УДК 622.831.312: 004.42: 519.688

А. И. Слащев, студент

(НГУ)

ПОВЫШЕНИЕ ВЫЧИСЛИТЕЛЬНОЙ ЭФФЕКТИВНОСТИ АЛГОРИТМА ВИЗУАЛИЗАЦИИ РЕЗУЛЬТАТОВ РЕШЕНИЯ СЛОЖНЫХ ГЕОМЕХАНИЧЕСКИХ ЗАДАЧ

У статті запропонований новий метод підвищення обчислювальної ефективності програмних комплексів і алгоритми автоматизованого управління великими масивами даних для його реалізації.

INCREASING COMPUTATIONAL EFFICIENCY OF THE VISUALIZATION ALGORITHM FOR THE SOLUTION OF DIFFICULT GEOMECHANICAL TASKS

In this article we propose a new method of increasing the computational efficiency of software systems and algorithms automated control of large amounts of data for its implementation.

Для оценки устойчивости породного массива, зданий, подземных выработок и др. применяют методы математического моделирования, основанные на методе конечных элементов и методе начальных напряжений [1, 2]. Программа, реализующая данные методы, например, комплекс имитационного моделирования ИГТМ НАН Украины «ГЕО-РС», имеет интерактивную систему визуализации. Несмотря на то, что она построена на функциях прямого доступа библиотеки Direct X Graphics и напрямую работает с аппаратным обеспечением компьютера, приложение требует значительных ресурсов. Кроме того, в результате решения сложных геомеханических задач получают большие массивы данных, управление которыми очень трудоемко. К основным ресурсам, в данном случае, следует отнести объемы оперативной памяти, выделяемой в процессе выполнения программы под текстуры, сетки (Mesh), вершинные и индексные буферы, шейдеры и массивы рассчитанных данных. Ресурсы в больших приложениях часто дублируются, ненужные части не выгружаются, оперативной памяти не хватает, вследствие чего снижается скорость вычислений или на слабых машинах программа не работает. Если Вы столкнулись с трудностью повышения вычислительной эффективности большого приложения, то одним из способов выхода из данной ситуации является оптимизация управления ресурсами. В этой статье предлагаются апробированные нами методы повышения вычислительной эффективности и алгоритмы экономии ресурсов для больших программ и вычислительных комплексов.

Мы не будем рассматривать диспетчеризацию Windows, отвечающую за работу подсистем перераспределения ресурсов в памяти [3] или стандартный сборщик «мусора» в управляемом коде Java – *Garbage collector* (*GC*), а покажем, как самостоятельно управлять в программе хранением и использованием большого количества однотипной информации.

Почти любую программу условно можно представить четырьмя блоками

(рис.1). В серверных программах отсутствует первый блок, во многих программах для научных вычислений зачастую используют только второй блок, который почти отсутствует в интернет менеджерах закачек и файлообменниках. В основе любого сложного приложения должны присутствовать последние два блока, поскольку без них невозможно создать многофункциональную научную программу с высокой вычислительной эффективностью, современной и удобной визуализацией результатов расчетов.

Рис. 1 – Структурная схема программного приложения

Давайте обратим внимание на третий блок – менеджер управления размещением ресурсов в памяти. Под этим менеджером мы будем иметь в виду программный базовый класс *ResourcesManager* (*RM*), который централизованно управляет ресурсами программы, вследствие чего она может достичь максимально возможной производительности. Этот абстрактный тип данных следит за ресурсами памяти (ведет рейтинг данных), а по окончанию выполнения программы очищает оперативную память и корректно завершает все процессы управления памятью. При разработке программы «ГЕО-РС» мы использовали следующие основные принципы управления ресурсами при создании данного класса: 1 – повторяющиеся данные в памяти хранятся только в одном экземпляре; 2 – рейтингование ресурсов производится по частоте их использования. Покажем преимущества применения *RM*.

