Федеральное агентство по образованию
САНКТ-ПЕТЕРБУРГСКИЙ
ГОСУДАРСТВЕННЫЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

В.Г. Шубников
В.С. Беляев
С.Ю. Беляев

ИНФОРМАТИКА

Программирование на языке ассемблера

Учебное пособие

Санкт-Петербург
Издательство Политехнического университета
2007

Шубников В.Г., Беляев В.С, Беляев С.Ю. **Информатика. Программирование на языке ассемблера:** Учеб.пособие. СПб.: Изд-во Политехн. ун-та, 2007. 101 с.

Пособие соответствует государственному образовательному стандарту дисциплины “Информатика” направления бакалаврской подготовки 510200 “Прикладная математика”.

Описаны основы синтаксиса языка ассемблера и его макрорасширений. Рассмотрены алгоритмы растеризации отрезков и треугольников, предназначенные для практической реализации на лабораторном практикуме. Один из разделов посвящен различным алгоритмам деления целых чисел, применяемым в настоящее время разработчиками процессоров и используемым при разработке программ для портативных и мобильных компьютеров.

Представлены методы оптимизации разнообразных алгоритмов на языке ассемблера для различных архитектур процессоров. Приведены задания лабораторного практикума по курсу программирования на языке ассемблера.

Предназначено для студентов третьего курса физико-механического факультета и факультета технической кибернетики, может быть использовано при проведении лабораторных работ по курсу “ Информатика ”.

Табл. 7. Ил. 25. Библиогр.: 18 назв.

Печатается по решению редакционно-издательского совета Санкт-Петербургского государственного политехнического университета.

© Санкт-Петербургский государственный политехнический университет, 2007

ISBN

ОГЛАВЛЕНИЕ

Введение…….. 4
1. История и применение языка ассемблера .. … ..……. 5
2. Синтаксис языка ассемблера… … … … … .. ……. 7
3. Первые шаги программирования на языке ассемблера ….. ..…… 24
4. Растеризация отрезков и треугольников… …. ..…….. …… 42
5. Алгоритмы деления целых чисел .. … … …….. ….. . 59
6. Приемы оптимизации программ … …. … …….. …. 74
7. Лабораторный практикум… … … … . … … .. … .. .. .. ……...….80
Библиографический список ……..…… 100

ВВЕДЕНИЕ

Программирование на языке ассемблера и связанное с этим понимание устройства и функционирования процессоров – базовый фундамент знаний для инженера-математика, применяющего свои знания на современных вычислительных устройствах.

В пособии кратко описываются макрорасширения языка ассемблера, на простых примерах показано, как можно начинать практическую работу по программированию на языке ассемблера.

Программирование на языке ассемблера исторически связано с определенным классом задач и алгоритмов, в которых применение данного языка может быть эффективно, например в улучшении производительности программ.

Рассмотрены простейшие алгоритмы растеризации графических примитивов в качестве целевых алгоритмов, предназначенных для программирования на языке ассемблера. Понимание этих алгоритмов необходимо для выполнения практических задач.

Один из разделов посвящен классу алгоритмов деления, предложены различные улучшения целочисленного алгоритма деления.

Пособие – не только вводный экскурс в программирование на языке ассемблера, языке наиболее низкого уровня абстракций, но также и отличное средство улучшения практических навыков программирования на языках высокого уровня, таких как C++.

Пособие может быть использовано как вводный курс в программирование на языке ассемблера и как пособие по базовым алгоритмам компьютерной графики.

Многие особенности различных процессоров отражены не в полном объеме, при прочтении могут возникнуть дополнительные вопросы об особенностях команд и алгоритмов. Пособие не претендует на звание полного справочника по языку ассемблера или же по архитектуре процессоров. Многие материалы пособия составлены на основании различной технической документации производителей процессоров и могут содержать неточности.

В конце пособия приведены примеры практических работ и описания заданий по курсу программирования на языке ассемблера и архитектуры процессоров.

1. ИСТОРИЯ И ПРИМЕНЕНИЕ ЯЗЫКА АССЕМБЛЕРА

Известно, что компьютеры выполняют алгоритмы, представленные в виде последовательности машинных команд. В этих командах в двоичной форме закодирована информация об операциях процессора и адресах памяти, где находятся объекты этих операций. Такую запись алгоритма называют машинной программой или машинным кодом, а файл, ее содержащий – исполняемым, двоичным или бинарным файлом.

Ручное программирование на таком машинном языке – дело чрезвычайно трудоемкое: двоичная (как и восьмеричная или шестнадцатеричная) форма кодирования непривычна для человека, а постоянное отслеживание того, в каких адресах памяти находятся объекты, требует огромного внимания. Поэтому уже в первом десятилетии компьютерной эры, в 1950 г., появились первые языки программирования, частично автоматизирующие этот труд, – языки ассемблера, или автокоды. В них коды машинных команд заменяются их мнемоническими обозначениями, вместо адресов памяти могут записываться имена переменных и символьные метки, а числовые константы пишутся в десятичной форме. Преобразование программы с языка ассемблера в машинный код (т.е. компиляцию) выполняет сам компьютер с помощью программы-ассемблера, откуда и происходит название языка (assembler – “сборщик” на английском языке). По структуре и размеру ассемблерная программа близка к машинной: каждый из ее операторов преобразуется в одну машинную команду (это не относится к макросам, которые будут рассмотрены ниже).

Более высокую степень автоматизации программирования дают алгоритмические языки, первые из которых появились в 1960-е г. Они названы так, потому что служат для записи непосредственно алгоритмов и не зависят от системы команд, т.е. от архитектуры процессора конкретного

типа. Один оператор алгоритмического языка транслируется с помощью компилятора в среднем в десяток машинных команд. Трудоемкость программирования на них существенно ниже, чем при использовании языка ассемблера, и поэтому они называются языками высокого уровня (имеется в виду уровень автоматизации программирования). На рис.1.1 приведена классификация языков программирования, показывающая место языка ассемблера.

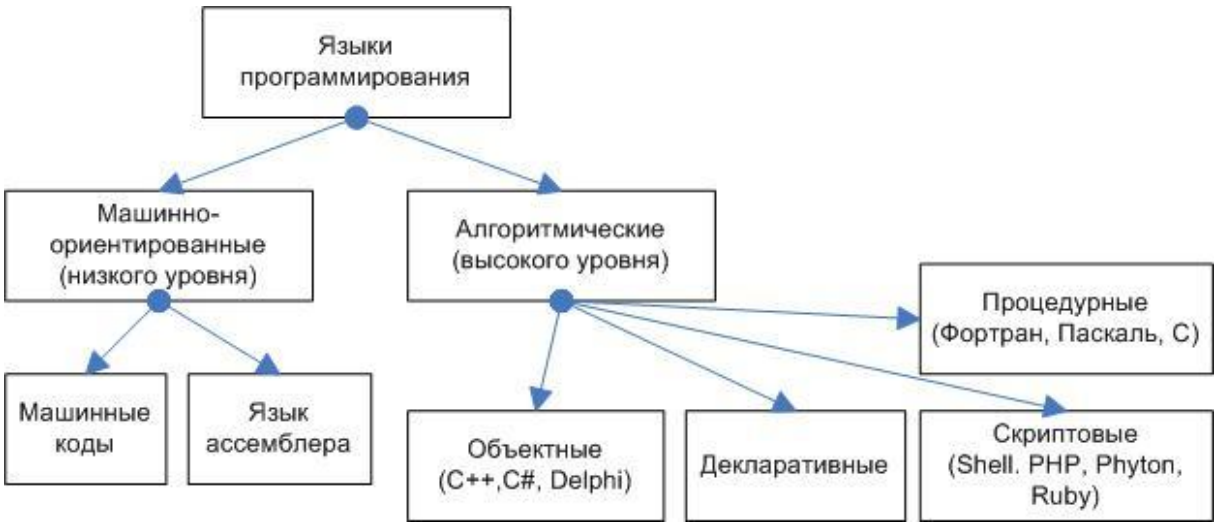

Рис. 1.1. Классификация языков программирования.

В наши дни именно алгоритмические языки служат основными инструментами программистов. К языку ассемблера им приходится прибегать только в тех случаях, когда остро стоит вопрос эффективности программы, прежде всего по производительности, т.е. когда необходимо написать максимально быструю программу (реже требуется эффективность по памяти). Обычно это требование относится к таким системным программам, как драйверы, а также к управляющим программам реального времени и графическим программам. Дело в том, что при программировании на языке ассемблера доступны все возможности архитектуры процессора, и программист может их использовать для реализации приемов ускорения программы. Языки же высокого уровня не дают такой возможности: генерацию машинных команд осуществляет компилятор, и управление этим процессом со стороны программиста ограничено.

Таким образом, для достижения высокой производительности программы приходится жертвовать производительностью труда ее разработчиков. Хорошим “компромиссом” между этими двумя показателями эффективности служит следующий подход: вначале программа пишется на языке высокого уровня, затем экспериментально выявляются ее “узкие места” – фрагменты кода, при выполнении которых происходят основные потери производительности (например, глубоко вложенные внутренние циклы). Далее эти фрагменты заменяются, переписываются на языке ассемблера; они могут составлять небольшую долю общего объема программы, но при этом существенно повышать ее производительность

К задачам, для решения которых возможно применение языка ассемблера, можно отнести:

- программирование драйверов устройств низкого уровня;
- оптимизация “виртуальной машины” для конкретного процессора;
- программирование систем шифрования и защиты программ;
- программирование, а также борьба с вирусами и другими вредными программами;
- программирование систем, работающих в масштабе реального времени;
- программирование технологических процессов и оборудования;
- оптимизация мультимедийных функций в современных программах.

Последняя группа задач – наиболее распространенная в современной практике использования языка ассемблера. На этой группе задач остановимся наиболее подробно, и рассмотрим многочисленные примеры.

На языке ассемблера можно написать столь же эффективную по размеру и времени выполнения программу, как и программу на языке машинных команд. Это преимущество отсутствует у языков высокого уровня.

2. СИНТАКСИС ЯЗЫКА АССЕМБЛЕРА

Свое название язык ассемблера получил от имени системной программы Ассемблер, которая преобразует исходные программы,

написанные на простом языке псевдокодов машинных операций, непосредственно в коды машинных команд. Термин “ассемблер” произошел от английского слова assembler – сборщик частей в одно целое. Частями здесь служат операторы, а результатом сборки – последовательность машинных команд. Процесс сборки называется ассемблированием. Обратите внимание на то, что результат работы компилятора ассемблера – именно машинные команды. Для современных систем программирования типа .NET термин “ассемблирование” применяется в несколько ином смысле – там происходит сборка программы (или ее частей) на так называемом “промежуточном коде”, т.е. собирается не последовательность машинных команд, а последовательность неких простейших операций, которая легко может быть преобразована в последовательность машинных команд.

Если для привычных языков программирования высокого уровня, таких как C/C++, Basic, Fortran, Pascal и других, используется единый синтаксис языка независимо от платформы, для которой разрабатывается программа, то для языка ассемблера картина иная. Для языков высокого уровня вполне могут подойти книги и учебные пособия, выпущенные десятки лет назад, для программирования на языке ассемблера нужно всегда “шагать в ногу со временем”, отслеживая последние новинки производителей процессоров и прилагаемых к ним всевозможных компиляторов, SDK (Software Development Kit – средство разработки приложений). В языках программирования высокого уровня операции для работы с памятью, структурами данных, указателями абстрагированы от конкретного процессора и операционной системы, а в языке ассемблера картина другая. Большая часть программы на языке ассемблера – это инструкции (команды) процессора. Так как все время появляются процессоры с новыми наборами команд, примеры программ на языке ассемблера тоже меняются кардинально. А при переходе на другую платформу разработчику программного обеспечения, возможно, придется “учиться” заново, так как он увидит совершенно новую для себя систему команд и, даже и синтаксис языка ассемблера.

Но не все так сложно и запутано в мире программирования процессоров. Кардинально новые подходы к вычислениям появляются

крайне редко, в основном новинки касаются улучшения хорошо известных, простых операций. Развитие архитектур процессоров в основном идет в сторону параллельного вычисления и воплощения в процессорах довольно сложных функций. Понимание тенденций развития микропроцессорной техники в мире позволяет более быстро ориентироваться в технических новинках и быстро переходить на программирование новой системы на самом низком уровне.

Основные операторы языка ассемблера для архитектуры процессоров фирмы Intel представлены в работах [1] - [8]. Также в Интернете, например сайт ru.wikipedia.org, www.friedspace.com/assembly/intro.php (в этом учебном пособии язык ассемблера описан лишь частично). По программированию для архитектуры ARM можно рекомендовать справочные материалы, размещенные на сайте www.arm.com. По программированию для архитектуры MMX рекомендуем заглянуть на http://www.codenet.ru/progr/optimize/mmx.php, где вы найдете описание инструкций MMX. Подробно о режимах адресации процессоров фирмы Intel можно прочитать на сайте:
http://www.intuit.ru/department/hardware/archhard2/6/. Здесь расположена хорошая статья по этой тематике :
http://dfe3300.karelia.ru/koi/posob/microcpu/arch4.htm.

Архитектура MMX появилась в первых процессорах фирмы Intel серии “Пентиум”. Дополнительный набор команд реализует SIMD архитектуру вычислений, что означает Single Instruction, Multiple Data (одной командой обработать несколько единиц данных). Архитектура ARM (Advanced RISC Machine) является основой всех бюджетных процессоров с низким энергопотреблением, используемых в портативных компьютерах и мобильных телефонах.

Рекомендуем сначала прочитать эти источники, а потом приступить к изучению данного пособия, в котором подробно рассмотрены макрорасширения языка ассемблера, помогающие программировать на более высоком уровне абстракций (эта информация редко встречается в публичных источниках).

Структура программы на языке ассемблера

Программа на ассемблере может состоять из следующих частей:

- блок включения заголовочных файлов;
- блок описания констант;
- блок описания переменных;
- блок функций.

Каждый из этих блоков может присутствовать в типичной программе на языке ассемблера. Блок функций, как правило, должен присутствовать всегда, за исключением такого экзотического случая, как использование исходного модуля на языке ассемблера ради описания данных программы. Каждая синтаксическая единица любого блока – выражение. Если весь проект состоит из исходного модуля (модулей) на языке ассемблера, то одна из функций должна быть main, или WinMain (в зависимости от того, как именуется точка входа в программу для конкретной операционной системы).

Блок включения заголовочных файлов содержит перечень включаемых в компиляцию заголовочных файлов, содержащих описания функций, видимых из других модулей. В языке ассемблера, как и в языке С, необходимы опережающие описания.

Блок описания констант может содержать константные выражения, которые используются компилятором только в процессе компиляции и не занимают места в сегменте данных программы.

Блок описания переменных содержит переменные программы, содержащиеся в сегменте данных программы.

Блок функций может содержать одну или более функцию, а также содержать точку входа в программу. Также блок функций может содержать просто участки программного кода, которые могут быть применены для формирования других участков кода с использованием техники самомодифицирующегося кода.

Формат выражений

Все выражения на языке ассемблера могут быть записаны в следующем условном виде:

```
[Метка:] [Оператор] [Операнды] [; Комментарий]
```

Метки в большинстве версий языка ассемблера записываются в виде слова и следующего за ним символа двоеточия. Однако в некоторых ассемблерах можно встретить и другие формы меток. Метка должна заканчиваться двоеточием. В качестве метки нельзя использовать имена регистров или мнемоники команд.

Как известно, использование меток при программировании засоряет исходный текст программы и ухудшает его читаемость. В современных версиях языка ассемблера введены специальные синтаксические конструкции, позволяющие избежать расстановки меток, однако расстановка меток, условных и безусловных переходов – это неотъемлемая часть классического программирования на языке ассемблера. Поэтому в некоторых современных версиях языка ассемблера введены специальные синтаксические конструкции для структурных разветвлений и циклов, позволяющие избежать расстановки меток.

Оператор – это название действия: мнемоника (имя) команды процессора (например, команда сложения к содержимому одного регистра содержимого другого регистра “add”) или название вспомогательных действий, необходимых компилятору языка ассемблера для правильного построения машинного кода (например, определение переменных, управление листингом, задание точки входа в функцию и т.д.). Во втором случае оператор называется директивой ассемблера.

Операнды – это необходимые аргументы команд процессора. Порядок операндов различен для ассемблеров на различных платформах. Общепринятой считается следующая последовательность:

```
Команда  [Назначение] [, Источник ] [, Цель ]
```

`Назначение` – это регистр или область памяти, объект программы, в который будет записан результат выполнения команды. `Источник` – это регистр или область памяти, откуда команда берет исходные данные. `Цель` – это дополнительные данные, необходимые команде.

Некоторым командам не нужны никакие операнды, например, команда установки флага направления STD в архитектуре процессоров Intel. Некоторые команды нуждаются в описании только назначения, например, команда INC [регистр / память] архитектуры Intel увеличивает на 1 содержимое регистра / ячейки памяти. В качестве цели может указываться регистр, ячейка памяти, а также непосредственный операнд.

Порядок перечисления операндов для команды тоже не всегда такой, как описано выше. Для некоторых архитектур после мнемоники команды следует написать источник, а затем назначение.

Комментарий в программе на языке ассемблера начинается с символа “; и идет до конца строки.

Константы

Для задания констант в языке ассемблера можно пользоваться различными системами исчисления. Если в языках высокого уровня это, как правило, десятичная и шестнадцатеричная системы исчисления, то для программ на языке ассемблера предоставляется куда более широкий спектр:

двоичные – последовательность цифр 0 или 1, заканчивающихся B или Y, например 10001111B, 101Y;

десятичные – последовательность цифр 0...9, которая может заканчиваться D или T, например 129D, 129, 129T;

шестнадцатеричные – 0...9, A...F, заканчивающиеся H, если первая цифра – одна из A...F, то число должно начинаться с 0, например 276H, 0A2FH;

восьмеричные – 0...7 заканчивающиеся на O или Q, например 37O;

вещественные – 0...9, ‘+', '-', '.', 'E', заканчивающиеся на R, например, 9.48E-3R;

строчные – набор символов в кавычках или апострофах, например “Text”, ‘text1’.

Совместимость с языками высокого уровня

Многие компиляторы ассемблера допускают синтаксическое расширение языка ассемблера ради совместимости с языками высокого уровня. Основная идея поддержки такой совместимости состоит в том, что большая часть программного текста программы создается на языке высокого уровня, а некоторые функции (особенно важные для производительности, криптозащищенности и т.д.) – на языке ассемблера. При этом важно обеспечить передачу параметров в функцию в правильном порядке, обеспечить контроль типов передаваемых параметров, управление стеком и т.д. Необходимо подчеркнуть, что не все современные компиляторы ассемблера позволяют пользоваться этим синтаксическим разрешением! Для платформ на базе процессоров Intel это, как правило, работает, для других платформ – зачастую нет. Читайте внимательно документацию к конкретной системе программирования и вашему компилятору языка ассемблера.

В табл. 2.1 приведены свойства языков высокого уровня, поддерживаемые компилятором ассемблера для обычных настольных компьютеров на базе архитектуры Intel.

