчутно більше часу, ніж така ж растрова картинка. Тому іноді потрібно жертвувати розмірами кліпу для досягнення гарної продуктивності. При цьому бажано налаштовувати співвідношення якість-обсяг імпортованої растрової графіки, щоб якомога менший обсяг вона займала.

Простий текст займає набагато менше місця ніж таке ж його графічне представлення. Він зберігається у вигляді текстового рядка і досить швидко відрисову-

ється системою, тому доцільно без особливої потреби не перетворювати текст в графіку. При цьому також потрібно зменшувати кількість шрифтів і стилів, так як при додаванні хоча б однієї букви обраного шрифту, копіюється весь шрифт в кліп. Тому краще користуватися вбудованими шрифтами, так як це економить розмір файлу. Але якщо все ж стандартні не підходять треба намагатися вибрати шрифтове оформлення, яке має більш просте накреслення символів і вказати які саме символи (заголовні, рядкові, цифри тощо) будуть використані, останнє можливе лише для динамічного тексту.

Анімація, як і графіка, теж поставлена на математичний фундамент. Звідси анімаційні послідовності набагато оптимальніше оформляти у вигляді «Tween» (звичайно, якщо це можливо), ніж у вигляді послідовностей кей-фреймів. «Tween» просто містить формулу з перетворення об'єкта анімації на певному відрізку, що значно економить розмір. Плеєру також краще таку анімацію виконувати з «movie clip», ніж з «graphic». Анімаційні послідовності треба переносити на окремі шари від всіх інших нерухомих елементів, це не тільки додає зручність, але і економить ресурси, так як в процесі анімації будуть оброблятися і статичні об'єкти, тобто виконуватися зайва робота.

Якщо потрібно зробити кілька однакових графічних символів з різними кольорами, то краще створити один бібліотечний графічний символ і там де він розташований просто міняти йому забарвлення у властивості Color: Tint, наприклад, коли треба створити багато різнокольорових кульок [5].

Оптимізація коду дуже важливе завдання для розробки сайтів пов'язаних з обробкою даних і 3D-моделювання. Для написання продуктивного коду треба знати не тільки тонкощі мови в даній технології, а й раціонально писати алгоритми обробки даних, тобто позбуватися від зайвої роботи. Рішеннями для проблемних місць можуть бути різні підходи, застосування певних методів сортування і обчислень, що істотно може поліпшити код, особливо разом з хорошою його оптимізацією.

Найнадійніший спосіб перевірити код на продуктивність – озброїтися тестовим кодом (якщо додаток досить великий, то бажано написати юніт-тести для всіх критично важливих місць) в якому перевіряти ті частини, які нас цікавлять. Простим інструментом може послужити тест-код простого порівняння часу виконання:

```
function GetTimeInterval(f:Function, count:Number):Number
{
        var start:Number = getTimer();
        for (var i:Number = 0; <count; i++) f();
        return getTimer() - start;
}
```

де, f – функція містить тестований код, count – кількість повторень для більш точної оцінки.

Можна проводити аналіз і оптимізацію за допомогою байт коду. Даний метод більш гнучкий і ефективний, так як дозволяє бачити послідовність команд для віртуальної машини флеш. Але треба бути дуже обережним, тому що не завжди коротший байт код повинен обов'язково швидше працювати, потрібно враховувати яким чином він виконується, чи не виконується більш громіздка робота, адже сама віртуальна машина може бути оптимізована на виконання певних команд.

Короткі описи виконуваних дій теж дають переваги. Так ініціалізація декількох змінних одним значенням: x = y = z = 0, працює трохи швидше, ніж порядкова, де кожній змінній окремо це значення присвоюється.

Дана особливість відноситься і до виразів, як числових так і логічних. Наприклад, всі числові обчислення, занесені в одну формулу, будуть виконуватися швидше, ніж ці ж обчислення розбиті на декілька послідовних підобчислень. Особливо слід відзначити здатність компілятора обчислювати і зберігати константні значення виразів в байт-код, як для математичних операцій, так і для функцій з сталим параметром:

x = Math.cos(Math.PI/4+1/2) + 5*3/12.

Тут значення у змінній x буде обчислено та збережено в байт-код під час компіляції, а при виконанні коду буде використано це значення. Щоб допомогти компілятору розпізнати вирази зі сталими в складній формулі треба виділити їх дужками, інакше ці обчислення будуть виконуватися під час роботи. У наведеному нижче коді арифметичні операції в дужках будуть обчислені компілятором і збережені в байт-код, а при виконанні цього рядка в програмі будуть вставлені на свої місця, що значно прискорить обчислення:

y = ((10-5)/(123 - 85))*x + (23*6/5).