Как известно, чем чаще системе приходится копировать данные с винчестера, тем на него больше нагрузка и тем медленнее работает программа, которая будет тратить время на подкачку данных вместо выполнения расчетов. Поэтому, уменьшая обращения компьютера к ресурсам на винчестере, мы увеличиваем общую производительность системы. Если памяти достаточно для выполнения приложения, то неиспользуемые ресурсы освобождаются только в конце выполнения программы. При этом менеджер «знает» обо всех ресурсах и не допускает их дублирования, что повышает быстродействие приложения.

Если ресурс захватит слишком много оперативной памяти и ее не хватает, это приводит к интенсивной перекачке данных на винчестер и резкому падению производительности всей системы. В этом случае производится поиск и удаление неиспользуемых данных, а малоиспользуемые данные в течение некоторого времени сохраняются на винчестере или архивируются.

Перед первым и последним использованием каждого ресурса класс *RM* выполняет вызов функции, вследствие чего появляется возможность применения счетчика, который всегда покажет, сколько классов используют данный ресурс.

Перед использованием ресурса указатель (или ссылка на него) возвращается специальной функцией. Если ресурс недоступен, возвратится нулевой указатель. Это свидетельствует о том, что ресурс пока не закачан и надо его запросить через некоторое время. Поэтому менеджер легко может контролировать ресурсы и собирать статистику об их использовании.

Менеджер содержит функцию для быстрой загрузки ресурса. Преимущество такого подхода в том, что если основной поток не имеет возможности продолжать работу без определенного ресурса, то менеджер направляет загрузку ресурса на основной поток (чтобы функция вернула ненулевой указатель) и, при необходимости, приостанавливает выполнение других потоков (например, чтобы быстрее произошла загрузка с винчестера, другие потоки не будут обращаться к нему).

RM может обслуживаться отдельным потоком или несколькими потоками. Обслуживание *RM* отдельным потоком дает потоконезависимость и поддержку многоядерных систем, вследствие чего загрузка файла не будет тормозить всю программу и даст возможность делать какие либо действия. Если обслуживать *RM* несколькими потоками, то это дает дополнительную возможность загружать данные с нескольких источников одновременно.

Таким образом, предлагаемый менеджер ресурсов унифицирован и может быть использован для разных программ.

Дискуссия на тему «Что лучше, компилятор или интерпретатор?» не закончится никогда, поэтому выбор варианта реализации зависит от того, какие методы программирования использованы в основном приложении. Для большей наглядности рассмотрим их параллельно. Разберем два простых примера минимальной организации базового класса *ResourcesManager* (табл. 1).

Таблица 1 - Варианты реализации базового класса *ResourcesManager*

Вариант 1	Вариант 2
Компилятор, поддерживающий указатели, например, С++.	*Интерпретатор,* не поддерживающий указателей, который, в основном, работает со ссылками на классы и все классы наследуют класс Object. Например, Java и C#.
ResourcesManager private: **Map**<String, ResourceParametres*> **mRes**; public: void **ClassResAdd** (String name); void * **GetRes** (String name);	**ResourcesManager** private **Hashtable**<String, ResourceParametres> **mRes**; public void **ClassResAdd** (String name); public Object **GetRes** (String name); public void **ClassResDelete** (String name);
Освобождение памяти осуществляется в деструкторе класса *RM* после синхронизации	Память освобождается *GC* после обнуления ссылок во внешних классах и в *RM*

Классы *Map* или *Hashtable* обеспечивают работу ассоциативного контейнера *mRes*, в котором каждому ресурсу соответствует уникальный ключ – его имя. Если имя файла начинается с «http://», то, следовательно, ресурс в интернете, а если «C:\» то значит на винчестере. Также в имя можно шифровать свои места поиска (например, «!List5» – надо достать из списка пятый номер элемента), какие-либо другие параметры или даже использовать имя как массив байт и загрузить туда все, что угодно. Несмотря на то, что классы *Map* или *Hashtable* не всегда являются оптимальным выбором, поскольку выбор субъективен и зависит от реализации, в данном алгоритме эти классы неплохо себя зарекомендовали. Ассоциативный контейнер *mRes* хранит параметры ресурсов. Его основная задача состоит в обнаружении дубликатов ресурсов. Если запрашиваемый ресурс уже есть в оперативной памяти, то загрузка его копии не выполняется.