Таблица 2.1. Свойства языков высокого уровня

Свойство	C	SYSCALL	STDCALL	BASIC	FORTRAN	PASCAL
Префикс “_” в именах функций	x		x			
Все имена большими буквами				x	x	x
Аргументы функций слева направо				x	x	x
Аргументы функций справа налево	x	x	x			
Вызывающая процедура восстанавливает стек	x		*			

Окончание таблицы 2.1. Свойства языков высокого уровня

Свойство	C	SYSCALL	STDCALL	BASIC	FORTRAN	PASCAL
BP сохраняется				x	x	x
Возможно переменное число аргументов функций	x	x	x			

* Тип STDCALL использует восстановление стека вызываемой процедуры, если используется переменное число аргументов, иначе вызываемая процедура должна восстановить стек.

Пример использования:

```
.model flat, stdcall
```

Модели памяти

Для задания типов адресации кода и данных по умолчанию в языке ассемблера можно указать тип модели памяти, используемой по умолчанию. В табл. 2.2 перечислены типы моделей памяти, которые можно использовать при программировании на языке ассемблера.

Таблица 2.2. Модели памяти

Модель	Адресация кода	Адресация данных	Размер сегмента	Смешиваются код и данные
Tiny	NEAR	NEAR	16-битный	Да
Small	NEAR	NEAR	16-битный	Нет
Compact	NEAR	FAR	16-битный	Нет
Medium	FAR	NEAR	16-битный	Нет
Large	FAR	FAR	16-битный	Нет
Huge	FAR	FAR	16-битный	Нет
Flat	NEAR	NEAR	32-битный	Да

В современных системах программирования используется в основном модель Flat, при этом сегментные регистры не используются, а размер данных и адреса по умолчанию считается 32 бита.

Директивы ассемблера

Для определения данных простых типов используются следующие синтаксические конструкции:

```
[name] [S]BYTE  initializer [,initializer] …
[name] [S]SWORD initializer [,initializer] …
[name] [S]DWORD initializer [,initializer] …
[name] FWORD    initializer [,initializer] …
[name] QWORD    initializer [,initializer] …
[name] TBYTE    initializer [,initializer] …
[name] REAL4    initializer [,initializer] …
[name] REAL8    initializer [,initializer] …
[name] REAL10   initializer [,initializer] …
```

Указание знаковости / беззнаковости имеет смысл только для представления данных в отладчиках.

Определение данных сложных типов

Для совместимости с языками высокого уровня в языке ассемблера предусмотрены следующие структурные типы данных.

Структуры

```
name STRUCT [alignment]
поля структуры
name ENDS
```

значением `alignment` может быть 1, 2 или 4; `name` – имя структуры.

Пример использования:

```
date STRUCT
     month    BYTE 0
     day  BYTE 0
year BYTE 0
sday BYTE "Wednesday"   ; максимально допустимый размер
date ENDS
xmas date <12,25,2000,"Friday">
```

Пример использования структуры в программе на языке ассемблера:

```
.CODE
mov xmas.year, 0
lea EBX, xmas
mov EAX, (date [EBX]).year
```

Объединения

```
name UNION [alignment]
  поля
name ENDS
```

Пример использования объединений в программе:

```
bwd UNION
     b    BYTE      0
     w    WORD      0
     d    DWORD         0
bwd ENDS
uvar bwd  { 0 }
```

Битовые маски

```
recordname RECORD field:width [,field:width] …
```

Определяет битовую структуру байт, слово или двойное слово по группам битов, например:

```
TimeRec  RECORD hours:5,minutes:6, seconds:5
; hhhhhmmm
; mmmssssss
today TimeRec { 19,02,31 }
mask fieldname
```

Определяет битовую маску из определенной ранее битовой структуры. Например:

```
xor AX, MASK minutes
width fieldname|recordname
```

Определяет длину в битах групп бит.

Определение синонима типа

```
name TYPEDEF type
name TYPEDEF PROTO prototypelist
```

`type` – любой из типов ассемблера (BYTE, WORD, …) или уже определенный директивой `typedef`, `prototypelist` – список параметров для процедуры.

Определение размера переменной

```
LENGTHOF var
SIZEOF var
```

Определяет размер переменной в байтах.

Управление компиляцией

Так как компилятор языка ассемблера превращает исходный текст программы в последовательность машинных команд, можно управлять различными настройками выравнивания. Например, для создания исполняемых модулей в формате COM, необходимо расположить точку входа в программу (первую инструкцию) на строго определенное смещение от начала всего исполняемого модуля. Такая же картина характерна для написания драйверов устройств (в том числе и для Windows XP). В языке ассемблера предусмотрены следующие синтаксические конструкции для управления компиляцией, или так называемые директивы ассемблеру:

`ALIGN number`

Выравнивание кода или данных после этой макрокоманды на границу, заданную `number`. Это число должно быть степенью 2.

`EVEN`

Это синоним ”ALIGN 2”

```
ORG expression
ORG $+expression
```

Задание текущего смещения в компилируемом сегменте (данные или код), `$` - текущее значение. Например:

```
Message DB “Hello, World”
MessageLength DB $ - Message
```

Процедуры и метки кода

```
FuncName PROC [distance] [langtype] [visibility]
[USES reglist][,parameter : type]…
     [LOCAL var:type[,var:type]…]
     операции
FuncName ENDP
```

`distance` = NEAR|FAR|NEAR16|NEAR32|FAR16|FAR32.

`langtype` = один из типов языков (см. табл. 2.1)

`visibility` = PRIVATE|PUBLIC|EXPORT по умолчанию PUBLIC.; EXPORT указывает компилятору разместить имя процедуры в список экспортируемых функций;

`reglist` – список регистров через пробел, которые будут помещены на стек перед телом процедуры и сняты перед выходом (для корректной работы программы при оптимизирующих компиляторах языков высокого уровня). Например:

```
WndProc  PROC FAR PASCAL,
hWnd:HWND,
```

```
nMsg:WORD,
wParam:SWORD,
lParam:SDWORD
         LOCAL hDC:HDC, ps:PAINTSTRUCT, point:POINT
         mov EAX, 0
         ret
WndProc  ENDP
```

`Label PROTO[distance][langtype] [,[parameter]:type]`

Объявление имени функции или переменной для дополнительного контроля типа. Часто используется в заголовочных файлах ассемблера. Например:

```
RasterCreate PROTO FAR C ,
                      nWidth: DWORD,
                      nHeight: DWORD
```

`invoke FuncName [,argument[,argument]…]`

Вызов функции с загрузкой аргументов на стек в соответствии с типом языка высокого уровня (см. табл. 2.1). Например:

```
invoke RasterCreate EAX, nH
```

Видимость и область действия

`PUBLIC [langtype] name[, [langtype] name ]…`

Это объявление делает имя видимым для других модулей программы (необходимо ставить, если имя будет использоваться в других модулях)

`EXTERNDEF [langtype]name:type[,[langtype]name:type] …`

Имена функций, переменных, структур, меток определены в других модулях. После этого объявления их можно будет использовать.

`INCLUDE filename`

Аналог C выражения `#include <FileName.h>` Расширения имен включаемых модулей обычно принято *.INC.

Полное описание сегментов

Некоторые операторы ассемблера не выполняют никаких действий для создания машинных команд непосредственно, а лишь меняют внутреннее состояние самого компилятора. Такие операторы называются псевдооператорами. Псевдооператоры `SEGMENT` и `ENDS` делят исходную

программу на сегменты. В программе может быть до пяти типов сегментов: данные, команды, дополнительный сегмент, стек, неинициализированные данные. На практике редко применяют полное описание сегментов, но это необходимо знать при анализе листингов дизассемблеров и ассемблерных листингов при компиляции программ на языках высокого уровня.

```
name SEGMENT [align][READONLY][combine][use]['class']
     выражения
name ENDS
```

`align` = BYTE | WORD | DWORD | PARA | PAGE;

`PARA` – выравнивание на 16, `PAGE` – на 256; по умолчанию `PARA`;

`combine` = PRIVATE | PUBLIC | STACK; определяет, как компоновщик соединяет сегменты;

`use` = USE16 | USE32 | FLAT; определяет тип размера сегмента (размер операндов по умолчанию);

`class` = CODE | DATA.

`ASSUME SegReg, SegmentName [, …]`

Описывает соответствие сегментного регистра и сегмента, определенного в программе. Можно определить несколько сегментов данных, компилятору ассемблера необходимо указать, какой из них использовать при адресации через DS (регистр сегмента данных по умолчанию).

`.DOSSEG`

Сортирует сегменты по соглашению, принятому в OS MS-DOS.

Упрощенное описание сегментов

`.MODEL memorymodel [,langtype] [,ostype] [,stackoption]`

Инициализирует программную модель памяти.

`memorymodel` = TINY | SMALL | COMPACT | MEDIUM | LARGE | HUGE | FLAT;

`ostype` = OS_DOS;

`stackoption` = NEARSTACK | FARSTACK.. При опции NEARSTACK SS==DS, при FARSTACK SS!=DS;

`.STARUP`
`.EXIT`

Эта пара директив генерирует стартовый код программы в соответствии с типом модели памяти, типом стека и код для возврата в операционную

систему после окончания работы программы. Если модель памяти TINY, то директива `.STARTUP` генерирует директиву `ORG 100h`.

`.CODE .STACK .DATA .DATA? .CONST`

Эти директивы описывают сегменты кода, стека, инициализированных данных, неинициализированных данных, констант.

Упрощенные директивы управления выполнением программы

```
.IF condition
     statements
[.ELSEIF condition
     elsestatements…]
[.ELSE
     elsestatements…]
.ENDIF
```

Используется для повышения читаемости условных блоков по аналогии с языками высокого уровня. Генерирует инструкции сравнения и условные переходы. Например, `.IF AX==0` заменяется на следующие инструкции:

```
or AX,AX
cmp AX,0
```

В качестве условия бывает удобно использовать специальные мнемоники для флагов:

```
CARRY?        CF
OVERFLOW?OF
PARITY?       PF
SIGN?         SF
ZERO?         ZF
```

Например:

```
sub EAX, 8
.IF SIGN?     ; jns
     dec EAX,8
     …
.ENDIF
.WHILE condition
     statements
.ENDW
.BREAK [.IF condition]
.CONTINUE [.IF condition]
```

Используется для “читаемого” описания повторяющихся блоков. Например:

```
.WHILE 1
```

```
    mov AH,07h
    int 21h  ; ждать нажатия на клавишу без вывода
    .BREAK .IF AL==13  ; выйти из цикла если ENTER
    .CONTINUE .IF (AL<'0') || (AL>'9')
      ; пропустить если не цифра
    mov DL,AL
    mov AH,02h
    int 21h  ;  вывести на экран введенный символ
.ENDW

.REPEAT
    statements
.UNTIL condition
.UNTILCXZ condition
     ; Тот же самый цикл только с постусловием окончания.
```

Макроопределения

Макроопределения (макросы) – это последовательность операторов на языке ассемблера, которые могут несколько раз появляться в программе. В отличие от функций, макросы при компиляции не образуют операторов вызова и возврата из процедуры (call, ret), они реализуются контекстной подменой. Макросы – очень мощное средство для написания эффективных кодов. Некоторые компиляторы/отладчики ассемблеров позволяют при отладке “заходить” внутрь тела макросов в отличие от высокоуровневых аналогов #define (для языка C). Существенное отличие макросов от процедур в том, что в коде тело процедуры встречается один раз, а тело макроса каждый раз полностью “разворачивается” компилятором, тем самым увеличивая размер кода.

Каждый макрос состоит из трех частей:

```
Заголовок: <MacroName> MACRO [<Parameter,…>]
Тело: Последовательность операторов ассемблера
(инструкций или псевдооператоров)
Концевик: ENDM
```

Пример макроса – напечатать строку в текстовом экране функциями MS-DOS:

```
.DATA
caMessage DB "Hello,world" , '$'
.CODE
PrintStringEDX MACRO
  push EAX
  mov  AH,9
  int  21h
```

```
  pop  EAX
ENDM
mov EDX, OFFSET caMessage
PrintStringEDX
```

В библиотеках, поставляемых вместе с ассемблером, есть расширенное определение этой операции.

```
LOCAL [список имен]
```

Это определение позволяет создать уникальные имена внутри макроопределения, не пересекающиеся с именами, определенными в теле программы. При каждой новой макроподстановке эти имена будут заменены на новые. Именами могут быть имена переменных, меток.

Оператор повторения REPT берет счетчик повторений из следующего за ним выражения, указанного в поле операнда. Пример: определить таблицу двойных слов заданной длины и проинициализировать ее элементы Table[i] = i:

```
Allocate MACRO Length
VALUE = 0
REPT Length
DD VALUE
VALUE = VALUE + 1
ENDM
ENDM
.DATA
nArray LABEL DWORD
Allocate 128
```

Псевдооператор IRP позволяет явно перечислить аргументы, которые будут подставлены вместо формального параметра, при каждом повторении. Например, определение сохранения в стеке всех регистров общего назначения:

```
PushComRegs MACRO
IRP RegName,<EAX,EBX,ECX,EDX>
push RegName
ENDM
ENDM
.CODE
PushComRegs
```

Такие макросы повышают читаемость кода. При макрорасширении PushComRegs превращается в последовательность:

```
push EAX
push EBX
push ECX
push EDX
```

IRPC похож на IRP, но его аргументами должны быть не числа, а строковые переменные. IFB и противоположный IFNB псевдооператоры определяют блок, макрорасширяемый только в том случае, если указанный параметр определен при подстановке макроса (не определен). В конце условного блока должен стоять ENDIF.

EXITM позволяет “выйти” из макроопределения. При подстановке будет определена метка и оператор `jmp`.

“&” позволяет осуществлять конкатенацию строк, символов.

Условная компиляция

Подобный языкам высокого уровня, существует механизм условной компиляции:

```
IF <Expression>, IFDEF <Name>, ELSE, ELSEIF,  ENDIF
```

Управление листингом

При компиляции текста на языке ассемблера можно заказать получение не только объектного кода, но и листинга (параметрами командной строки компилятора). В тексте программы можно использовать псевдооператоры, управляющие построением листинга:

```
TITLE/SUBTITLE Задание заголовка/подзаголовка
PAGE           Задание длины и ширины страницы
.LISTALL        Начало листинга всех выражений
.LISTMACRO      Начало листинга всех макросов
```

В различных компиляторах ассемблера эти псевдооператоры могут отличаться.

Задание процессора

Допустимы следующие задания (обычно в начале текста программы):

```
.186
…
.686
.686p
```

Если есть буква “p” на конце, то компилятор разрешит использование инструкций, специфичных для работы с защищенными режимами. Например, при определенном типе процессора .386 при компилировании строки rdtsc будет выдано сообщение об ошибке, при заданном типе .586 инструкция rdtsc будет легальной.

Прочие директивы

`ALIGN <Number>`

Выравнивает код или данные на заданную границу, вставляя либо nop (код), либо пустые псевдооператоры DB 0 (данные). Параметр выравнивания должен быть степенью двойки.

`EVEN`

Это синоним ALIGN 2.

`ORG <смещение>`

Устанавливает внутренний счетчик смещения в выходном объектном коде на заданное смещение. Часто используется ORG 100 для описания правильного заголовка COM файлов.

`ECHO <строка>`

Эта директива выводит строку в процессе компиляции.

3. ПЕРВЫЕ ШАГИ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ АССЕМБЛЕРА

Для наиболее полного понимания тонкостей программирования процессоров и разработки программ с использованием языка ассемблера необходимо попробовать на практике “улучшить” какую-нибудь несложную программу с использованием оптимизации на языке ассемблера.

Рассмотрим простейшую программу для операционной системы Windows, которая выводит на экран среднее арифметическое двух изображений. По технологии программирования можно воспользоваться одним из двух способов использования фрагментов на языке ассемблера:

- фрагменты кода на языке ассемблера внутри модуля на языке C/C++; используется встроенный компилятор ассемблера;
- внешний исходный текстовый модуль на языке ассемблера. При этом используется внешний компилятор языка ассемблера.

Обратите внимание, что не все среды разработки приложений на языках высокого уровня имеют встроенные компиляторы языка ассемблера. Широко распространенный Microsoft Visual Studio имеет встроенный

компилятор языка ассемблера, а начиная с версии 2003 г., имеет и внешний компилятор языка ассемблера. Сначала рассмотрим оптимизацию при помощи встроенного компилятора языка ассемблера, а затем – и с помощью внешнего компилятора.

Чтение картинок с диска

Пусть на диске имеется два файла с картинками (рис. 3.1., 3.2).

Рис 3.1 Файл test.bmp содержит вид города Гонконг

Рис 3.2. Файл brick.bmp содержит картинку кирпичной кладки

Пусть для простоты обе картинки (см. рис. 3.1 и 3.2) имеют размер 256×256 и заданы в цветовом формате 24 бита на пиксель, т.е. по 8 бит на каждую компоненту цвета – зеленый (R), синий (G), красный (B).

В программе объявим указатели на начало каждой картинки следующим образом:

```
char *bitmapBits1;
char *bitmapBits2;
```

Для общего случая размеры картинок и экрана, конечно, надо представлять переменными, но в данном руководстве для простоты будем упрощать задачу.

Пусть размеры наших картинок и экрана определены так:

```
#define  WINDOW_WIDTH  256
#define  WINDOW_HEIGHT 256
```

Функцию чтения картинки можно для простоты реализовать следующим образом:

```
char *readBitmap(char *fileName)
{
  BITMAPFILEHEADER    head;
  HFILE               file;
  char                *bits;

  file = _lopen(fileName , OF_READ); // open file
  if ( file == HFILE_ERROR )    return NULL;
  bits = (char*)GlobalAllocPtr(
               GPTR ,
               WINDOW_WIDTH*WINDOW_HEIGHT*3);
  if (!bits) return NULL;
  _lread( file , &head , sizeof(head) ); // read BMPheader
  if (        // check is this BMP file
        head.bfType != 19778 ||
        head.bfOffBits != 54
     )
  {
    _lclose(file);
    return NULL;
  }
  _llseek(file, head.bfOffBits , FILE_BEGIN );
  _lread( file, bits , WINDOW_WIDTH*WINDOW_HEIGHT*3 );
  _lclose(file);
  return bits;
}
```

На этапе инициализации программы эта функция будет вызываться для наших двух картинок так:

```
bitmapBits1 = readBitmap("test.bmp");
bitmapBits2 = readBitmap("brick.bmp");
```

На этапе завершения программы надо освободить память от картинок:

```
GlobalFreePtr(bitmapBits1);
GlobalFreePtr(bitmapBits2);
```

На этом подготовительную работу с картинками можно считать законченной. Обратите внимание на то, что в реальных программах требуется намного более “интеллектуальные” процедуры чтения картинок, учитывающие различные форматы файлов, масштабирование, различные пиксельные форматы и т.д. В целях концентрации внимания на функции формирования изображения все прочие задачи намеренно упрощены.