При використанні оператора умови, необхідно поміщати по-можливості всі логічні значення або вираження відразу:

```
if (a && b && c){...}
```

Так як це працює значно швидше, ніж опис всіх гілок:

```
if (a)
{
        if (b)
        {
                if (c){ ... }
        }
}
```

Зробити код більш компактним можуть допомогти і спеціальні оператори, наприклад ?:, який замінює дві гілки оператора if с привласненням.

Як і в багатьох мовах програмування в Action Script існує неявне перетворення типу, заданого значення до необхідного типу. Цим можна користуватися для зменшення зайвих операцій в коді. Найчастіший приклад такого використання – логічні прапори з булевими і числовими значеннями написання:

```
n_hide = 1;
b_hide = true;
b_move = false;
if (n_hide && b_hide && !b_move) {...}
```

краще ніж:

```
n_hide = 1;
b_hide = true;
b_move = false;
if ((n_hide == 1) && (b_hide == true) && (b_move == false)) {...}
```

В останньому випадку ми виконуємо зайву роботу в порівнянні. Такий підхід дає чимале поліпшення продуктивності в циклах зменшують ітератор до нуля, для цього достатньо умовою циклу написати саму змінну, при нульовому значенні умова поверне false і цикл завершиться:

```
for (var i = 10; i; --i){...}
while (—i){...}
```

Ніякий серйозний код не може обійтися без функцій, вони дають виграш у розмірі коду, щоб встановити час однієї і тієї ж дії в різних місцях, роблять код більш осмисленим і логічним. Але виклик функції займає додатковий час. Наприклад, ось такий код:

```
function Sum(x:Number, y:Number):Number
{
        return x + y;
}
x = 1;
y = 2;
z = Sum(x, y);
```

працює в 1.5-2 рази повільніше, ніж аналогічний:

```
x = 1;
y = 2;
z = x + y;
```

І це досить логічно, так як відбувається багато зайвих дій: виклик функції, передача параметрів, повернення значення - все в кінцевому підсумку позначається на продуктивності. Тому бажано обмежувати використання викликів функцій в місцях з підвищеною вимогою продуктивності або ж зовсім відмовлятися від них шляхом прямого вбудовування її коду.

Потрібно будувати код переважно на стандартні функції обробки даних, так як вони виконуються значно швидше визначених у програмі. Наприклад, функція Math.max, буде працювати вдвічі швидше, ніж будь-яка її аналогічна реалізація в коді або у вигляді окремої функції.

При використанні глобальних змінних як оголошених на глобальному об'єкті _global, так і на будь-якому іншому, слід пам'ятати що доступ до таких змінних уповільнений і постійне проживання у пам'яті під час програвання призводить до втрат продуктивності коду. Тому доцільно у випадках тривалого невикористання видаляти їх з пам'яті командою delete або ж у крайньому випадку встановлювати значення рівне null (забирає менше часу на зберігання і обробку).

Дуже важливим моментом в оптимізації коду є правильне використання змінних. Якщо час життя змінної обмежений виконанням функції, то її обов'язково потрібно оголошувати як локальну, використовуючи ключове слово «var». Доступ до локальних змінних набагато швидший і вони знищуються після виконання функції. Якщо ми пишемо оголошення змінної без ключового слова «var», то фактично оголошуємо нову змінну (глобальну, так як можемо до неї звернутися при звернені до об'єкту) для об'єкту кліпу на якому ця функція викликається. Тобто наступний код не оптимальний:

```
function a()
{
        x:Number = 10;
        y:Array = new Array();
        for (i=0; i<x; i++) y[i] = i;
}
```

призводить до створення зайвих змінних x, y, і на об'єкті де функція викликається (наприклад, _root), які будуть перебувати постійно в пам'яті, що може уповільнювати роботу даного коду в багато разів. Тому в даному випадку функція повинна мати вигляд:

```
function a()
{
        var x: Number = 10;
        var y:Array = new Array();
        for (var i:Number=0; i<x; i++) y[i] = i;
}
```

Для програмного управління анімацією створюються об'єкти муві-кліпи. Управління ними здійснюється через доступ до їх властивостей, викликом їх функцій. В останніх версіях macromedia рекомендується використовувати «dot» нотацію, для доступу до об'єктів через глобальний шлях. Це є найшвидшим способом роботи в порівнянні з усіма іншими. Глобальний шлях позбавляє плеєр проводити пошук зазначеного об'єкту у всіх областях видимості, тобто, якщо всередині даної галузі він нічого не знаходить по заданому імені він шукає вище. Тому для швидкодії бажано не опускати глобальний шлях, навіть якщо об'єкти лежать в одній області видимості і можна писати просто їх імена. Простий код:

```
my_mc._x = 10;
```

на 25% поступається за часом виконання найбільш оптимальному аналогу:

```
_root.my_mc._x = 10;
```

Ключовий момент оптимізації – цикли, так як найчастіше саме в них відбувається інтенсивна обробка великих обсягів даних. Головна рекомендація щодо їх оптимізації – прибрати всю зайву роботу і зробити код всередині циклу якомога компактнішим. Для цього підходять вище описані рекомендації, але є також ряд особливостей.