Указатель (вариант 1) может ссылаться на несуществующий ресурс, так как он может быть удален *RM***.** Это приводит к указанию на произвольные данные и ошибке памяти. Поэтому, для исключения ошибок надо делать синхронизацию, тогда этим указателем может пользоваться только один поток. В этом случае *RM* будет удалять ресурс, когда он не будет востребован программой. Если запрос на удаление ресурса производится ссылкой (вариант 2), то *RM* не может освободить память, пока на ресурс есть ссылка во внешнем классе. В этом случае надо во всех внешних классах после использования ресурса обнулять ссылку.

Метод *ClassResAdd* класса *RM* вызывается перед первым использованием ресурса во внешнем классе (рис. 2). При выполнении метода производится поиск ресурса. Если ресурс отсутствует, то он создается. После чего рассчитываются его параметры. При нехватке памяти, она освобождается.

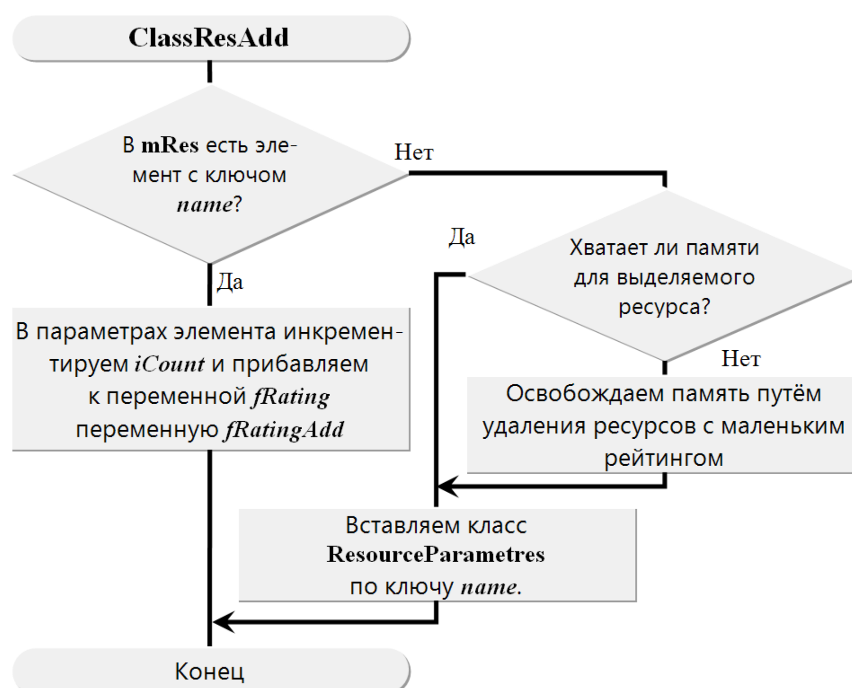

Рис. 2 – Блок-схема метода *ClassResAdd* класса *ResourcesManager*

Метод *GetRes* класса *RM* вызывается перед использованием ресурса во внешнем классе (рис. 3). Результатом выполнения данного метода является указатель или ссылка на ресурс, если он загружен в память. Метод рассчитывает рейтинг ресурса.