Буфер экрана

Для вывода на экран результирующего изображения потребуется буфер экрана – байтовый массив, содержащий пиксели экрана. Объявим указатель на буфер экрана и проинициализируем его на этапе старта программы:

```
char *screenBuffer;
screenBuffer = (char*)GlobalAllocPtr( GPTR,
WINDOW_WIDTH*WINDOW_HEIGHT*3);
```

Точно так же, как и для картинок, будем для простоты полагать, что экран имеет размерность 256×256. Для вывода экранного буфера в Windows будем пользоваться функцией `SetDIBitsToDevice`. Один из аргументов функции требует указатель на “описание” буфера. Это описание буфера проинициализируем один раз на этапе загрузки программы:

```
static    BITMAPINFO    bmpInfo;
bmpInfo.bmiHeader.biSize          =
sizeof(BITMAPINFOHEADER);
bmpInfo.bmiHeader.biWidth         = WINDOW_WIDTH;
bmpInfo.bmiHeader.biHeight        = WINDOW_HEIGHT;
bmpInfo.bmiHeader.biPlanes        = 1;
bmpInfo.bmiHeader.biBitCount      = 24;
bmpInfo.bmiHeader.biCompression   = BI_RGB;
bmpInfo.bmiHeader.biSizeImage     = WINDOW_WIDTH *
WINDOW_HEIGHT * 3;
```

На этапе завершения программы, естественно, необходимо освободить занимаемую буфером экрана память:

```
GlobalFreePtr(screenBuffer);
```

Сам вывод буфера на физический экран, в окно программы, осуществляется вызовом одной функции:

```
int onPaint(HWND hwnd)
{
  HDC hdc;
  hdc = GetDC(hwnd);
  SetDIBitsToDevice(
                      hdc,
                      0, 0,
                      WINDOW_WIDTH, WINDOW_HEIGHT,
                      0, 0,
                      0,                // first scan line
                      WINDOW_HEIGHT,    // num scan lines
                      screenBuffer,     // bits
```

```
                        &bmpInfo,
                        DIB_RGB_COLORS
                    );
  ReleaseDC(hwnd, hdc ) ;
}
```

Осталось только лишь заполнить правильно сам массив `screenBuffer` и – все!

Функция получения текущего такта процессора

Для измерения скорости работы функции формирования экранного буфера потребуется какая-либо функциональность, измеряющая время или что-то зависящее от времени, например, функция, возвращающая текущий шаг (или “тик”) процессора. Безусловно, производительность можно измерять и функциями времени, но дело в том, что на современных компьютерах тактовая частота настолько велика, что вся процедура формирования буфера экрана занимает несколько миллисекунд, а функция Windows `timeGetTime` возвращает текущее время в миллисекундах, т.е. разрешения этой функции не хватит для точного измерения производительности. Требуется нечто более “короткое”, чем миллисекунды реального времени. Для этой цели вполне подойдет внутренний счетчик процессора, показывающий номер текущего шага, или “такта” процессора. Текст функции представлен ниже:

```
int getTick()
{
  int tick;
  __asm rdtsc
  __asm mov  tick, EAX
  return tick;
}
```

Обратите внимание на то, что в данном фрагменте каждая инструкция на языке ассемблера начинается с ключевого слова `__asm`. Этим ключевым словом обозначаем фрагмент исходного текста, который требуется компилировать встроенным компилятором ассемблера.

Возможен и другой подход: последовательность инструкций на языке ассемблера обрамляется фигурными скобками:

```
__asm
{
  rdtsc
  mov  tick, EAX
}
```

В этом случае текст программы становится более читаемым, и более похож на классические тексты на языке ассемблера из учебников.

Функция смешивания картинок на языке С

Рассмотрим сначала процесс смешивания двух картинок в процедуре на языке высокого уровня – языке С. Это необходимо для понимания сути происходящего процесса:

```
extern char *screenBuffer;
extern char *bitmapBits1;
extern char *bitmapBits2;

long  buildScreen(void)
{
  int                   i;
  unsigned char  *dst;
  unsigned char  *src1;
  unsigned char  *src2;
  unsigned long  r1, g1, b1;
  unsigned long  r2, g2, b2;
  unsigned long  r, g, b;

  int     start, end;
  int ticks;
  start = getTick();

  dst = (unsigned char*)screenBuffer;
  src1 = bitmapBits1;  // первая картинка
  src2 = bitmapBits2;  // вторая картинка

  for (i = WINDOW_WIDTH * WINDOW_HEIGHT; i > 0; i--)
  {
    b1 = *src1; src1 ++; // прочитать RGB из первой
    g1 = *src1; src1 ++;
    r1 = *src1; src1 ++;
```

```
    b2 = *src2; src2 ++; // прочитать RGB из второй
    g2 = *src2; src2 ++;
    r2 = *src2; src2 ++;

    r = (r1 + r2) >> 1; // найти средние цвета для RGB
    g = (g1 + g2) >> 1;
    b = (b1 + b2) >> 1;

    dst[0] = (unsigned char)b; // записать RGB
    dst[1] = (unsigned char)g;
    dst[2] = (unsigned char)r;

    dst += 3;
  }

  end = getTick();
  ticks = end - start;
  return ticks;
}
```

Вся функция формирования изображения представляет собой цикл с длиной, равной количеству пикселей на экране. За один повтор цикла производятся следующие операции:

- чтение цветовых компонент r, g, b из картинки №1;
- чтение цветовых компонент r, g, b из картинки №2;
- смешивание цветовых компонент с простейшим осреднением;
- запись результирующего цвета в буфер экрана.

За каждый повтор цикла формируется один пиксель буфера экрана. В начале и конце этой функции используется функция получения текущего такта процессора для замера общего количества тактов, необходимого для формирования всего буфера экрана.

Очевидно, что “узкое место” производительности нашей функции сосредоточено в теле цикла формирования всех пикселей буфера экрана. Попробуем улучшить внутренний цикл, переписав фрагмент этой функции на языке ассемблера.

Функция смешивания картинок на языке ассемблера

Будем хранить указатели на текущие пиксели исходных картинок в регистрах ESI и EDI, а счетчик цикла – в регистре ECX:

```
  __asm
  {
    mov   ESI, src1 // картинка1 в ESI
    mov   EDI, src2 // картинка2 в EDI
    mov   ECX, (WINDOW_WIDTH * WINDOW_HEIGHT)
again:
    // обнулить результирующий цвет в регистре EBX
    xor   EBX, EBX
    // получить компоненту B
    mov   EAX, [ESI]
    mov   EDX, [EDI]
    and   EAX, 0xff
    and   EDX, 0xff
    add   EAX, EDX
    shr   EAX, 1
    or    EBX, EAX
    // получить компоненту G
    mov   EAX, [ESI]
    mov   EDX, [EDI]
    and   EAX, 0xff00
    and   EDX, 0xff00
    add   EAX, EDX
    shr   EAX, 1
    and   EAX, 0xff00
    or    EBX, EAX
    // получить компоненту R
    mov   EAX, [ESI]
    mov   EDX, [EDI]
    and   EAX, 0xff0000
    and   EDX, 0xff0000
    add   EAX, EDX
    shr   EAX, 1
    and   EAX, 0xff0000
    or    EBX, EAX
    // записать полученный цвет из EBX
    mov   EAX, dst
    mov   [EAX], EBX
    // увеличить указатели
    add   ESI, 3
    add   EDI, 3
    mov   EBX, dst
    add   EBX, 3
    mov   dst, EBX
     // перейти к следующей итерации цикла или завершить
    dec   ECX
```

```
    jnz   again
  }
```

Обратите внимание на то, что операция осреднения цветов

```
c = (c1 + c2) >> 1
```

выполняется в этой версии сложением с последующим сдвигом вправо (например, для красной компоненты цвета):

```
    mov   EAX, [ESI]
    mov   EDX, [EDI]
    and   EAX, 0xff0000
    and   EDX, 0xff0000
    add   EAX, EDX
    shr   EAX, 1
```

Однако после этого обязательно нужно выполнить отсечение незначимых битов:

```
    and   EAX, 0xff0000
```

Это происходит вследствие того, что сдвиг вправо на 1 бит имеет ту же сущность действия, что и операция деления на 2, за тем исключением, что в регистре EAX хранится красная компонента цвета в битах 16 – 23 и возможный единичный бит в позиции 16 после сдвига вправо на 1 бит переместится в позицию 15, а эта позиция принадлежит зеленой компоненте цвета.

Попробуем улучшить эту процедуру на ассемблере, выполняя операцию осреднения для всех трех цветовых компонент одновременно!

Оптимизированная функция смешивания картинок на языке ассемблера

Рассмотрим подробнее схему хранения битов для компонент цвета в 32-битном регистре (рис. 3.3).

31 24	23 16	15 8	7 0
x x x x x x x x	R R R R R R R R	G G G G G G G G	B B B B B B B B

Рис. 3.3 Хранение компонент цвета в 32 битах

Здесь “x” обозначает незначимые величины, которые не используются в вычислениях. Если сложить два таких числа, то очевидно, сложение по компонентам цвета не будет произведено корректно, так как значимые

биты переместятся в “чужие” цветовые компоненты. Чтобы этого избежать, попробуем искусственно “разрядить” битовое наполнение 32-битного регистра незначащими битами, свободные 8 бит у нас есть.

Добавим незначимую пару битов между каждой компонентой цвета. После этого 32-битный регистр, содержащий цвет будет выглядеть так, как показано на рис. 3.4.

31 24	23 16	15 8	7 0
x x x R R R R	R R R R x x G G	G G G G G G x x	B B B B B B B B

Рис. 3.4. Разрежение бит

Храня цвета из двух исходных картинок в таком представлении, можно безбоязненно их складывать! Перенос битов в “чужие” компоненты цвета происходить не будет. Вместо трех сложений получится одно. Но важно то, что по сравнению с исходной версией на ассемблере в нашей улучшенной версии будет значительно меньше операций чтения исходных цветов из памяти. Правда, за это придется “заплатить” дополнительными операциями по “разрежению” 32-битного регистра, в котором хранится цвет, и дальнейшими операциями по извлечению результирующих компонент цвета из такого сложного представления:

```
  dst = (unsigned char*)screenBuffer;
  src1 = bitmapBits1;
  src2 = bitmapBits2;
  i = (WINDOW_WIDTH * WINDOW_HEIGHT);
  __asm
  {
    mov   ESI, src1 // ESI указывает на картинку 1
    mov   ECX, src2 // ECX указывает на картинку 2
    mov   EDI, dst
again:
    movzx EBX, BYTE PTR [ESI] // прочитать байт B
    movzx EAX, BYTE PTR [ESI + 1] // прочитать байт G
    shl   EAX, 10
    or    EBX, EAX
    movzx EAX, BYTE PTR [ESI + 2] // прочитать байт R
    shl   EAX, 20
    or    EBX, EAX // теперь EBX содержит RGB разреженно
    movzx EDX, BYTE PTR [ECX] // прочитать байт B
    movzx EAX, BYTE PTR [ECX + 1] // прочитать байт G
    shl   EAX, 10
    or    EDX, EAX
    movzx EAX, BYTE PTR [ECX + 2] // прочитать байт R
```

```
  shl   EAX, 20
  or    EDX, EAX  // теперь EDX содержит RGB разреженно
  add   EBX, EDX  // сложить два цвета
  mov   EAX, EBX
  shr   EAX, 1    // найти половину
  mov   BYTE PTR [EDI], AL  // записать компоненту B
  mov   EAX, EBX
  shr   EAX, 11
  shr   EBX, 21
  mov   BYTE PTR [EDI + 1], AL // записать компоненту G
  mov   BYTE PTR [EDI + 2], BL // записать компоненту R
  // следующая итерация цикла
  add   ESI, 3
  add   ECX, 3
  add   EDI, 3
  mov   EAX, i
  dec   EAX
  mov   i, EAX
  jnz   again
}
```

Это исходный текст улучшенного цикла. Обратите внимание на то, как используются битовые сдвиги на 10 и 20 бит для получения цветов исходных картинок в “разреженном” формате внутри регистров EBX, EDX. После их сложения результат надо бы сдвинуть вправо на 1 бит (выполнить деление на 2), но этого можно не делать, просто помня, что результирующие компоненты цветов будут храниться, начиная с битов 1, 11, 21, а не с 0, 10, 20, как в исходных регистрах EBX, EDX перед осреднением.

Вставка в проект исходного модуля на языке ассемблера

Если требуется выделить функцию на ассемблере в отдельный модуль, можно воспользоваться средой разработки Visual Studio (начиная с версии 2003 в ней появилась поддержка исходных файлов на языке ассемблера), в проекты которой можно добавлять исходные модули на языке ассемблера так же, как и исходные модули на языке C/C++.

Рис. 3.5. Проект с модулем на языке ассемблера.

На рис. 3.5 показано как выглядит проект в Visual Studio с добавленным модулем на языке ассемблера. При добавлении среда разработки сама “догадается”, каким образом надо компилировать файл с расширением *.ASM. Если это по какой-то причине не произойдет, можно всегда самостоятельно настроить проект таким образом, чтобы файл *.ASM компилировался правильно. Среда разработки обычно добавляет следующую командную строку для файла *.ASM:

```
ml.exe /c  /nologo /Fo".\Debug\Xxx.obj" /W3 /Zi
/errorReport:prompt  /Ta
```

Здесь:

- `ml.exe` – выполнимый модуль компилятора языка ассемблера (должен находиться в папке BIN среды разработки MS Visual Studio);
- `/c` – ключ, показывающий, что надо только компилировать исходный текст в объектный модуль;
- `/nologo` – ключ, выключающий вывод информации о компиляторе;
- `/Fo` – ключ, задающий явно имя результирующего объектного файла;
- `/W3` – ключ, задающий уровень предупреждений №3;
- `/Zi` – ключ, добавляющий отладочную информацию в объектный модуль;
- `/Ta` – ключ, показывающий, что надо компилировать файл независимо от его расширения как файл, содержащий исходный текст на языке ассемблера.

Преимуществом добавления отдельного исходного модуля на языке ассемблера является то, что в нем можно использовать более широкий набор синтаксических конструкций, чем во встроенном ассемблере. Например, для нашей программы можно использовать во внутреннем цикле:

```
mov         loopCounter, (WINDOW_WIDTH * WINDOW_HEIGHT)
mov         ESI, bitmapBits1
mov         ECX, bitmapBits2
mov         EDI, screenBuffer
.REPEAT
  movzx EBX, BYTE PTR [ESI]
  movzx EAX, BYTE PTR [ESI + 1]
  ...
  dec   EAX
  mov   loopCounter, EAX
.UNTIL sign?
```

Здесь использована конструкция `.REPEAT … .UNTIL`, позволяющая избежать добавления в исходный текст на ассемблере метки и условного перехода на метку. Это, конечно, делает исходный текст программы более читаемым.

Добавим, что и без среды разработки Visual Studio можно компилировать программы, написанные на языке ассемблера. Для этого в командной строке (или при помощи BAT файла) нужно просто набрать:

```
ml /c /W3 /Zi /Ta xxx.asm
```

Результаты работы программы и сравнение производительности.

На рис. 3.6 показан результат работы программы.

Рис. 3.6. Результат работы функции осреднения изображений.

Обратите внимание, что визуально получается один и тот же результат для всех трех версий процедуры формирования буфера экрана.

Замеры длительности процедуры в тактах позволяют объективно оценить производительность нашей программы с использованием различных функций формирования буфера экрана. Также можно определить производительность работы программы в кадрах в секунду. Эта величина дает общую оценку производительности для конечного пользователя и является объективной характеристикой нашей программы, но общая производительность в кадрах в секунду, или так называемый FPS (Frames Per Second), зависит от множества факторов: скорости выполнения нашей процедуры формирования буфера экрана, частоты самого центрального процессора, производительности видеокарты, с помощью которой осуществляется вывод изображения на физический экран, производительности операционной системы Windows и многих других факторов.

Выполненные замеры производились на компьютере Intel Pentium 4, 2.6 GHz, 1 Gb RAM с видеокартой NVIDIA GForce4 Ti 4200, 128 M. В табл. 3.1 показаны измерения производительности нашей простейшей графической программы.

Таблица 3.1. Измерение производительности

Версия программы	Время, в тиках процессора	FPS, кадров/секунду
Исходная версия на C, режим debug	9 173 714	353
Простейшая версия на ASM, режим debug	5 675 668	468
Исходная версия на C, режим release	4 063 917	568
Улучшенная версия на ASM, режим debug	3 351 141	595

Подчеркнем важную деталь: числа, представленные в табл. 3.1 – осредненные за несколько сотен кадров. При работе программы эти числа не будут постоянными вследствие разнообразной загрузки кэша памяти, кэша кода, а также других динамически меняющихся свойств процесса выполнения команд процессором.

Может возникнуть справедливый вопрос: нужно ли заниматься оптимизацией таких программ, если компилятор языка высокого уровня C/C++ имеет настройку “строить максимально быстрый код”, Представленные данные (см. табл. 3.1) дают ответ на этот вопрос. Более того, задачу осреднения двух изображений можно улучшить еще, используя технику “раскрученного цикла”, а также используя технологию MMX.

Усложнение задачи.
Смешивание двух изображений с весовым коэффициентом.

Усложним первоначально поставленную задачу следующим образом. Будем искать не среднее арифметическое двух изображений, а вычислять результирующий цвет по формулам:

$$R = R_1 w + R_2(1-w); \quad (3.1)$$

$$G = G_1 w + G_2(1-w); \quad (3.2)$$

$$B = B_1 w + B_2(1-w), \quad (3.3)$$

Здесь, R, G, B обозначают цветовые компоненты результирующего пикселя, а R_1, G_1, B_1 – цветовые компоненты пикселя изображения №1, соответственно с индексом 2 –изображения №2; величина w – весовой коэффициент, принимающий вещественные значения в интервале [0…1].

Чтобы избавиться от вычислений с вещественными числами, будем считать, что параметр w может принимать значения в интервале [0...255], но после умножений и сложения для каждой компоненты результат будем делить на 256. Таким образом, формулы (3.1), (3.2), (3.3) перепишем следующим образом:

$$R = (R_1 w + R_2(255 - w)) / 256 \quad (3.4)$$

$$G = (G_1 w + G_2(255 - w)) / 256 \quad (3.5)$$

$$B = (B_1 w + B_2(255 - w)) / 256 \quad (3.6)$$

Деление на 256 заменяется логическим сдвигом вправо на 8 бит или арифметическим, так как значения компонент цвета существенно положительные. На языке высокого уровня цикл вывода смешанного изображения будет выглядеть так:

```
dst = (unsigned char*)screenBuffer;
src1 = bitmapBits1;
```

```
src2 = bitmapBits2;
for (i = WINDOW_WIDTH * WINDOW_HEIGHT; i > 0; i--)
{
  b1 = *src1; src1 ++;
  g1 = *src1; src1 ++;
  r1 = *src1; src1 ++;
  b2 = *src2; src2 ++;
  g2 = *src2; src2 ++;
  r2 = *src2; src2 ++;
  r = (r1 * weight + r2 * (255 - weight) ) >> 8;
  g = (g1 * weight + g2 * (255 - weight) ) >> 8;
  b = (b1 * weight + b2 * (255 - weight) ) >> 8;
  dst[0] = (unsigned char)b;
  dst[1] = (unsigned char)g;
  dst[2] = (unsigned char)r;
  dst += 3;
}
```

Очевидно, что “узкое место” производительности нашей программы спрятано в шести операциях умножения на каждый пиксель результирующего изображения.