1) Змінні – всі оголошення локальних змінних використовуваних в циклі краще розміщувати перед циклом, це позбавить плеєр від необхідності кожного разу створювати і видаляти змінну з пам'яті при виконанні циклу, що значно прискорює роботу.

2) Кешування шляху – при частому використанні будь-якого глобального об'єкту або змінної, вельми не ефективно кожен раз звертатися до всіх його властивостей через абсолютний шлях в циклі. При цьому ми фактично ставимо кожен раз операцію «точка» для доступу до його властивості, що змушує плеєр проходити один і той же шлях [6]. Більш оптимальне рішення вказати прямий шлях до об'єкту в локальній змінній (кешуємо під час виконання) і винести її за межі циклу. Таким чином ми значно прискорюємо доступ до об'єкту. Наступний код:

```
        for (var i=0; i<10; i++)
{
        _root.obj.a = 1;
        _root.obj.b = 2;
        _root.obj.c = 3;
}
```

використовує глобальний об'єкт obj та його поля через повний шлях. Але це майже в два рази працює повільніше (чим більше операцій з об'єктом тим повільніше) ніж аналогічний код, де повний шлях ми зберігаємо локально:

```
var v_obj:Object = _root.obj;
for (var i=0; i<10; i++)
{
        v_obj.a = 1;
        v_obj.b = 2;
        v_obj.c = 3;
}
```

3) Обхід масиву – практично будь-яка обробка великої кількості даних зводиться до обробки масиву. Це реалізується звичайним циклом. Найпростіший цикл для обходу масиву «arr» виглядає наступним чином:

```
for (var i=0; i<arr.length; i++)
        arr[i] = i
```

Вище наведений код бажано ніколи не писати! По-перше, якщо масив «arr» не локальний, то на нього повинна бути описана локальна змінна, як описувалася вище. По-друге, вираз «i <arr.length» виконується при кожній ітерації, отже кожен раз відбувається зчитування властивості «arr.length». Тому, помістивши довжину

масиву в локальну змінну до циклу, ми отримуємо ще додатковий виграш за часом. Отримаємо, код, який працює в два рази швидше:

```
var varr = _root.arr;
var len = varr.length;
for (var i=0; i<len; i++)
        varr[i] = i;
```

ВИСНОВКИ

Обґрунтована необхідність, при підготовці розробників складних проектів, використання сучасних технологій оптимізації вихідного коду програми. Приведені приклади оптимізації вихідного коду, використовуючи досвід сучасних розробників програмного забезпечення, та зроблено порівняльний аналіз варіантів написання коду з оптимізацією та без оптимізації, які необхідно розглянути на заняттях студентів зі спеціальності «Інженерія програмного забезпечення».

ЛИТЕРАТУРА:

1. Devid Makfarland CSS: The Missing Manual - SPb: Piter, 2016 - 720 s.
2. Dzhennifer Robbins HTML5: Pocket Reference - M .: Vilyams, 2015 - 192 s.
3. Mykola Prokhorenok HTML, JavaScript, PHP i MySQL. Dzhentlmenskyy nabir Web-maystra // Volodymyr Dronov - SPb .: BKHV-Peterburh, 2015 - 766 s.
4. Ben Khenik Shlyakh do sovreshenstvu - SPb: Piter, 2011 - 336 s.
5. Stiv Makkonnell Doskonalyy kod - SPb: Piter, 2007 - 896 s.
6. Robin Nikson Stvoryuyemo dynamichni veb-sayty za dopomohoyu PHP, MySQL, JavaScript, CSS i HTML5 - SPb: Piter, 2015 - 688 s.
7. Kolisnychenko D. Profesiyne prohramuvannya na PHP - SPb: BKHV-Peterburh, 2007 - 211 s.
8. Den Siderkholm CSS3 dlya web-dyzayneriv - M .: Mann, Ivanov i Ferber, 2013 - 144 s.
9. Shmitt K. CSS. Retsepty prohramuvannya - SPb: BKHV-Peterburh, 2011 - 672 s.
10. Dzhennifer Niderst Robbins HTML5, CSS3 i JavaScript. Vycherpne kerivnytstvo - M .: Eksmo, 2014 - 528 s.

Рецензент: *д.т.н., проф. Коваленко В. Ф.*
Херсонський національний технічний університет