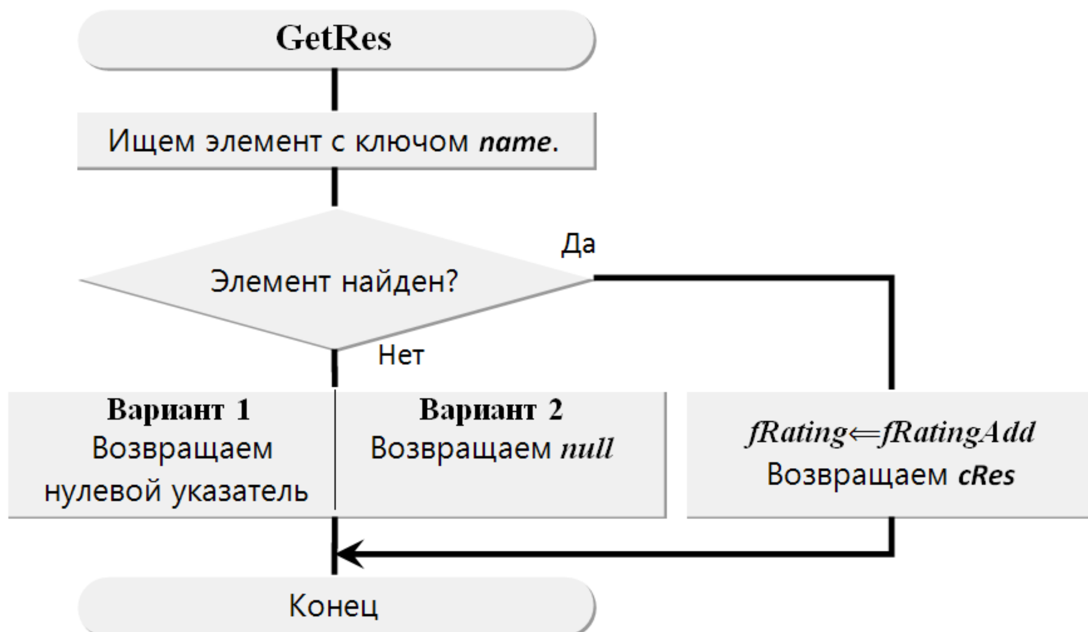

Рис. 3 – Блок-схема метода *GetRes* класса *ResourcesManager*

Метод *ClassResDelete* класса *RM* (рис. 4) вызывается перед удалением внешнего класса, использовавшего ресурс. При выполнении метода производится поиск ключа *name*, и если элемент найден, декрементируется счетчик *iCount*. В противном случае метод завершается. Такая ситуация происходит, если не хватало памяти, а у текущего ресурса был маленький рейтинг.

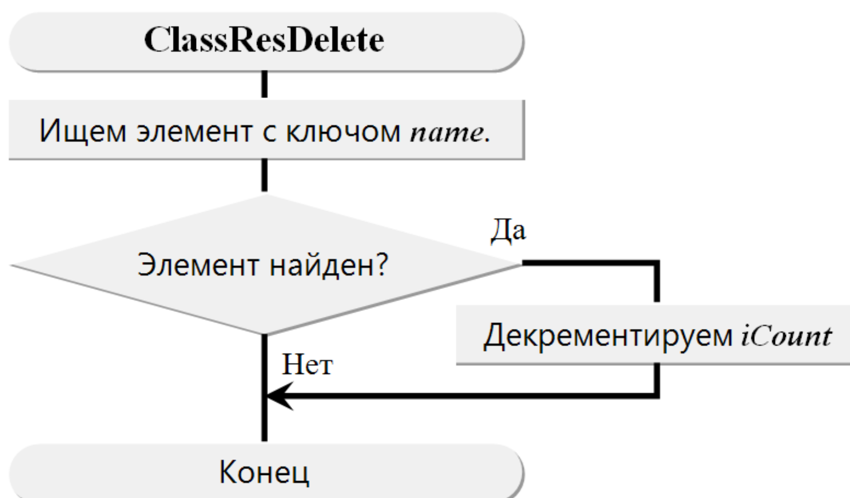

Рис. 4 – Блок-схема метода *ClassResDelete* класса *ResourcesManager*

Методы *ClassResAdd* и *ClassResDelete* можно не использовать, при этом

Вы потеряете возможность узнать, сколько классов используют ресурс. Но эта возможность не обязательна, все равно, в результате работы метода *GetRes* у неиспользуемых элементов рейтинг (*fRating)* будет стремиться к нулю.