Оптимизация алгоритма смешивания изображений с помощью технологии MMX

Воспользуемся аппаратной командой технологии MMX – параллельным умножением 16-битных слов. Используя эту команду процессора, можно вместо трех умножений на каждую компоненту цвета выполнить одну операцию умножения (сразу для четырех 16-битных слов, одно из которых не будем использовать). Каждый регистр MMX может хранить четыре целых числа длиной 16 бит, при этом арифметические операции будут совершаться над всеми четырьмя числами параллельно.

Так как значение веса инвариантно (не меняется) относительно тела цикла, то заполним регистр MM5 четырьмя словами, в каждом из которых будет храниться величина weight:

```
mov       EAX, weight
movd      MM5, EAX
punpcklwd MM5, MM5
punpcklwd MM5, MM5
```

Такую же операцию выполним перед началом цикла для величины (255-weight):

```
mov       EAX, weight
mov       EBX, 255
sub       EBX, EAX
movd      MM6, EBX
punpcklwd MM6, MM6
punpcklwd MM6, MM6
```

Еще нам потребуется какой-нибудь регистр MMX, содержащий нули во всех битах:

```
pxor      MM7, MM7
```

Усовершенствованный цикл с использованием технологии MMX будет выглядеть следующим образом:

```
  mov       ESI, src1
  mov       EBX, src2
  mov       EDI, dst
  mov       ECX, (WINDOW_WIDTH * WINDOW_HEIGHT)
again:
  movd      MM0, [ESI]
  movd      MM1, [EBX]
  punpcklbw MM0, MM7    // 0 x 0 r 0 g 0 b
  punpcklbw MM1, MM7
  pmullw    MM0, MM5
  pmullw    MM1, MM6
  paddusw   MM0, MM1
  psrlw     MM0, 8
  packuswb  MM0, MM7
  movd      EAX, MM0
  mov       WORD PTR [EDI],AX
  shr       EAX, 16
  mov       BYTE PTR [EDI+2],AL
  // следующая итерация
  add   ESI, 3
  add   EBX, 3
  add   EDI, 3
  dec   ECX
  jnz   again
  emms
```

Основная идея обработки каждого пикселя состоит в загрузке 4 байт подряд в регистр MMX (при этом нас интересуют только 3 младших байта – это компоненты цвета). Регистры MM0 и MM1 получат 4 байта из первого и второго изображения соответственно. Далее командой punpcklbw можно последовательность байтов превратить в последовательность 16-битных слов. Для этого необходим обнуленный регистр MM7

x	x	x	x	x	R	G	B

x	R	G	B

Рис. 3.7. Содержимое регистра MMX.

На рис.3.7 показано содержимое регистра MMX до выполнения операции распаковки и после.

После этого выполняется параллельное умножение командой pmullw, сразу четыре 16-битных значения умножаются, и в результирующие части регистра-приемника записывается младшая половина результата умножения. Затем цветовые компоненты, представленные в регистрах MMX в виде 16-битных слов, складываются, и над ними производится операция логического сдвига вправо на 8 бит. Далее надо “упаковать” цветовые компоненты в байтовое представление, это выполняется с помощью команды packuswb. На этом этапе 3 младших байта регистра результата содержат цвет, в который надо покрасить текущий пиксель на экране.

После завершения цикла обязательно выполняется команда emms, необходимая для корректного выполнения сопроцессором команд для работы с вещественными числами (если они встречаются еще где-то в нашей программе).

Сравнение результатов работы программы

Проведя оптимизацию на языке ассемблера с использованием технологии MMX, получают результаты производительности программы и функции смешивания, продемонстрированные в табл. 3.2

Таблица 3.2. Результаты производительности

Версия программы	Время, в тиках процессора	FPS, кадров в секунду
Исходная версия на C, режим debug	6 774 525	342
Исходная версия на C, режим release	4 651 317	528
Улучшение. Версия на MMX ассемблере	3 702 033	572

Видно, что, используя язык ассемблера для кодирования внутренних циклов, можно добиться большей производительности и улучшить программу.

4. РАСТЕРИЗАЦИЯ ОТРЕЗКОВ И ТРЕУГОЛЬНИКОВ

Многие задачи в современных графических программах связаны с растеризацией геометрических фигур – отрезков, треугольников, сфер и т.д. Рассмотрим алгоритмы растеризации отрезков.

Классический алгоритм Брезенхема

Заданы две точки с координатами (x_0, y_0) и (x_n, y_n) на плоскости, которые надо соединить линией. На рис. 4.1 показан отрезок, который необходимо нарисовать:

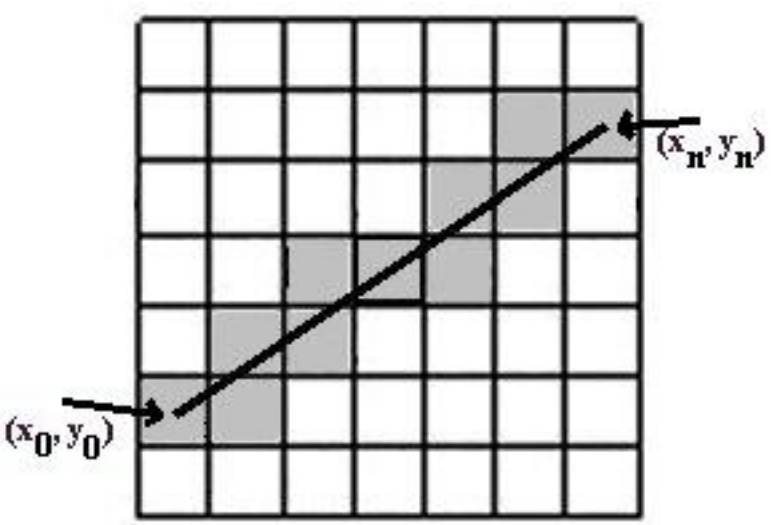

Рис. 4.1 Рисование отрезка алгоритмом Брезенхема

Решение этой задачи на псевдокоде выглядит следующим образом:

```
x = x0; y = y0;
Δx = |xn - x0|
Δy = |yn - y0|
p = 2*Δy - Δx;
for (x = x0; x <= xn; x++)
{
  if (p < 0)
```

```
    p = p + 2*Δy;
  else
  {
    y = y + 1;
    p = p + 2*Δy - 2*Δx;
  }
  SetPixel(x,y);
}
```

Подразумевается, что `SetPixel` – некоторая внешняя функция, выводящая точку (пиксель) в заданные координаты экрана. Классический алгоритм Брезенхема имеет существенный недостаток: для каждого выводимого пикселя выполняется разветвление, которое может приводить к задержкам конвейерной обработки команд в процессоре. Этот недостаток преодолевается с помощью условных присваиваний (movsx) на архитектуре Intel, либо условных префиксов команд на архитектуре ARM. В чистом виде классический алгоритм Брезенхема крайне редко используется на практике. Вы можете ознакомиться подробнее с алгоритмами графики на сайте http://algolist.manual.ru/graphics/painting/line.php.

Усовершенствованный алгоритм Брезенхема.

В литературе можно встретить термин DDA (Digital Difference Algorithm). Пусть отрезок задан координатами концов (x_1, y_1), (x_2, y_2). Координаты вещественные. Растровыми (целочисленными) координатами концевых точек становятся округленные значения исходных координат:

```
xstart = round(x1);
ystart = round(y1);
xend   = round(x2);
yend   = round(y2);
```

Количество шагов итерации выбирается как максимум из разностей координат:

L = max ((xend-xstart), (yend-ystart)) + 1;

В начале цикла вещественным переменным x, y присваиваются исходные координаты $x = x_1$; $y = y_1$. На каждом шаге цикла эти координаты

получают приращение `(xend-xstart)/L`, `(yend-ystart)/L`. Растровые координаты, выдаваемые на каждом шаге, являются результатом округления вещественных значений *x и y*. Псевдокод алгоритма DDA:

```
void dda_line (float x1, float y1, float x2, float y2)
{
 int i, L, xstart, ystart, xend, yend;
 float dX, dY, x[1000], y[1000];
 xstart = roundf(x1);
 ystart = roundf(y1);
 xend = roundf(x2);
 yend = roundf(y2);
 L = max(abs(xend-xstart), abs(yend-ystart));
 dX = (x2-x1) / L;
 dY = (y2-y1) / L;
 i = 0;
 x[i] = x1;   // запоминаем точку в массиве отрезка
 y[i] = y1;
 i++;
 while (i < L)
 {
  x[i] = x[i-1] + dX;
  y[i] = y[i-1] + dY;
  i++;
 }
 x[i] = x2;
 y[i] = y2;
 // вывод точек из массива отрезка
 i = 0;
 while (i <= L)
 {
  SetPixel(roundf(x[i]), roundf(y[i]));
  i++;
 }
}
```

Существенный недостаток алгоритма DDA (в его оригинальном виде) – применение вычислений с вещественными числами, в частности операция округления на каждой итерации. Также серьезным недостатком алгоритма является то, что некоторые точки линии будут выводиться дважды, что приводит к неэффективному использованию памяти и, как следствие, к снижению производительности алгоритма.

Несимметричный алгоритм DDA.

Несимметричный алгоритм DDA выполняет линейную интерполяцию функции $y = y0 + Ax$, где A – тангенс угол наклона.

$y_{k+1} = y_0 + A(k+1) = y_0 + Ak + A = y_k + A;$

Таким образом, для получения следующего значения y (шаг цикла) требуется всего одно сложение. В таком виде этот алгоритм вряд ли пригоден для практической реализации, так как величина A – вещественная, а значения y на каждом шаге необходимо получать целые, поскольку адресация точки в буфере экрана осуществляется в целых числах (адреса – целые). В этом виде на каждом шаге придется делать преобразование вещественного числа в целое, а это потеря производительности. Если же итерационную формулу для вычисления следующего y переписать в целых числах, то получим очень эффективно вычисляемую формулу. Оптимизация достигается за счет введения величины

$D = (int)(A * 2^R),$

где R – количество битов точности при замене вещественного параметра A целым числом.

Каждый шаг алгоритма состоит в вычислении следующего значения функции $y' = y + A$. При переходе к целым числам это сводится к целочисленному сложению и сдвигу вправо на R бит (отбрасывание битов точности). Выбор числа R зависит от диапазона $x_0 ... x_n$, а также от размера регистра, в котором будет содержаться это значение:

```
D = ( ( yn - y0 ) << R ) / ( xn - x0 );
yF = y0 << R + (int)( 0.5 * (1<<R) );
for ( x =x0 ; x <= xn ; x ++ )
{
  y = yF >> R;
  yF += D;
  // обработать полученное значение y
  SetPixel(x,y);
}
```

Несимметричный алгоритм DDA применяется на практике как в вещественном “исполнении”, так и в целочисленном. В зависимости от процессора, на котором будет выполняться программа, надо выбрать, в

каком варианте лучше программировать растеризацию линии. Это зависит от того, насколько эффективно выполняются процессором арифметические инструкции над вещественными числами и насколько быстрое преобразование из вещественного числа в целое. На архитектуре ARM однозначно надо выбирать целочисленную реализацию. На процессорах Intel Pentium последних поколений можно выбрать реализацию на вещественных числах.

Растеризация треугольников

Современные вычислительные системы обеспечивают растеризацию треугольников на аппаратном уровне. Растеризация треугольников – это базовая часть функциональности любой графической оболочки или SDK. Превосходные примеры, снабженные хорошими описаниями – библиотеки DirectX и OpenGL.

Современное программирование графики и мультимедиа для настольных компьютеров сводится к эффективному использованию той или иной графической библиотеки, выполняющей растеризацию треугольников на аппаратном уровне. Однако большинство “бюджетных” вычислительных устройств, портативных и мобильных компьютеров не имеют таких аппаратных возможностей, поэтому задача растеризации треугольников программными средствами актуальна и сейчас. Более того, эта задача в ближайшие несколько лет не потеряет своей актуальности вследствие взрывной распространенности мобильных устройств, построенных на базе архитектуры ARM.

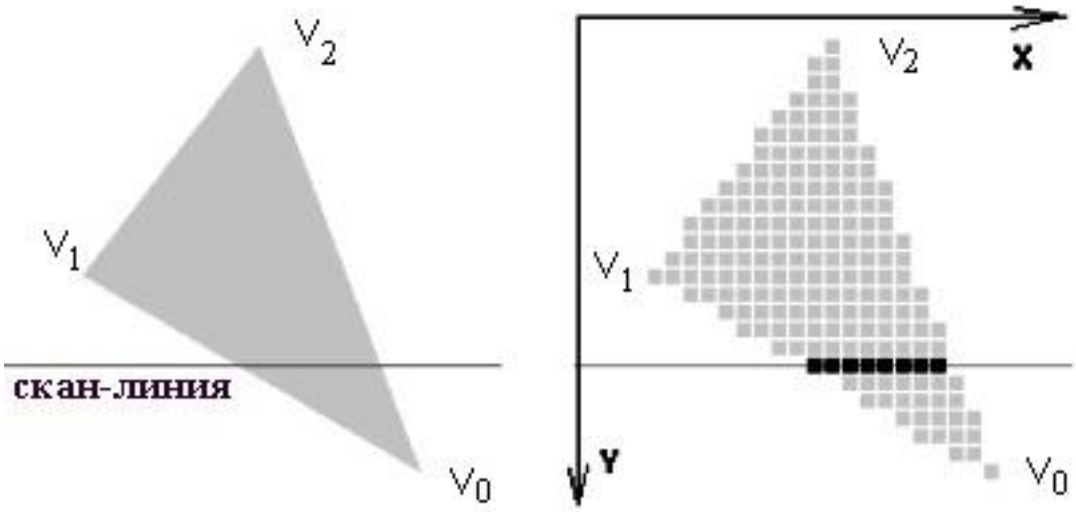

Рис. 4.2. Растеризация треугольников.

Отсортируем вершины так, чтобы V_2 была верхней, а V_0 – нижней. На рис. 4.2 показан нужный порядок сортировки вершин. Тогда

```
minY = V2.y
maxY = V0.y
```

Необходимо пройтись по всем линиям от `minY` до `maxY`. Рассмотрим какую-нибудь линию, соответствующую значению sy на экране:

```
V2.y ≤ sy ≤ V0.y
```

Если `sy` меньше `V1.y`, то такая скан-линия пересекает стороны V_1V_2 и V_2V_0. Если sy больше или равно `V1.y`, то такая скан-линия пересекает стороны V_1V_0 и V_2V_0.

Существует два принципиально разных подхода к задаче растеризации треугольника. В обоих подходах сначала набор исходных данных (координаты, текстурные координаты, освещенность и т.д.) интерполируется по ребрам треугольника, задавая тем самым начала и концы скан-линий. Затем осуществляется растеризация горизонтальной скан-линии с известными параметрами начала и конца.

Отличие в первой стадии. Первый подход выполняет интерполяцию по ребрам треугольника динамически, храня лишь значения для текущей горизонтальной скан-линии. Второй подход заполняет специальный массив, соответствующий началу и концу каждой скан-линии.

Первый подход более эффективный по производительности, зато второй подход более прост в реализации и проще в понимании.

Первый подход. Рассмотрим детали реализации алгоритма растеризации треугольника на базе первого подхода. Предположим, что в вершинах треугольника заданы следующие значения:

- экранные координаты *x, y;*
- цвет *r, g, b.*

Это соответствует следующей структуре данных:

```
struct WaVertex
{
  int   x, y;     // координаты точки экрана
  int   r, g, b;  // значения цветов в [0...255]
};
```

Сначала упорядочим вершины треугольника таким образом, что они будут перечислены против часовой стрелки. Это достигается построением векторов ($V_0 \rightarrow V_1$), ($V_1 \rightarrow V_2$). После чего строим нормаль к вектору ($V_0 \rightarrow V_1$) и умножаем скалярно эту нормаль на вектор ($V_1 \rightarrow V_2$). Если результат отрицательный, то вершины с индексами 1 и 2 надо поменять местами. Далее сортируем вершины треугольника таким образом, чтобы вершина с индексом 2 имела бы наибольшую координату.

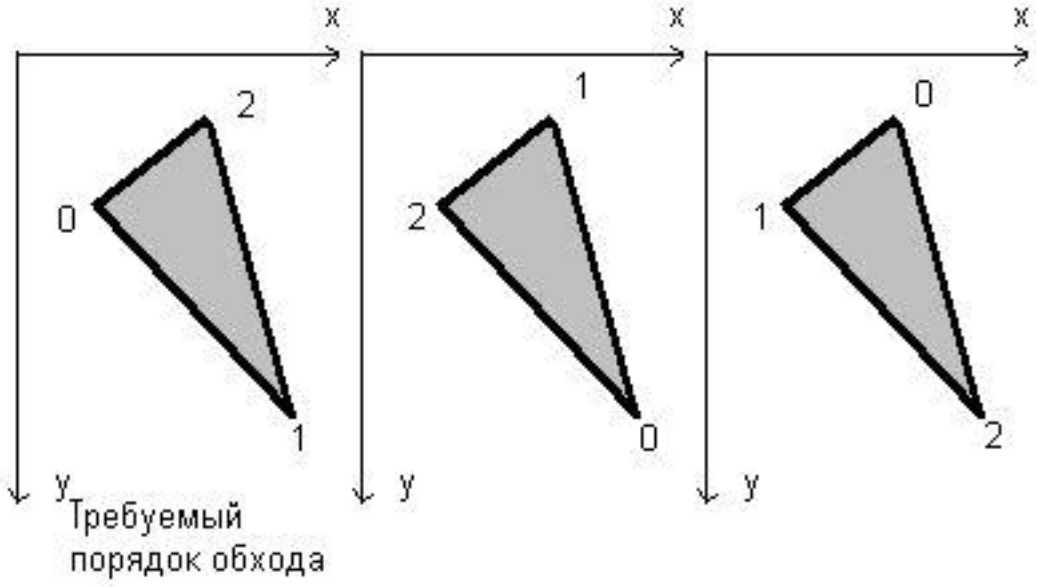

Рис 4.3. Возможные порядки обхода вершин треугольника.

На рис. 4.3 показаны три возможных варианта порядка вершин треугольника.