ResourceParametres – класс хранения параметров ресурса. В этот класс можно включить много специфических параметров, которые будут впоследствии использоваться. Этими параметрами являются: время загрузки ресурса; время его последнего использования; переменные для расчета рейтинга; флаги, указывающие путь извлечения информации (это позволяет не анализировать имя ресурса) и др. Сейчас мы напишем пример такого класса с его основными параметрами (табл. 2).

Таблица 2 - Класс хранения параметров ресурса *ResourceParametres*

Вариант 1	Вариант 2
ResourceParametres ***ResourceParametres();*** int ***iCount***; float ***fRating***, ***fRatingAdd***; void ***cRes**; ……	**ResourceParametres** ***ResourceParametres();*** int ***iCount***; float ***fRating***, ***fRatingAdd***; Object **cRes**; ……
cRes – указатель, связывающий класс параметров ресурса с ресурсом	*cRes* – ссылка, связывающая класс параметров ресурса с ресурсом

В данном классе: *iCount* – количество классов, использующих ресурс; *fRating* – рейтинг ресурса, чем он выше, тем ресурс важнее; *fRatingAdd* – добавляемый рейтинг ресурса при его однократном использовании. Конструктор данного класса показан на рис. 5.

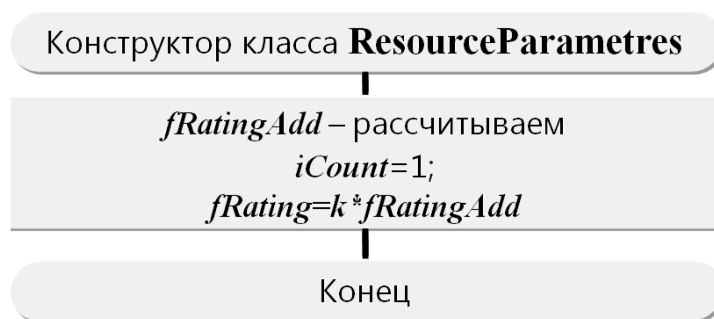

Рис. 5 – Блок-схема конструктора класса *ResourceParametres*

Коэффициент *k* (рис. 5) необходим для того, чтобы при первой загрузке ресурса его рейтинг не был занижен. К моменту загрузки нового ресурса, скорее всего, другие употребляемые ресурсы будут иметь значение *fRating* больше чем *fRatingAdd* и, чтобы новый ресурс не был сразу малорейтинговым, коэффициент *k* должен быть больше 1, например 2.

В *RM* применен статистический метод поиска малоиспользуемой информации. Рейтинг ресурсу назначается на основе частоты его использования.

Чем больше значение переменной *fRating,* тем ресурс более важен и больше используется.

Основные принципы установки рейтинга ресурсу:

1. Чем больше используется объект, тем он более важен. При каждом использовании объекта переменная *fRating* наращивается на величину *fRatingAdd.*

2. Чем дольше загружается объект, тем он важнее. Если изображение загружается с удаленного сервера, то его нужно хранить дольше чем при загрузке того же изображения с винчестера, и, чтобы этому ресурсу прибавить важность, следует умножить *fRatingAdd* на коэффициент, который показывает во сколько раз дольше идет загрузка с сервера, чем с винчестера. Или можно сохранить изображение на винчестер и назначить ему обычный приоритет.

3. Чем меньше объект, тем он важнее. Это связано с тем, что один большой объект загружается быстрее, чем много маленьких, которые в сумме дают такой же размер. Следовательно, если объект занимает мало физической памяти, то ему нужно увеличить значение *fRatingAdd*. Большие объекты легче перегружать в оперативную память, если они не используются много раз.