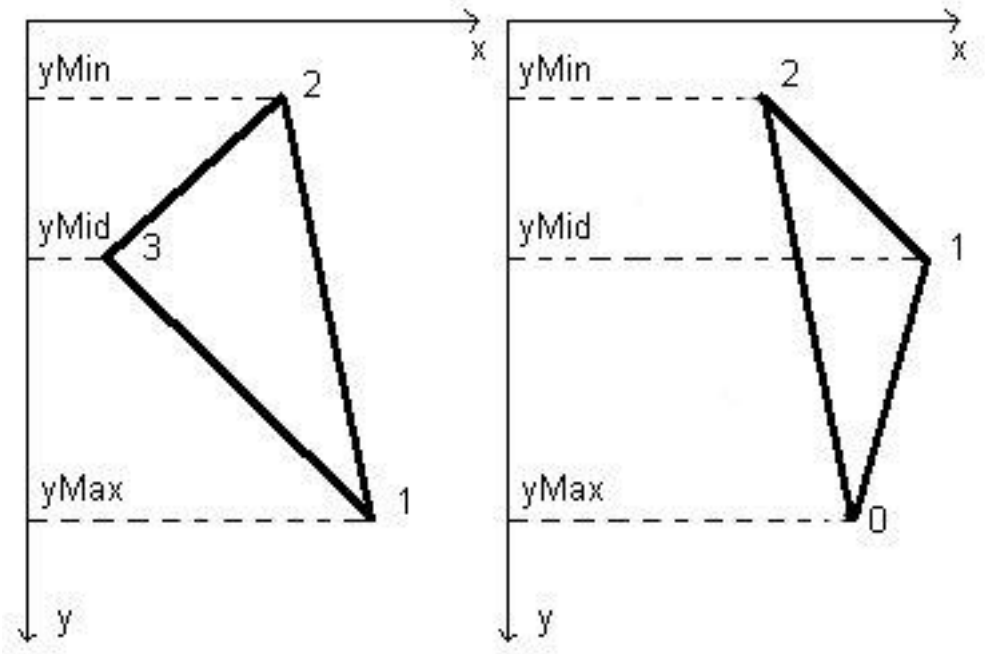

Рис 4.4. Возможные варианты расположения сторон треугольника после сортировки вершин.

На рис. 4.4 показаны два возможных варианта треугольников (после сортировки). В обоих случаях алгоритму необходимо найти три величины:

`yMin` – минимальная координата `y` в треугольнике;

`yMid` – “средняя” координата `y` в треугольнике. На этой координате происходит смена ребер;

`yMax` – максимальная координата в треугольнике.

Далее необходимо выполнить проверку площади треугольника. Если площадь оказывается нулевой, то такой треугольник не выводится. Затем необходимо определить значение шагов интерполируемых параметров вдоль горизонтальной скан-линии. Это вычисление можно сделать один раз для всего треугольника. Вывод формулы рассмотрим во втором подходе к растеризации.

Здесь просто напишем формулы определяющие приращения величин *r*, *g*, *b*:

$$stepR = \frac{(r_2 - r_0)(y_1 - y_2) + (r_1 - r_2)(y_0 - y_2)}{(x_2 - x_0)(y_1 - y_2) + (x_1 - x_2)(y_0 - y_2)};$$

$$stepG = \frac{(g_2 - g_0)(y_1 - y_2) + (g_1 - g_2)(y_0 - y_2)}{(x_2 - x_0)(y_1 - y_2) + (x_1 - x_2)(y_0 - y_2)};$$

$$stepB = \frac{(b_2 - b_0)(y_1 - y_2) + (b_1 - b_2)(y_0 - y_2)}{(x_2 - x_0)(y_1 - y_2) + (x_1 - x_2)(y_0 - y_2)}.$$

Далее процесс растеризации треугольника будет состоять из двух частей. В первой части растеризуются горизонтальные скан-линии, лежащие по координате “*y*” в интервале [*yMin*... *yMid-1*]. Во второй части – горизонтальные линии, лежащие по координате `y` в интервале [*yMid*...*yMax*]

Для изображения каждой части будем использовать одну и ту же идею. В каждой части известны начальная и конечная координата `y` для всех скан-линий. Скан-линия характеризуется левым и правым концами, на которых должны быть заданы все необходимые для растеризации параметры:

- экранная координата *x*;
- значение цвета *r*, *g*, *b*.

Такой блок параметров, который можно описать структурой данных:

```
struct ParamBlock
{
  int x;
  int r, g, b;
};
```

подходит также и для описания приращения этих параметров. Можно вместо этой структуры данных использовать исходную структуру данных для описания вершины (WaVertex). Так как поля этой структуры данных целые, инициализация интерполируемых параметров и их приращений (шагов) должна быть сделана с домножением на 2^{16} и дальнейшим приведением к целому типу данных. Это можно сделать с помощью такой процедуры:

```
static  inline void  makeIteratorAndStep(
                                  WaVertex *vertFrom,
                                  WaVertex *vertTo,
                                  WaVertex *iter,
                                  WaVertex *step
                                         )
```

```
{
iter->x = vertFrom->x << 16;
iter->y = vertFrom->y << 16;
iter->r = vertFrom->r << 16;
iter->g = vertFrom->g << 16;
iter->b = vertFrom->b << 16;
WaInt32   deltaY = vertTo->y - vertFrom->y;
if (!deltaY)
  return;
step->x = ((vertTo->x - vertFrom->x) << 16) / deltaY;
step->y = 1 << 16;
step->r = ((vertTo->r - vertFrom->r) << 16) / deltaY;
step->g = ((vertTo->g - vertFrom->g) << 16) / deltaY;
step->b = ((vertTo->b - vertFrom->b) << 16) / deltaY;
}
```

Обратите внимание на то, что в этой процедуре обязательно сравнение разности координат “*y*” с нулем. Это соответствует случаю, когда вершины 0 и 1 лежат на одной высоте.

Для прохода по скан-линиям потребуется две пары блоков величин (одной и той же структуры). Каждая пара блоков – это блок самих величин (обозначены iter в программном тексте) и блок приращений (обозначены step в программном тексте). Одна пара нужна для определения параметров на левом конце скан-линии, а другая – на правом.

После растеризации каждой скан-линии каждый блок величин необходимо обновить, используя блок приращений. Обновление легко выполняется следующей функцией:

```
static inline void doStep(WaVertex *iter, WaVertex *step){
  iter->x += step->x;
  iter->y += step->y;
  iter->r += step->r;
  iter->g += step->g;
  iter->b += step->b;
}
```

Обратите внимание, что этот подход может быть применен к растеризации треугольника со значительно большим количеством интерполируемых величин.

Отметим, что все операции, выполняемые внутри этих функциональных блоков – однородные, это позволяет эффективно реализовать их на SIMD-архитектурах.

Первая часть треугольника растеризуется следующей последовательностью:

```
for (y = minY; y < midY; y++)
{
    rasterizeHorLine(
                        screenBufferParameters,
                        &iteratorLeft,
                        &iteratorRight,
                        &stepPixel
                    );
    doStep(&iteratorLeft,   &stepLeft);
    doStep(&iteratorRight,  &stepRight);
}
```

Здесь:

`iteratorLeft` – блок параметров для левого конца скан-линии;

`iteratorRight` – блок параметров для правого конца скан-линии;

`stepPixel` – блок приращений для итерации внутри самой скан-линии (от пикселя к пикселю).

Так как внутри треугольника все параметры интерполируются линейно, то этот блок приращений один и тот же для всех скан-линий треугольника. После завершения растеризации первой части треугольника, надо перейти ко второй.

Второй подход. Усложним немного задачу. Добавим в растеризацию вывод текстуры. Для рендеринга треугольника необходимо задать следующие величины (для каждой вершины):

- координаты *x,y* экрана;
- координату *z* дальности вершины в пространстве (для перспективной коррекции и *z*-буфера);
- текстурные координаты *u,v;*
- другие свойства вершины, например освещенность, степень “затуманенности” и т.д.

Текстурные координаты – это 2D координаты в плоскости текстуры, используемой для рендеринга. На рис. 4.5 показан метод задания текстурных координат.

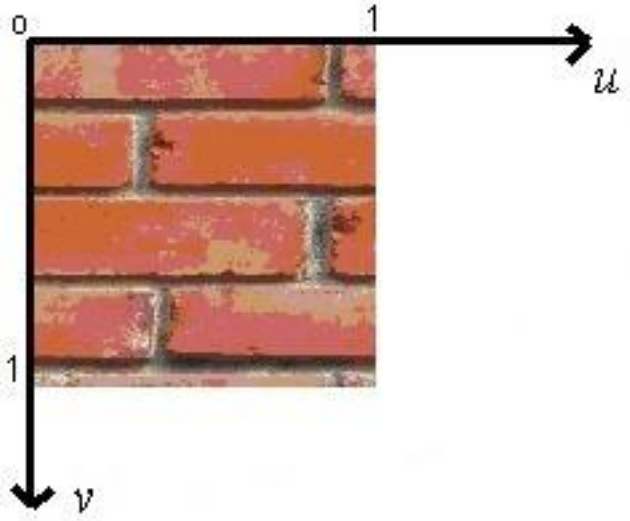

Рис. 4.5 . Система координат для задания текстуры.

Предварительная подготовка данных

На этом этапе заполняются два массива: для левых(ой) и правых(ой) ребер треугольника. Производится линейная интерполяция несимметричным алгоритмом DDA всех исходных величин относительно экранной координаты y. В результате получаем для каждой скан-линии треугольника (горизонтальной линии, рассекающей треугольник на высоте y-экранная) набор исходных величин слева (начало скан-линии) и справа (конец скан-линии).

Растеризация скан-линии треугольника

На этом этапе проходим построенные массивы от минимальной по y горизонтали до максимальной. Для каждой горизонтали уже вычислены координаты x начальное и x конечная на экране. Вычисляем также шаг текстурных координат u и v вдоль горизонтальной скан-линии треугольника. Для перспективно-корректной растеризации текстурированного треугольника этот шаг зависит от скан-линии и не линеен! Для упрощенной растеризации он общий для всех скан-линий

треугольника и может быть вычислен один раз в начальной стадии алгоритма.

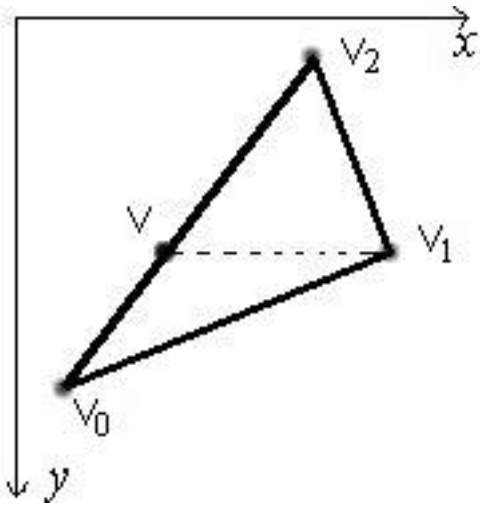

Рис 4.6. Вид треугольника для растеризации

Порядок вершин при обходе треугольника примем следующим:

- вершина 2 (V_2) имеет минимальную координату *y* на экране. (ось *y* направлена вниз на экране);
- вершины 2, 0, 1 располагаются в порядке обхода против часовой стрелки.

Для любого треугольника на входе всегда возможно переименовать вершины, чтобы данные правила соблюдались. Рис. 4.6 показывает пример требуемой нумерации вершин треугольника.

Для примера вычислим шаг для компоненты *u* текстурных координат. Для этого сначала надо найти координаты для вершины V (см. рис.4.6). Затем вычислим требуемый шаг, пользуясь координатами V и V_1.

Уравнение для отрезка V_2 - V_0 выглядит так в параметрической форме:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} + t \begin{bmatrix} x_0 - x_2 \\ y_0 - y_2 \end{bmatrix}$$

Параметр t при этом принадлежит диапазону [0...1]. Для вершины V у нас есть следующее условие $v_y = y_1$.

Таким образом,

$$y_1 = y_2 + t(y_0 - y_2);$$

$$t = \frac{y_1 - y_2}{y_0 - y_2}.$$

Для компоненты *u* имеем такое же уравнение, как и для экранных координат *x, y:*

$$u = u_2 + t(u_0 - u_2).$$

Заметим, что данные вычисления относятся к случаю, когда текстурные координаты линейно интерполируются по треугольнику. Основываясь на полученных выражениях. можно найти значения координаты *x* и значения *u* в вершине V:

$$x = x_2 + (x_0 - x_2)\frac{(y_1 - y_2)}{(y_0 - y_2)}; \qquad (4.1)$$

$$u = u_2 + (u_0 - u_2)\frac{(u_1 - u_2)}{(u_0 - u_2)}. \qquad (4.2)$$

Для вычисления шага будем использовать простую формулу

$$stepU = \frac{u_1 - u}{x_1 - x}.$$

Упрощая (4.1) получим:

$$x_1 - x = \frac{(x_1 - x_2)(y_0 - y_2) - (x_0 - x_2)(y_1 - y_2)}{(y_0 - y_2)}.$$

То же самое для *u*, используя (4.2):

$$u_1 - u = \frac{(u_1 - u_2)(y_0 - y_2) - (u_0 - u_2)(y_1 - y_2)}{(y_0 - y_2)}.$$

Окончательно получаем выражение для шага по координате *u:*

$$stepU = \frac{(u_1 - u_2)(y_0 - y_2) - (u_0 - u_2)(y_1 - y_2)}{(x_1 - x_2)(y_0 - y_2) - (x_0 - x_2)(y_1 - y_2)}.$$

Оптимизации внутреннего цикла растеризации

На внутреннем цикле, как видно из предшествующего описания, осуществляется последовательность целочисленных сложений изменяемых от пикселя к пикселю величин и их приращений. Количество таких сложений варьируется в зависимости от того, сколько свойств треугольника мы интерполируем. Не все предлагаемые оптимизации внутреннего цикла идеально подходят для всех платформ. Конкретное

решение о методе оптимизации надо принимать в зависимости от свойств конкретного процессора.

Использование половин регистров

Рассмотрим оптимизацию, основанную на использовании половин регистров. Предполагается, что текстура, используемая для рендеринга, имеет размеры 256×256. На самом деле существенно, чтобы расстояние между двумя строками текстуры было 256 байт. Это ограничение позволяет представить данные текстуры так, что алгоритм будет работать для четырех текстур размером 64×64 представленных в памяти так, как показано в табл. 4.1:

Таблица 4.1. Размещение строк текстуры в памяти

0	64	128	192
L0, T0	L0, T1	L0, T2	L0, T3
L1, T0	L1, T1	L1, T2	L1, T3
…	…	…	…
L63, T0	L63, T1	L63, T2	L63, T3

Здесь обозначены: L0, L1, … – строки текстур. T0, T1, T2, T3 – номера текстур. В памяти хранятся четыре текстуры в едином пространстве, причем строки текстур перемешаны! Если количество текстур в программе не кратно 4, то будем заполнять оставшееся свободное пространство памяти незначащими нулями.

Будем вычислять значения *u*, *v*, и шаги по *u*, *v* с точностью 8 бит. Предполагаем, что целые значения шагов также помешаются в 8 бит. Тогда имеем для текущего значения *u* представление:

регистр ECX

		uInt	uFrac
		8	8

Здесь uInt, uFrac – целая и “вещественная” части значения *u*.

Для текущего значения *v* представление:

регистр EAX

		vInt	vFrac
		8	8

Шаги по *u* и *v* (назовем их *du, dv*) можно затолкнуть в один регистр:

регистр EDX

dvInt	dvFraq	duInt	duFrac

Вычисление следующего значения пары *u, v*:

```
add   CL, DL
adc   CH, DH
rol   EDX, 16
add   AL, DL
adc   AH, DH
rol   EDX, 16
mov   BL, CH
mov   BH, AH
mov   BL, [EBX] ; BL содержит нужный цвет из текстуры
```

Эти инструкции вставляются во внутренний цикл. Улучшение этого цикла состоит в “раскручивании” цикла, тогда исчезнут инструкции inc EDI, jnz @@inner. Недостаток кода – использование частей регистров, что замедляет работу современных процессоров.

Оптимизация внутреннего цикла на основе представления величин *u, v* в одном регистре

Также можно представить текущие координаты *u* и *v* (а также их приращения) внутри одного 32-битного регистра, как показано на рис. 4.7:

25	18	14	7	0
uInt	uFrac	Не используется	vInt	vFrac

Рис. 4.7. Представление текстурных координат в 32-битном регистре

В таком представлении получить смещение в текстуре (128×128) можно так:

```
mov EBX , EAX
mov EDX , EAX
shr EBX , 25
and EDX , 03f80  ; (127 << 7)
or  EBX , EDX
```

А следующее значение пары *u, v* получается одним сложением:

```
add EAX , EBP
```

Самомодифицирующийся код

Техника оптимизации, основанная на применении самомодифицирующегося кода, применяется не только для задач оптимизации внутренних циклов. Эта техника часто используется в системах защиты программ от несанкционированного доступа, в системах шифрования данных, а также во вредоносных вирусных программах. Идея такой техники состоит в том, что выполняемый код не находится в статичном виде в исполняемом модуле, а генерируется динамически другой частью кода, либо не генерируется полностью, а создается частично, по “кускам”, составленным из другого кода. Такая техника не всегда применима. Например, некоторые операционные системы и процессоры имеют аппаратные средства защиты от выполнения самомодифицирующегося кода.

Для нашей оптимизации будем повторять одну и ту же последовательность действий несколько раз. Например, 16. Оптимизация будет достигаться за счет того, что количество ветвлений на всю скан-линию будет уменьшено по сравнению с классической версией алгоритма, и, что самое главное, вычисление очередной текстурной координаты будет спрятано внутри этого куска машинного кода, и выполнено один раз, на этапе подготовки кода. Далее этот фрагмент машинных команд будет применен необходимое количество раз для скан-линии произвольной длины, перед каждым применением потребуется лишь настроить некоторые регистры правильно.

Для каждой скан-линии предварительный код (более медленный) сначала прописывает в код необходимые смещения в текстуре, а затем вызывает измененный код:

```
mov AL,[ESI+XXX]
mov [EDI+0],AL
mov AL,[ESI+XXX]
mov [EDI+1],AL
. . .
mov AL,[ESI+XXX]
mov [EDI+15],AL
```

Здесь `XXX` – не определенное к началу выполнения растеризации смещение. Для выполнения такого приема необходимо знать операционный код инструкции `mov AL,[ESI+XXX]` а также быть уверенным в том, что операционная система позволяет модифицировать код.

Предварительная подготовка будет представлять собой цикл из 16 итераций, в котором просчитываются текстурные координаты (u, v) для последовательности из 16 точек на экране. Предварительная подготовка будет воспринимать результирующий код в качестве данных и будет изменять в нем числа `XXX` на актуально рассчитанные смещения в текстуре для заданной линии на экране.

5. АЛГОРИТМЫ ДЕЛЕНИЯ ЦЕЛЫХ ЧИСЕЛ

Из всех элементарных арифметических операций деление – наиболее сложная, для нее требуется максимальное количество вычислительных ресурсов (либо в кремнии, чтобы реализовать деление аппаратно, либо во времени, чтобы реализовать его программно). Во многих компьютерных программах деление встречается значительно реже, чем сложение, вычитание и умножение, поэтому многие процессоры, предназначенные для цифровой обработки сигналов (DSP) не имеют встроенной инструкции деления. Очень важный аспект дизайна многих современных процессоров – их цена. В конкурентной борьбе за снижение цены многие производители процессоров отказываются от выпуска сложных решений, и поэтому большинство дешевых карманных вычислительных устройств также “не умеют” делить целые и вещественные числа.

Семейство микропроцессоров класса ARM (используемых в большинстве мобильных телефонов, коммуникаторов и карманных компьютеров) не имеет встроенной инструкции деления. В связи с этим программирование операции деления до сих пор имеет актуальное значение (как бы это странно ни звучало), и до сих пор проводятся различные исследования оптимизации алгоритма деления.

Современные вычислительные системы уже далеко продвинулись в сторону более сложных вычислительных процессов, реализованных на аппаратном уровне, поэтому задача деления целых или вещественных чисел может показаться давно решенной и очень несовременной. Для крупных вычислительных устройств – персональных компьютеров, рабочих станций и серверов – это действительно так. Но электроника развивается не только в сторону укрупнения вычислительных устройств, но также по пути создания крохотных, миниатюрных вычислительных устройств, помещающихся на ладони, в кармане и т.д. Для таких устройств основными критериями конкурентоспособности являются минимальные размер, энергопотребление и стоимость. Последнее требование, возможно, является одним из наиболее существенных. Процессоры класса ARM удовлетворяют этим требованиям и поэтому активно используются в качестве основы для новых микроустройств. Быстродействие крохотных вычислительных устройств тоже немаловажно для конкурентной борьбы на рынке техники, и поэтому оптимальные вычисления становятся крайне важной задачей для многих производителей аппаратуры, не перестающих изобретать все новые и новые алгоритмы для реализации операций, которые не могут быть выполнены аппаратно в силу тех или иных причин.

Алгоритмы целочисленного деления до сих пор являются важной темой для научных семинаров и исследований. В настоящее время существует несколько алгоритмов деления, применяемых в современных аппаратных и программных средствах:

- алгоритм с умножением (multiplicative);
- алгоритм Ньютона–Рафсона;
- простейший алгоритм с вычитанием (крайне редко используется);
- одноразрядный алгоритм деления в столбик;
- алгоритм SRT (по именам создателей: Sweeny, Robertson, Tocher);
- алгоритм Голдшмидта.

Алгоритм с умножением

Воспользовавшись приблизительным равенством

$$\frac{X}{Y} = \frac{(2^{32} + Y - 1)}{Y} X \frac{1}{2^{32}},$$

можно получить простой алгоритм деления, создав заранее таблицу для

$$K(Y) = \frac{(2^{32} + Y - 1)}{Y}.$$

При этом деление будет осуществляться умножением со сдвигом

$$\frac{X}{Y} = K(Y) X \frac{1}{2^{32}} = (K(Y) X) >> 32.$$

Этот алгоритм не точен; точность алгоритма выше для относительно небольших значений *Y*. Также алгоритм имеет огромный недостаток в том, что приходится хранить таблицу чисел *K(Y)*, причем размер таблицы совпадает с диапазоном возможных делителей. В тех задачах, где делители могут быть взяты из небольшого диапазона, этот алгоритм можно успешно применять.

Для примера вычислим 18 243 / 58 (что равно 314.5344827):

```
K(58) = 74051161
X/Y = (K(58)•18243) >> 32 = 314
```

Алгоритм Ньютона–Рафсона

Задача очень похожа на предыдущую. Требуется найти

Y = 1 / X,

или, что то же самое,

X = 1 / Y,

X - 1 / Y = 0,

т.е. требуется найти такое значение *Y*, что выражение

f(Y) = X - 1 / Y.

будет равно 0. Предлагается итерационный процесс, который приведет значение *f(Y)* к 0 с каждым новым приближением *Y*:

$$Y_1 = Y,$$

$$Y_2 = Y_1 - \frac{f(Y_1)}{f'(Y_1)} = Y_1 - \frac{(X - \frac{1}{Y_1})}{\frac{1}{Y_1^2}} = Y_1 - X * Y_1^2 + Y_1 = Y_1(2 - X * Y_1)$$

и так далее. Например, рассмотрим процесс нахождения 1 / 0.5. В качестве начального приближения $Y_1 = 1$:

Y_2 = 1 * (2 – 0.5 * 1) = 1.5,

Y_3 = 1.5 * (2 – 0.5 * 1.5) = 1.875,

Y_4 = 1.875 * (2 – 0.5 * 1.875) = 1.992,

Y_5 = 1.992 * (2 – 0.5 * 1.992) = 1.99996.

Процесс можно прервать, когда значение Y_i будет отличаться от предыдущего не более чем на некоторую заранее заданную величину (ошибку).

На базе этого классического алгоритма Ньютона–Рафсона можно реализовать целочисленный алгоритм, воспользовавшись идеей из предыдущего раздела. Будем вычислять $f(Y) = X - K/Y$, где K – какое-либо большое число, степень двойки. Возьмем, к примеру, $K = 65536$:

$$Y_{i+1} = Y_i - \frac{f(Y_i)}{f'(Y_i)} = Y_i - \frac{(X - \frac{K}{Y_i})}{\frac{K}{Y_i^2}} = Y_i - XY_i^2 \frac{1}{K} + Y_i = Y_i(2 - \frac{XY_i}{K}).$$

Для примера опять же посчитаем 18243 / 58, при этом берем $K = 65536$ Пусть $Y_1 = 600$:

Y_2 = 600 * (2 – (58*600) / 65536) = 881,

Y_3 = 881 * (2 – (58*881) / 65536) = 1075,

Y_4 = 1075 * (2 – (58*1075) / 65536) = 1127,

Y_5 = 1127 * (2 – (58*1127) / 65536) = 1129,

Y_6 = 1129 * (2 – (58*1129) / 65536) = 1129.

На этом итерационный процесс сходится, мы получаем в целых числах точный ответ 1 / 58 = 1129 (с точностью 16 бит). Далее осталось найти:

18243 / 58 = (18243 * 1129) / 65536 = 314.

Подчеркнем, что при программной и аппаратной реализации вместо деления на 65536 можно (и нужно) использовать битовый сдвиг вправо на 16 бит.

Необходимо отметить, что некоторые процессоры имеют аппаратные инструкции вычисления некоторого количества приближений (1 / X). И в зависимости от того, насколько важна точность вычисления обратной

величины, можно в той или иной степени ускорить реализацию алгоритма, содержащего операцию деления.

Простейший алгоритм с вычитанием

Заданы делимое и делитель в целых числах. Для простоты считаем, что оба числа положительные. Требуется найти частное и остаток. Пусть X – делимое, Y – делитель, Z – частное, Q – остаток, тогда:

$X = Y * Z + Q.$

Требуется найти Q и Z. Условие $X > Y$ естественно, в противоположном случае задача тривиальна ($Z = 0$, $Q = X$)

Простейший (очень неэффективный) алгоритм состоит в последовательном вычитании Y из X до тех пор, пока результат вычитания положителен. Количество итераций вычитания будет равняться частному Z, а последний положительный результат итерации будет равен остатку Q.

На языке С данный алгоритм может быть закодирован следующим образом:

```
int Divide(int X, int Y, int *Qret)
{
  int   Z, Q;
  int   Qnext;
  Z = 0;
  Q = X;  // изначально остаток равен делимому
  while(1)
  {
    Qnext = Q - Y; // вычислить следующий остаток
    if (Qnext < 0) // если меньше нуля, то выход
      break;
    Q = Qnext;
    Z++;
  }
  *Qret = Q;
  return Z;
}
```

Заметим, что сложность данного алгоритма сильно зависит от отношения между делителем X и делимым Y. В худшем случае количество

итераций равно максимально допустимому значению *X*, т.е. для целых 16-битных чисел худшее количество итераций равно 65535! Именно этот факт делает простейший алгоритм деления практически неприменимым для задачи деления в общем виде. Однако в некоторых задачах можно воспользоваться этим алгоритмом, если диапазон делимых относительно мал.

Одноразрядный алгоритм деления “в столбик”

Вспомним школу! Более эффективный алгоритм, как ни странно, известен из школьной программы. Рассмотрим деление на примере 429 / 17. (429 / 17 = 25, в остатке 4), что в шестнадцатеричном представлении 0x1ad / 0x11 = 0x11, остаток 0x4, в двоичном 110101101 / 10001 = 11001, остаток 100.

```
110101101 | 10001
10001       11001
 10011
 10001
    101
    000
    1010
    0000
    10101
    10001
      100
```

Заметим, что каждый шаг алгоритма порождает в качестве результата 1 бит (0 или 1). После получения очередного бита результата частное сдвигается влево на один бит и полученный новый бит добавляется к частному. Этот алгоритм в своей основе содержит умножение и вычитание.

Если разобраться в алгоритме более формально, то можно представить себе, что для начала в делитель добавляется справа некоторое количество 0, чтобы сравнять битовую длину делимого и делителя. На каждом шаге производится попытка вычесть делитель из текущего делимого. Если результат положительный, то в частное добавляется бит 1, если отрицательный – то бит 0.

Дополним делитель добавочными битами слева до 9 бит (как в делимом): 100010000.

После каждого шага сдвигаем делитель вправо на один бит. Процесс проиллюстрирован ниже:

```
110101101  |100010000
100010000   1 -> в частное вдвигаем налево
 10011101  |10001000
 10001000   1 -> в частное вдвигаем налево
    10101  |1000100
        0   0 -> в частное вдвигаем налево
    10101  |100010
        0   0 -> в частное вдвигаем налево
    10101  |10001
    10001   1 -> в частное вдвигаем налево
      100
```

Итак, первый, предварительный шаг алгоритма начинается с определения номера старшего бита для делимого и делителя. Многие процессоры умеют вычислять номер наиболее старшего значимого бита в числе одной инструкцией. Если такой инструкции нет, то придется написать эту простейшую процедуру:

```
int GetNumBits(int V)
{
  int Vtest;
  int NumBits;

  NumBits = 0;
  Vtest   = V;
  while (Vtest != 0)   // пока старший бит не достигнут
  {
    NumBits++;
    Vtest = Vtest >> 1;
  }
  return NumBits;
}
```

Заметим, что количество циклов этой процедуры в худшем случае равно 31 итерации (для 32-битных целых чисел). Или, иными словами,

вычислительная сложность равна количеству битовых разрядов допустимых для *X*, *Y* чисел. Теперь осталось привести весь алгоритм целиком:

```
int Divide(int X, int Y, int *Qret)
{
  int   Z;
  int   TopX, TopY, BitDiff;
  int   YY, Sub, Mask;
  if (X < Y)
  {
    *Qret = Y;
    return 0;
  }
  TopX = GetNumBits(X);
  TopY = GetNumBits(Y);
  BitDiff = TopX - TopY;
  // уравнять делитель с делимым по битовой длине
  YY = Y << BitDiff;
  // частное точно содержит этот старший бит
  Mask = Z = 1 << BitDiff;
  while (X > Y)
  {
    Sub = X - YY;
    if (Sub >= 0)
    {
      X = Sub;
      Z |= Mask;
    }
    YY >>= 1; Mask >>= 1;
  }
  *Qret = X;
  return Z;
}
```

Еще раз отметим, что на многих процессорах функция GetNumBits реализуется аппаратно одной инструкцией! При отсутствии такой аппаратной инструкции, рекомендуется не вычислять битовую длину *X* и *Y*, а сдвигать вправо *Y* до тех пор, пока *Y* < *X*. В качестве *YY* взять

предпоследний результат сдвига. Текст данного алгоритма предлагается написать читателю самостоятельно.

Вычислительная сложность этого алгоритма деления целых чисел равна опять же максимально допустимому количеству битовых разрядов *X*. Этот алгоритм, написанный в интуитивно понятном виде, содержит ветвление внутри основного цикла, что, безусловно, понижает производительность всего алгоритма в целом (на большинстве процессоров). Путем несложной оптимизации внутренний цикл алгоритма переписывается в цикл, не содержащий ветвления:

```
while (X > Y)
{
  Sub = X - YY;
  MaskNeg = Sub >> 31;
  MaskPos = ~MaskNeg;
  X = (MaskPos & Sub) | (MaskNeg & X);
  Z |= (MaskPos & Mask);
  YY = >>= 1; Mask >>= 1;
}
```

На различных процессорах (в зависимости от того, насколько быстро работает условный переход) следует применять одну из версий этого алгоритма.

Алгоритм поиска старшего бита

Как уже упоминалось ранее, для одноразрядного алгоритма деления двух чисел “в столбик” необходимо выравнять битовые длины делимого и делителя путем добавления нулей в делитель справа. Функция GetNumBits, которая была приведена выше, не самая эффективная для этой цели. В ней есть цикл длиной 32 итерации и условный переход. Попробуем разработать алгоритм поиска старшего бита числа, в котором не будет этих недостатков. Будем предполагать, что мы имеем дело с 32-битными числами. Далее будет понятно, что это не накладывает принципиальных ограничений на алгоритм, и он может быть легко переделан для 16-битных или 64-битных чисел.

Будем двигаться от простого к сложному. Представим, что задача упрощена так, что на входе задано 2-битное число (т.е. 0, 1, 2 или 3), и необходимо узнать номер старшего бита (0 или 1)

Если наше число 2 или 3, то у него установлен бит №1. Это можно легко понять, если вычесть из нашего числа 2. Если входное число 2 или 3, то результат такого вычитания неотрицательный. Этим можно воспользоваться при вычислении.

Пусть *t* – входное число. Тогда:

```
m = ~( (t -2) >> 31) & 1;
```

будет искомым значением номера старшего бита,

m будет равно 1, если в *t* бит №1 установлен, или 0 — в противном случае.

Теперь немного усложним поставленную задачу. Пусть входное число – 4-х битное (т.е. 0, 1, 2, ..., 14, 15):

3	2	1	0

Сначала выясним, есть ли единичный бит в старшей половине числа (установлен бит 2 или 3). Если установлен, то ответ будет 2 или 3;

```
m = ~( (t -4) >> 31) & 2;
```

Будет равно 2 в том случае, если в *t* есть единичный бит в старшей половине, или 0 если нет. Осталось теперь выяснить, есть ли единичный бит в позиции 3, а также необходимо узнать, есть ли единичный бит в позиции 1. Для этого сдвинем вправо исходное число на *m* бит и таким образом сведем задачу к предыдущей задаче:

```
n = 0;
m = ~( (t -4) >> 31) & 2;
n |= m;
t = t >> m;
m = ~( (t -2) >> 31) & 1;
n |= m;
```

Такая последовательность действий найдет номер старшего единичного бита в 4-х битном числе *t*. Распространяя эту последовательность далее, получим для 32-битного числа следующую эффективную функцию поиска номера старшего установленного бита:

```
int GetDegree(int t)
{
  int n, m;
```

```
  n = 0;
  m = ~( (t - (1L<<16)) >> 31) & 16;
  n |= m;
  t = t >> m;
  m = ~( (t - (1L<<8)) >> 31) & 8;
  n |= m;
  t = t >> m;
  m = ~( (t - (1L<<4)) >> 31) & 4;
  n |= m;
  t = t >> m;
  m = ~( (t - (1L<<2)) >> 31) & 2;
  n |= m;
  t = t >> m;
  m = ~( (t - (1L<<1)) >> 31) & 1;
  n |= m;
  t = t >> m;
  return n;
}
```

Высокоразрядный алгоритм деления

Вышеописанный алгоритм (один из его вариантов) аппаратно реализован во многих современных процессорах. Если же приглядеться к школьному алгоритму деления “в столбик”, то можно заметить, что важной чертой этого алгоритма является “разрядность” (radix) чисел, которые мы делим. Для описанного алгоритма разрядность равна 1 (за одну итерацию алгоритма получаем 1 бит результата). Естественно предположить улучшение алгоритма с использованием разрядности, большей, чем 1. Например, за одну итерацию добавлять сразу 4 или 8 бит к результату (частному). Очевидно, что тем самым, мы уменьшим общее количество итераций в алгоритме, что будет достигнуто ценой более сложной генерации очередного “разряда” частного. Более “продвинутый” алгоритм отличается от однобитного алгоритма как раз тем, что за одну итерацию добавляем сразу несколько битов к результату. Алгоритм усложняется тем, что если в предыдущем случае к результату прибавлялась 1 или 0 (в зависимости от результата вычитания), то в этом

алгоритме придется вычислять очередной разряд в результате, делая предположение об очередном разряде частного. Именно этот этап высокоразрядного алгоритма деления подлежит возможной оптимизации.

Рассмотрим простейший подход. Для генерации очередного разряда будем умножать текущее делимое на k(k = 0, 1, 2, …) до тех пор, пока результат такого умножения меньше текущего делимого *YY*. В качестве разрядности возьмем 4.

Код модифицированного алгоритма приведен ниже (заметим, что для поиска начального значения *YY* использовалась “подгонка” значения *Y* сдвигами направо по 4 бита):

```
int Divide(int X, int Y, int *Qret)
{
  int   Z;
  int   YY, R, Value;
  if (X < Y)
  {
    *Qret = X;
    return 0;
  }
  // сдвинуть Y влево максимально
  YY = Y;
  while (YY < X)
  {
    YY = YY << 4;
  }
  YY = YY >> 4;
  Z  = 0;
  while (X > Y)
  {
    // найти следующее значение R
    R = 1;
    Value = YY;
    while (Value < X)
    {
      R++;
      Value += YY;
    }
```

```
    R--;
    Value -= YY;
    X = X - Value;
    Z = Z << 4;
    Z |= R;
    YY = YY >> 4;
  }
  *Qret = X;
  return Z;
}
```

Отметим, что в данном алгоритме количество итераций во внутреннем цикле равно 15. Оптимизация этого алгоритма состоит в упрощении внутреннего цикла. Все современные аппаратные версии алгоритмов деления базируются на этой схеме с улучшениями, относящимися к определению очередного значения *R*.

Если расписать итерационный процесс более формально, получим:

X – делимое;

D – делитель;

Q – частное;

R – остаток;

r – порядок (radix).

Эти обозначения приняты во многих исследовательских работах по данной проблеме. В качестве первого приближения остатка берут $R^0 = X$.

Требуется найти X / D. Основа итерационного вычисления:

$R^{(j+1)} = r\, R^{(j)} - q_{j+1}\, D.$

Для первых итераций:

$R^{(1)} = r\, R^{(0)} - q_1\, D;$

$R^{(2)} = r\, R^{(1)} - q_2\, D = r\,(\, r\, R^{(0)} - q_1\, D) - q_2\, D = r^2\, R^{(0)} - r\, q_1\, D - q_2\, D.$

И далее получим:

$R^{(n)} = r^n\ R^{(0)} - (r^{n-1}\ q_1 + \ldots + r\ q_{n-1} + q_n)\ D.$

После деления этого выражения на $(r^n\ D)$ получим:

$$\frac{R^{(0)}}{D} = \sum_{j=1}^{n} r^{j} * q_{j} + \frac{r^{-n} R^{(n)}}{D}\ .$$

В итоге:

$$Q = \sum_{j=1}^{n} r^{-j} q_{j}\ ,$$

$R = r^{-n} R^{(n)}$.

Поиск очередного значения q_{i+1} является темой многих исследований и аппаратных реализаций алгоритма деления. Все такие алгоритмы можно условно поделить на две группы:

- алгоритм без возврата;
- алгоритм с возвратом.

Безвозвратный (non-restoring) алгоритм получает очередное значение q_{i+1} совершенно точно. Но ценой этого является внутренний цикл для уточнения данного значения, на базе некоторого предположения (Различные алгоритмы используют разные схемы для этого предположения).