4. Чем больше стоимость загрузки ресурса, тем больше его рейтинг и, в результате, он может дольше храниться в памяти.

Формула вычисления *fRatingAdd* предлагается в виде:

$$fRatingAdd = k_p\left(2-\frac{\upsilon-\upsilon_{\min}}{\upsilon_{\max}-\upsilon_{\min}}\right)+(1-k_p)\left(2-\frac{V-V_{\min}}{V_{\max}-V_{\min}}\right), \quad (1)$$

где V, $V_{\min}$, $V_{\max}$ – текущий, минимальный и максимальный размеры ресурсов программы; υ, $\upsilon_{\min}$, $\upsilon_{\max}$ – текущая, минимальная и максимальная скорости загрузки ресурсов; k_p – коэффициент от нуля до единицы, характеризующий приоритет скорости загрузки ресурса относительно других рейтингов.

Понятно, если необходимые файлы загружаются из интернета и требуемый ресурс находится только на удаленном оплачиваемом сервере, то ему, безусловно, присваивается наибольший рейтинг. Переменная *fRatingAdd* имеет значение от одного до двух единиц и необходима для сравнительной характеристики ресурсов. Эта формула, в силу специфики той или иной задачи, может не подходить, однако сравнение относительных рейтингов ресурсов по ней выполняется хорошо.

Заметим, что переменная *fRating* растет при использовании ресурса, а когда он не используется, то *fRating* нужно уменьшать. Для этого необходимо через некоторое время пересчитывать массив параметров ресурсов и выполнять уменьшение значения рейтинга. Если Вы пишете графическую программу на локальном компьютере и в ней не более 2000 ресурсов, то этот пересчет можно осуществлять раз в прорисовку или через несколько прорисовок по формуле *fRating=0,99*fRating*, которая хорошо себя показала при расчетах.

Но если менеджер обрабатывает ресурсы на сервере, то рациональнее выполнять пересчет раз в несколько минут или более по формуле *fRating=0,6÷0,9*fRating*.

Таким образом, применение предложенного метода управления ресурсами программы имеет ряд существенных преимуществ и позволяет: проводить автоматическое наблюдение за динамической памятью (при переполнении находить данные с низким рейтингом и сохранять их на винчестере с последующим восстановлением); ускорить расчеты за счет группирования маленьких буферов и распределить вычисления на несколько процессов (равномерно нагрузить ядра процессора); повысить удобство доработки программы с использованием одного интерфейса для загрузки разных данных, не разрабатывать повторного кода загрузки и кэширования данных в разных местах программы.

В результате была повышена вычислительная эффективность программного комплекса геомеханического моделирования по надежности, скорости прорисовки графических примитивов, используемой памяти и процессорному времени. В целом, программа показала свою работоспособность и была успешно апробирована при определении напряженно-деформированного состояния и оценке устойчивости породного массива, элементов конструкций наземных и подземных сооружений.

СПИСОК ЛИТЕРАТУРЫ

1. Булат, А. Ф. Экспериментально-аналитический метод прогноза направлений и интенсивности газовых потоков [Текст] / А. Ф. Булат, С. А. Курносов, И. Н. Слащев [и др.] // Геотехническая механика. – Днепропетровск : ИГТМ НАНУ, 2005. – Вып. 59. – С. 10-21.

2. Слащев, И. Н. Моделирование трещиноватости как основа прогноза газового режима добычных участков глубоких шахт [Текст] / И. Н. Слащев, М.Ю. Иконников // Сб. науч. тр. НГУ № 31. – Днепропетровск: РИК НГУ, 2008.– С. 236-245.

3. Рихтер, Дж. Windows для профессионалов: создание эффективных Win32 приложений с учетом специфики 64-разрядной версии Windows [Текст] / Дж. Рихтер ; пер. с англ. - 4-е изд. - СПб; Питер; М.: Издательско-торговый дом "Русская Редакция", 2001. - 752 с.; ил.