Высокоразрядный целочисленный алгоритм деления с возвратом

Как уже было отмечено, на каждом шаге итерации алгоритм получает сразу несколько разрядов в ответе. Алгоритм с возвратом получает приближенное значение результата, а потом его можно уточнить.

Для простоты примера рассмотрим деление с разрядностью 10, т.е. то, что наиболее приближено к делению в столбик.

Пусть надо разделить число
X = 1 875 930 243 (0x6fd06c83) на D = 85 629 (0x14e7d). В результате должно получиться 21 907 (0x5593).

Сначала получим приблизительный ответ для первого разряда результата (десятичного разряда). Будем вычислять наименьшее возможное значение очередного разряда ответа, и в дальнейшем добавлять к нему числа:
1 800 000 000 / 899 999 999 = 2.

Мы заменили все значения разрядов после 18 на 0 в делимом, а все разряды после 8 на 9 в делителе, т.е. нашли минимальное возможное значение разряда. Итак, первая приблизительная десятичная цифра ответа 2:
R = 1 875 930 243 – 85 629 * 20 000 = 163 350 243.

Умножать надо на 20 000 (четыре нуля дописали), потому что для “уравнивания»” разрядов надо добавить 4 десятичных разряда к делителю.

Проверим, может, мы ошиблись, и, возможно, первая цифра— не 2, а 3, 4 и т. д.:

R = 1 875 930 243 – 85 629 • 30 000 = - 692 939 757.

Значит, цифру нашли точно с первого раза. Итак, первая цифра ответа 2, новое делимое 163 350 243.

Ищем следующую приблизительную цифру ответа для деления:

163 350 243 / 85 629.

Это будет 160 000 000 / 89 999 999 = 1,

R = 163 350 243 – 85 629 • 1000 = 77 721 243.

Умножать надо на 1000 (три нуля дописали), потому что для “уравнивания” разрядов пришлось добавлять 3 разряда к делителю. Проверим, может, мы ошиблись, и, возможно, первая цифра— не 1, а 2, 3 и т.д.:

R = 163 350 243 – 85 629 • 2 000 = - 7 907 757.

Нашли правильно!

Итак, следующая цифра результата 1. Далее ищем ответ для

77 721 243 / 85 629:

77 000 000 / 8 999 999 = 8.

На этом этапе мы получили неточную цифру очередного разряда ответа (точная цифра 9):

R = 77 721 243 – 85 629 • 800 = 9 218 043.

Дописываем два нуля. Проверим, может быть, это 9:

R = 77 721 243 – 85 629 • 900 = 655 143.

Итак, мы скорректировали ошибку, и точная цифра следующего разряда 9. Ищем новый разряд для 655 143 / 85 629:

600 000 / 89 999 = 6.

Дописали 0 нулей, а на предыдущем шаге дописывали два. Это значит, что очередная цифра результата деления 0, и мы нашли уже следующую приближенную цифру:

R = 655 143 – 85 629 • 6 = 141 369.

Проверим, может быть, это 7:

R = 655 143 – 85 629 • 7 = 55 740.

Проверим, может быть, это 8:

R = 655 143 – 85 629 • 8 = - 29 889.

Значит последняя цифра результата 7. Итак, собрав все цифры ответа, получим 21 907.

При этом мы еще получили и остаток деления
$R = 55\ 740$.

Легко убедиться, что все правильно. Заметим, что все полученные приближенные цифры на каждой итерации можно записать в таблицу. Таким образом, этот алгоритм тратит некоторый необходимый объем памяти для предсказания очередной цифры ответа в разряде, а также небольшое время на итерацию для уточнения.

На базе этого алгоритма можно построить алгоритм с разрядностью 4, 8 и 16.

6. ПРИЕМЫ ОПТИМИЗАЦИИ ПРОГРАММ

Многие приемы оптимизации программ на языке ассемблера не являются общими и не применимы для любого процессора, однако приемы оптимизации необходимо изучить для того, чтобы на конкретно взятом процессоре попробовать улучшить быстродействие или качество вашей программы. Общие рекомендации:

- Используйте по возможности самый современный компилятор, позволяющий максимально учесть особенности архитектуры.
- Минимизируйте количество используемых глобальных переменных и запутанное управление выполнением.
- Где возможно, используйте const модификатор, избегайте register (компилятор сам сообразит, что можно заменить регистрами)
- Избегайте косвенных вызовов.
- Улучшайте предсказуемость ветвлений. Это одна из наиболее значимых оптимизаций для PII и PIII.
- Используйте, где это уместно, SIMD расширения архитектуры.
- Избегайте задержек при использовании части регистра (при последующем обращении к полному регистру).
- Применяйте правильное выравнивание данных.

- Переставляйте инструкции в коде так, чтобы минимизировать промахи в кэше, и оптимизируйте предвыборку в кэш.
- Избегайте чтения малого количества данных после записи большого количества данных и наоборот.
- Избегайте самомодифицирующегося кода (для Windows XP).
- Избегайте располагать данные в сегменте кода.
- Вычисляйте адрес для записи на возможно более ранней стадии.
- Избегайте инструкций, требующих для выполнения более четырех микроопераций и инструкций, код которых длиннее чем 7 байт.
- Старайтесь повышать параллелизм выполнения кода.

Копирование памяти

Здесь мы научимся копировать блок памяти быстрее, чем это делает стандартная процедура C memcpy(), которая по сути представляет собой инструкцию rep movsd + rep movsb для остатка байтов, не кратного 4.

Первое улучшение (выравнивание адресов). Стандартная процедура выполняет копирование двойных слов, даже стартовый адрес не выравнен на границу двойного слова. Исключение – библиотека Watcom C, в которой memcpy при копировании выравнивает адреса. При копировании из системной памяти в системную память ускорение достигает отношения 3.19/2.05 тактов на 1 байт. Приведем улучшенный вариант memcpy:

```
void MemCopyAligned(void *pDest, void *pSource,unsigned
long nCount)
{
  while ( ( ( ( long ) pDest ) & 3 ) && nCount )
  {
     (char*)pDest = *(char *)pSource;
     pDest = (void*)(((char*)pDest)+1);
     pSource = (void*)(((char*)pSource)+1);
     nCount--;
  }
  memcpy(pDest,pSource,nCount);
}
```

Второе улучшение (спаривание и отказ от rep movsd). На “Пентиум-Про” rep movsd занимает 1.8 тактов на двойное слово, если источник и приемник располагаются в кэше L1. Процедура копирования, приведенная ниже, занимает 1.4 такта на двойное слово (обратите внимание, что все инструкции спариваются):

```
.REPEAT
mov EAX,[ESI+ECX+0*4]
mov EBX,[ESI+ECX+1*4]
mov [EDI+ECX+0*4],EAX
mov [EDI+ECX+1*4],EBX
mov EAX,[ESI+ECX+2*4]
mov EBX,[ESI+ECX+3*4]
mov [EDI+ECX+2*4],EAX
mov [EDI+ECX+3*4],EBX
mov EAX,[ESI+ECX+4*4]
mov EBX,[ESI+ECX+5*4]
mov [EDI+ECX+4*4],EAX
mov [EDI+ECX+5*4],EBX
mov EAX,[ESI+ECX+6*4]
mov EBX,[ESI+ECX+7*4]
mov [EDI+ECX+6*4],EAX
mov [EDI+ECX+7*4],EBX
add ECX,32
.UNTIL ZERO? ; 9 тактов на 32 байта
```

Третье улучшение (предварительная инициализация кэша L1). Если “Пентиум” записывает в память, для которой нет соответствия в кэше L1, то это вызывает дополнительные накладные расходы на инициализацию кэша. Дополнительная инструкция mov EAX,[EDI+ECX+28+32] загружает кэш-линии для чтения из источника за цикл до операции копирования. При этом загрузка новой кэш-линии идет параллельно с выполнением тела цикла:

```
.REPEAT
mov EAX,[ESI+ECX+7*4]
mov [EDI+ECX+7*4],EAX
mov EAX,[ESI+ECX+6*4]
mov [EDI+ECX+6*4],EAX
```

```
mov EAX,[ESI+ECX+5*4]
mov [EDI+ECX+5*4],EAX
mov EAX,[ESI+ECX+4*4]
mov [EDI+ECX+4*4],EAX
mov EAX,[ESI+ECX+28+32]
mov EAX,[ESI+ECX+3*4]
mov [EDI+ECX+3*4],EAX
mov EAX,[ESI+ECX+2*4]
mov [EDI+ECX+2*4],EAX
mov EAX,[ESI+ECX+1*4]
mov [EDI+ECX+1*4],EAX
mov EAX,[ESI+ECX+0*4]
mov [EDI+ECX+0*4],EAX
add ECX,32
.UNTIL ZERO?            ; 11 тактов + 2 промаха кэша
```

Четвертое улучшение (использование FPU). Для некэшированных данных самым быстрым способом копирования для “Пентиума” является использование устройства с плавающей точкой (FPU). Чтение/запись при этом производится сразу 8 байтами:

```
fldfstp MACRO num,reg1,reg2
fld QWORD PTR [ESI+ECX+num*8]
mov reg2,[ESI+ECX+num*8+8+4]
and reg1,ebp
fstp QWORD PTR [EDI+ECX+num*8]
.IF reg1==EBX
mov reg1,[ESI+ECX+num*8+4]
mov [EDI+ECX+num*8+4],reg1
.ENDIF
ENDM

mov EBP,7ff80000h
mov EBX,7ff80000h
mov EDX,[EDI+ECX+0*8+4]
.REPEAT
  mov AL,[ESI+ECX+63]  ; предварительная загрузка кэша
  fldfstp 0,EDX,EAX         ; 5 тактов
  fldfstp 1,EAX,EDX         ; 5 тактов
```

```
  fldfstp 2,EDX,EAX          ; 5 тактов
  fldfstp 3,EAX,EDX          ; 5 тактов
  add ECX,3                  ; 1 такт
.UNTIL ZERO?                 ; 22 такта
```

Случайные числа

Многие программы используют алгоритм генерации случайных чисел. Скорость получения следующего “случайного” числа часто оказывается “узким местом” в производительности программы. Важной характеристикой алгоритмов генерации случайных чисел является период повторения. Не стоит забывать, что все такие алгоритмы генерируют не чисто случайные числа, а псевдослучайные числа (в книге [18] есть хорошая теоретическая основа для алгоритмов, строящих последовательность псевдослучайных чисел).

Улучшения алгоритмов генерации. Комбинирование двух или более различных по типу генераторов случайных чисел путем операции `xor` дает более “случайный” результат, чем любой генератор по отдельности. Хороший прием при создании генератора – перемешивание выхода. Представим, что есть массив из 256 значений с первыми 256 случайными значениями. Когда нужно получить случайное число, второй генератор выдает случайный индекс (0...255), по которому значение из массива отдается программе. Затем использованное значение в этом элементе массива переписывается первым генератором на новое случайное значение.

Диапазон случайных чисел. Если надо получать случайные числа в диапазоне 0...11, то на первый взгляд вы должны использовать код типа:

```
nNum = ((unsigned long)rand()) % 12;
```

Здесь два нежелательных последствия: 2^{32} не кратно 12. В результате образуется систематическое отклонение. А главное – оператор “%12” транслируется в инструкцию `div`, которая занимает более 42 тактов процессора. Эти неприятности легко устраняются простым приемом:

```
do
{
     nNum = rand() & 0xf;
```

```
} while ( nNum >= 12);
```

Примеры генераторов случайных чисел. Стандартный ANSI C метод:

```
nRandom = ((nSeed = nSeed * 1103515245 + 12345) >> 16 ) &
0x7fff;
```

В результате получаем более-менее хорошую последовательность, но большую часть времени занимает операция умножения.

Простейший генератор:

```
Random PROC NEAR C PUBLIC
mov EAX , nSeed
mov EDX , 1664525
mul EDX
add EDX , 1013904223
mov nSeed , EAX
ret
Random ENDP
```

К сожалению, младшие биты в такой последовательности имеют короткий период повторения.

Сдвиговый генератор:

```
long ShiftRandom()
{
  unsigned long n,bit,temp;

  temp = nSeed;
  for ( n = 0 ; n < 32 ; n ++ )
  {
  bit = ((temp>>0) ^ (temp>>1) ^ (temp>>2) ^
         (temp>>3) ^ (temp>>5) ^ (temp>>7)) & 1;
  temp = (((temp>>1) | (temp<<31)) & -1 ) | bit;
  }
  nSeed = temp;
  return ( temp );
}
```

Алгоритм R250. Практически это наибыстрейший алгоритм, дающий последовательность с большим периодом. Заполняется таблица смещений индексов (приращение на 1 + сброс в 0 для последнего элемента):

```
unsigned long Table[250];
```

```
unsigned long IncrementTable[250];
unsigned long Index1 = 0 ;
unsigned long Index2 = 103;
long Random250()
{
  long ret;
  ret = (Table[Index1] ^= Table[Index2]);
  Index1 = IncrementTable[Index1];
  Index2 = IncrementTable[Index2];
  return ( ret );
}
```

7. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Задачи практикума

Для успешного изучения курса “Архитектура процессоров. Язык ассемблера” вам необходимо выбрать один из пакетов заданий и успешно реализовать его на практике:

- одна курсовая работа + реализация алгоритмов умножения, деления;
- две отдельные задачи на выбор + реализация алгоритмов умножения, деления.

Пакет заданий должен быть реализован так, чтобы задачи можно было продемонстрировать в работе в учебной аудитории. В качестве языка высокого уровня (если он вам нужен) можно выбирать любой вам известный, при выборе компилятора C тоже нет ограничений. Рекомендуется взять среду разработки приложений MS Visual Studio. Компилятор ассемблера: рекомендуется MASM, но также можно пользоваться TASM, WASM и другими, которые вам больше нравятся. Будьте внимательны – при выборе компиляторов, отличных от рекомендуемых, необходимо, чтобы у вас была возможность строить программу из модулей, написанных на разных языках программирования (например, C + ассемблер). Также, возможно, вам потребуется написать код на языке ассемблера с использованием инструкций MMX, не все

компиляторы ассемблера их поддерживают. Вывод изображений во всех задачах можно делать любым способом: напрямую в видеопамять (DOS программа), с использованием Windows GUI, с использованием MS DirectX SDK и др. Тем не менее рекомендуется способ вывода с использованием Windows GUI как наиболее простой и совместимый между компьютерами (аппаратно-независимый). Не рекомендуется выбирать какие-либо продвинутые библиотеки вывода изображений на экран, так как есть опасность “застрять” в их освоении, и тем самым, сорвать выполнение пакета заданий в срок. Однако если вы хорошо знаете такие пакеты и у вас есть практический опыт их использования, то можно их использовать. Главное – возражения типа “А у меня все работает дома, в офисе и так далее” не принимаются. Помните: после окончания зачетной недели задания могут быть усложнены.

Форма отчета

В качестве отчета необходимо представить несколько работающих версий программы, реализующей вашу задачу, и по итогам всех версий составить сравнительную таблицу. Критерием оценки эффективности вашей задачи (процедуры) является среднее количество тактов процессора, потраченных на выполнение этой процедуры и средняя производительность программы в кадрах в секунду (FPS). Результирующая таблица может выглядеть следующим образом(табл. 7.1.).

Таблица 7.1. Форма отчета

Алгоритм/Улучшение/Метод оптимизации	FPS	Такты
Версия 1.0 (Float, C)		
Версия 1.1 (Int, C)		
Версия 1.2 (Int, Asm)		
Версия 1.3 (раскрученный цикл, Asm)		
Версия 1.4 (улучшенное спаривание, Asm)		
Версия 1.5 (вывод двойными словами, Asm)		
Версия 1.6 (инструкции MMX, Asm)		
Версия 1.7 (MMX+раскрученный цикл, Asm)		

Количество версий улучшаемой процедуры должно быть не менее шести, иногда – и более. К некоторым задачам можно применять индивидуальные методы оптимизации алгоритма.

Описание задач

Wolfenstein 3D

Технология рисования псевдо-3D мира методом старой игры Wolfenstein 3D. В каталоге Projects\Wolf Вы найдете работающий пример и исходные тексты. В каталоге Projects\Wolf\Doc есть небольшой документ, иллюстрирующий основные принципы рендеринга использованные в этой технологии. Файл redame.txt содержит полный список подзадач, которые необходимо реализовать для выполнения этого задания-проекта. Вы также можете нарастить этот список подзадач, предложив в результате полноценный программный продукт. Эта задача в полном масштабе доставит вам удовольствие от выполнения полноценного проекта. При этом список подзадач, предложенный в readme.txt – обязательный к выполнению. Перечень обязательных подзадач, которые вам необходимо реализовать на языке ассемблера:

- дописать случай вывода вертикальной линии, когда она не помещается в пределах экрана;
- переписать на языке ассемблера процедуру вывода стен лабиринта.

Перечень дополнительных подзадач, которые необходимо выполнить в рамках курсового проекта:

- избавиться от многочисленных инструкций ветвления в процедуре заливки вертикальных стен;
- объединить процедуры поиска пересечения лучей и стенок вместе с процедурой вывода вертикальной линии;
- сделать mipmap-ы для улучшения качества рендеринга;
- перейти на целые в процедуре определения пересечения луча и стены;
- разработать процедуру заливки горизонтальных полов и потолков;

- сократить площадь заливки черным цветом буфера экрана перед подготовкой кадра. Если реализована процедура заливки пола и потолка, то убрать очистку буфера экрана совсем;
- корректно отталкиваться от стенок, не проходя сквозь них;
- реализовать процедуру вывода карты из отрезков-стен для более удобной навигации в пространстве;
- сделать более удобную реакцию на нажатия клавиш, например, с помощью функции GetKeyboardState (а не сообщения Windows);
- добавить персонажей в виде спрайтов (прямоугольников, параллельных экрану).

Описание деталей реализации проекта Wolfenstein 3D. Мир может быть задан в текстовом файле, что позволяет легко выводить различные сцены (уровни), пользуясь одной и той же программой. Пример описания лабиринта в текстовом файле приведен на рис. 7.1. Здесь цифрами 1, 2, 3, 4 обозначены номера текстур. Эти текстуры находятся в файлах tex01.bmp, tex02.bmp, tex03.bmp, tex04.bmp. Символом “*” обозначена стартовая точка в лабиринте. Камера в начале визуализации располагается в центре клетки лабиринта, в которой поставлена “*”.

```
1111122222222222
1*  1 3        2
1 4 1 3 222222 2
1   1 3 3    2 2
1 4 1 3 3 44 2 2
1   1 3 3 44 2 2
1 4 1 3 3    2 2
1   1 3 3 44 2 2
1 4 1 3 3 44 2 2
1   1 3 3    2 2
1 4 1 3 3 44 2 2
1   1 3 3 44 2 2
1   1        2 2
1   1111111111 2
1              2
1111111111111111
```

Рис. 7.1. Карта лабиринта.

Весь лабиринт может быть представлен как плоская схема, однако при визуализации создается ощущение объема.

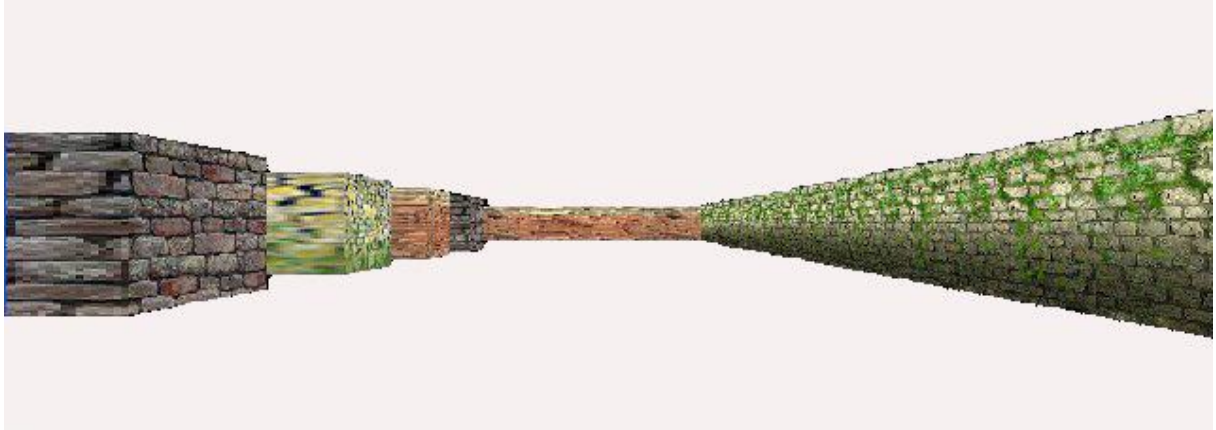

Рис. 7.2. Примерный вид изображения на экране

На рис. 7.2 показан пример вывода на экран программы Wolfenstein. При выводе на экран потолок и пол не изображаются.

Каждая клетка карты лабиринта (см. рис. 7.1) соответствует определенному расстоянию (в метрах) в реальном трехмерном пространстве.

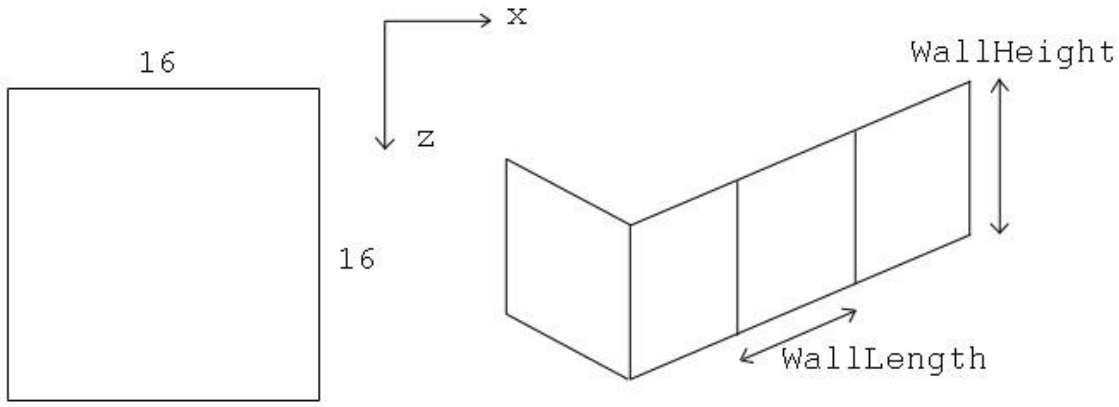

Рис. 7.3. Составные блоки стен лабиринта

На рис. 7.3 показано соответствие клеток стен и карты лабиринта.

Основная идея алгоритма – зарисовка экрана осуществляется вертикальными линиями. Каждая вертикальная линия на экране соответствует некоторой плоскости в пространстве, причем все такие плоскости проходят через точку, в которой в данный момент времени находится камера наблюдателя.

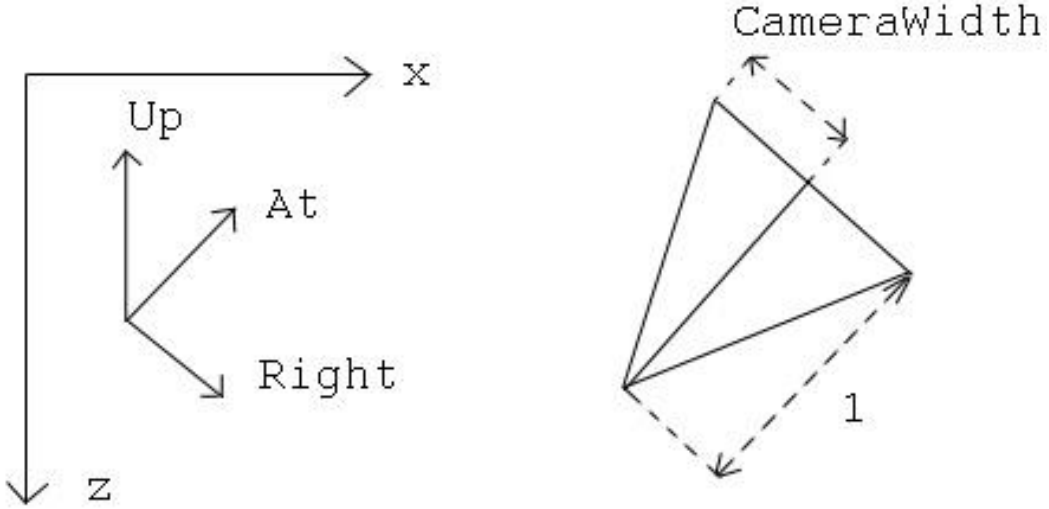

Рис. 7.4. Ориентация камеры наблюдателя в пространстве.

На рис. 7.4 показано, как задается “широкоугольность” камеры. Каждая вертикальная линия на экране соответствует лучу, исходящему из точки, в которой расположена камера в одном из направлений. Луч достаточно рассматривать в 2D плоскости *xOz*, так как наклоны камеры вокруг вектора *At* невозможны. Каждому лучу соответствует параметр *t* в интервале [-0.5; 0.5], луч *N* определяется вектором:

$$N = At + Right \cdot t \cdot CameraWidth$$

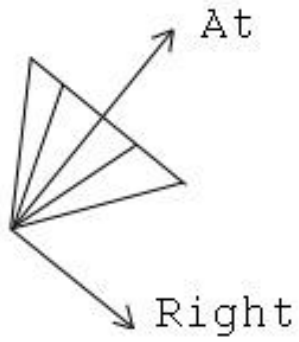

Рис. 7.5. Вертикальные плоскости в конусе видимости камеры.

На рис. 7.5 показаны лучи, соответствующие каждой вертикальной строке экрана. Для каждого луча, соответствующего вертикальной линии экрана, проводится поиск пересечения луча и объектов в сцене. Для простого случая (без персонажей внутри лабиринта) объектами являются только стены лабиринта. Так как стены лабиринта представляют собой регулярную решетку с одинаковым шагом в пространстве между узлами, процедуру поиска пересечения луча и стены можно эффективно реализовать. Для начала определяется индекс квадрата, в котором находится камера, и производится поиск пересечений луча со всеми потенциальными стенками. На этом этапе луч рассматривается в горизонтальной плоскости, в которой задана карта стен. Таким образом, задача поиска пересечения сводится к плоской задаче.

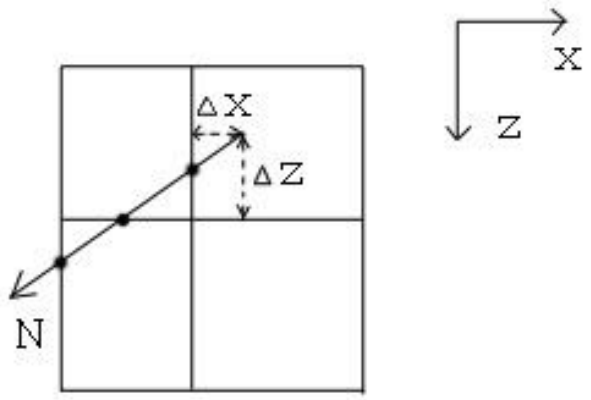

Рис 7.6. Поиск пересечений на карте лабиринта.

На рис. 7.6 показано, что процесс поиска пересечений сводится к поиску “жирных” точек – одну за другой, последовательно, по мере удаления от камеры.

Все пересечения с вертикальными стенками определяются формулой

$z = CamPoz.z + (N.z / N.x)\ \Delta x$

Все пересечения с горизонтальными стенками определяются формулой

$x = CamPoz.x + (N.x / N.z)\ \Delta z$

Проверяют, лежит ли точка пересечения на ребрах квадрата. Если да, то проверяют, есть ли на карте стена в этой клетке. После нахождения

пересечения луча и стены необходимо построить проекцию вертикального отрезка стены на экран.

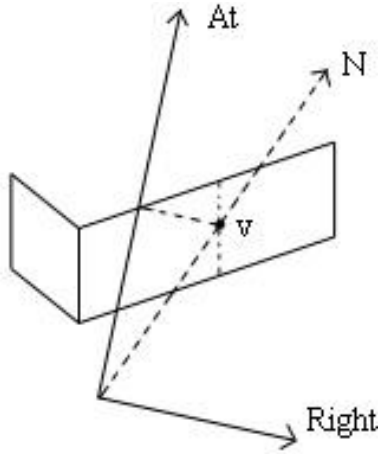

Рис. 7.7. Пересечение луча и стены лабиринта.

На рис. 7.7 показана вертикальная линия стены, которую необходимо вывести на экран. При поиске проекции лучей на экран, необходимо оперировать вершинами, заданными в экранной системе координат. Предположим, что известно пересечение произвольного луча со стеной. Обозначим эту точку v. Тогда v_z в системе координат камеры будет вычисляться как:

z_{cam} = (v – CamPos) CamAt

Осталось выяснить, как найти проекцию линии на экран от произвольного луча из камеры.

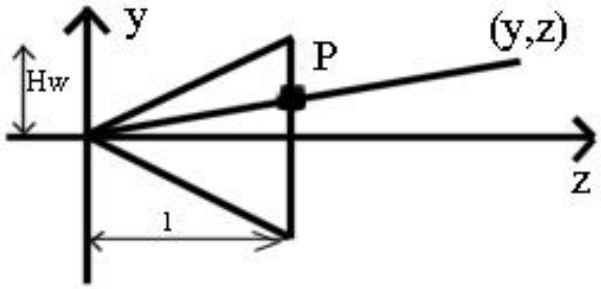

Рис. 7.8. Проекция луча на картинную плоскость камеры.

Камера может быть представлена в виде конуса. Глубина конуса предполагается равной 1. А вот размеры по осям x и y могут быть разными. Этим задается “прямоугольность” камеры и ее “раствор”.

Рассмотрим вертикальную проекцию. Обозначим вертикальный параметр конуса камеры за H_w. Ясно, что при H_w=1 мы получим камеру с углом обзора 90°. Для более “узких” камер этот параметр меньше 1.

Рис. 7.8 иллюстрирует процесс поиска координат точки P. Для произвольной точки с координатами (y, z) в системе координат камеры найдем координаты точки P – это точка проекции луча в конус камеры. Из подобия треугольников получим координату y, Координата z будет равна 1.

$y_p = H_w\ y\ /\ z$

$z_p = 1$

После того, как получены координаты точки P на конусе камеры, или в так называемой картинной плоскости, надо найти координаты точки P на экране. Для этого необходимо отобразить прямоугольник картинной плоскости конуса на экран. Получим

$y_s = (H_s/2) - (H_s/2)y_p$

Здесь y_s – координата y точки P на экране; H_s – высота в пикселях экрана.

Для поиска горизонтальной проекции используются те же выкладки, с заменой на рисунке плоскости (y, z) на плоскость (x, z)

Для нашей конкретной задачи предполагаем, что камера перемещается только горизонтально на высоте y = 0. Высота стены лабиринта фиксирована и равна WallHeight. Стена в мировых координатах расположена по вертикали ровно в центре. Таким образом, в мировой системе координат любая точка на верхнем ребре стены имеет координату WallHeight/2. Так как и камера по координате y находится тоже на нуле, то и в системе координат камеры любая точка верхнего ребра стены тоже будет иметь координату y = WallHeight/2. Таким образом, получаем для верхней точки стены:

$y_s = (H_s/2) - (H_s/2)H_w(\mathrm{WallHeight}/2)\ /\ z$

Для нижней точки получаем симметричное выражение:

$ys = (H_s/2) + (H_s/2)H_w(\mathrm{WallHeight}/2)\ /\ z$

View3d

Просмотр 3D моделей. В каталоге Projects\View3d вы найдете работающий пример и исходные тексты. Эта программа позволяет читать экспортированные из 3D Studio MAX модели в текстовом формате ASE и выводить их на экран.

Список подзадач, обязательных для выполнения:

- все процедуры растеризации треугольника должны быть эффективно реализованы на ассемблере;
- матрично-векторные операции должны быть реализованы на ассемблере;
- должно быть более корректное вращение камеры вокруг модели;
- реализовать более реалистическую растеризацию треугольников модели.

Возможен один из вариантов на ваш выбор:

1) вычислить нормали в вершинах модели и реализовать освещение модели направленным источником, расположенным в бесконечности (диффузное освещение); реализовать процедуру растеризации треугольника по Гуро: на входе у каждой вершины задана яркость или цвет вершины;

2) читать текстурные координаты для вершин модели, читать текстуры из BMP файлов с диска, использованных в модели; реализовать процедуру рендеринга треугольника с текстурой, на входе которой задаются текстурные координаты всех вершин треугольника и указатель на текстуру для этого треугольника; процедура должна учитывать перспективу и корректно интерполировать величины u/z, v/z вдоль скан-линий.

Эта задача – хороший старт для глубокого ознакомления с курсом компьютерной графики.

Рис 7.9. Визуализация трехмерной модели с текстурированием.

На рис. 7.9 показан пример трехмерной модели, выведенной с помощью программы просмотра трехмерных моделей.

Чтение и вывод JPG файлов

Эффективное чтение и вывод JPG файлов. Основная HTML страница курса содержит ссылку на публично доступные исходные тексты библиотеки чтения JPG файлов. Необходимо реализовать на языке ассемблера эффективное дискретное синус-косинус преобразование, и сделать вывод прочитанного битового изображения на экран.

Чтение и проигрывание MP3 файлов

В Интернете есть публично доступные исходные тексты библиотеки чтения звуковых файлов MP3 и их проигрывания. Необходимо реализовать на языке ассемблера эффективное дискретное синус-косинус преобразование и проигрывание пофрагментно музыки из MP3 файла.

Свободный выбор

Вы также можете предложить свои задачи в качестве курсового проекта. Если предложенная вами задача не уступает по сложности предложенным выше, то вы можете выполнить свою задачу.

Отдельные задачи

Ниже приведен большой список задач, в котором вам предлагается выбрать одну из них. В качестве второй задачи необходимо взять пример из Projects\Wolf и прочитать список обязательных задач к этому проекту. Это и есть ваша вторая задача. В тех задачах, которые имеют ссылку на поставляемые примеры решения задач (каталоги Projects\…), использовать предоставленный исходный текст желательно, но необязательно, по вашему усмотрению вы можете по-своему реализовать задачу, необходимо лишь скопировать идею, для чего примеры и предоставляются.

Эффект “смазанного” изображения(Motion Blur)

Этот эффект применяется к изображениям быстро движущихся объектов. Один из возможных подходов: берем прямоугольное изображение и маску прозрачности (дополнительное изображение, в котором 1 значит видимый пиксель, 0 – невидимый) и будем двигать его по экрану. Например, можно взять изображение автомобиля и двигать влево-вправо. Специальный буфер запоминает несколько последних положений изображения на экране (координаты на экране), и после каждого нарисованного кадра буфер обновляется, вытесняя наиболее позднее положение, реализуя таким образом очередь положений.

При отрисовке кадра изображение выводится в соответствии с положениями, запомненными в очереди начиная с наиболее старого, при этом изображение выводится с прозрачностью, уменьшающейся от самого первого (почти прозрачно) до последнего положения (совсем непрозрачно). В последнем (текущем) положении изображение выводится в том виде, в каком оно было бы без эффекта “Motion Blur”.

Освещение изображения точечным источником с эффектом Bump

Каталог Pjojects\Bump содержит работающий пример с исходными текстами. В этом задании необходимо на языке ассемблера реализовать

эффективную процедуру освещения каждого пикселя в результирующем изображении.

Источник задается в пространстве “над” изображением. На входе алгоритма задается цветное изображение, по которому строится изображение в градациях серого, которое используется как карта высот. По этой карте для каждого пикселя изображения вычисляется нормаль, таким образом строится карта нормалей. При выводе изображения на каждом пикселе выполняется следующая операция: строится нормализованный вектор от источника к пикселю изображения, скалярно умножается на нормаль в пикселе, результат умножается на подбираемый коэффициент и результат прибавляется к исходному цвету пикселя.

Закраска прямоугольного изображения по Гуро

Каталог Projects\Fade содержит работающий пример с исходными текстами. В этом задании необходимо реализовать на языке ассемблера процедуру вывода прямоугольного изображения с линейной интерполяцией дополнительного освещения.

Получение карты высот алгоритмом воды и применение карты к изображению

Каталог Projects\Wave содержит работающий пример и исходный текст.

Для каждого пикселя исходного изображения определяется его высота (строится карта высот). Карта высот динамически меняется, при выводе пикселя на экран по карте высот определяется направление наклона поверхности (нормаль) и по этому направлению выбирается пиксель исходного изображения, после чего ставится на экран. Цикл прохода по экрану линейный (подряд все пиксели), при этом пиксели из исходного изображения выбираются нелинейно (для негоризонтальной карты высот).

Начальное состояние карты высот – горизонтальное (с высотой 0), затем программа случайно назначает высоту какой-либо точке карты и далее программа на каждом кадре обновляет содержимое карты высот. Для карты высот запоминается ее состояние на предыдущем кадре. Разность высот для точки карты (частицы воды) считается скоростью движения частицы. Значение разности высоты точки и осредненной высоты соседних

точек, умноженное на подбираемый коэффициент считается силой воздействующей на точку.

Далее для расчета новой высоты применяется формула расчета позиции точки под воздействием силы и скорости с демпфирующим фактором (для того чтобы погасить неустойчивость колебательной модели)

```
F′ = F – V * Dkoef;
V′ = V + F * DeltaT;
H′ = H  + V′ * DeltaT;
```

Эффективная процедура умножения матрицы на массив векторов

Каталог Projects\View3d содержит работающий пример и исходный текст. Модуль Projects\View3d\Src\DiTransf.c содержит процедуру DiMatrixTransformPoints, которая умножает матрицу на массив векторов, реализуя операцию перехода в новую систему координат. Необходимо переписать эту процедуру на ассемблере и выполнить замеры ее эффективности по сравнению с той, что предлагается в примере.

Эффективный вывод треугольников с использованием алгоритма с Z буфером

Каталог Projects\View3d содержит работающий пример и исходный текст. Модуль Projects\View3d\Src\DiRender.c содержит процедуру DiRenderTriangle, которая выводит треугольник с фиксированным цветом с алгоритмом с Z буфером. Необходимо эффективно переписать эту процедуру на ассемблер, используя различные подходы к реализации этого алгоритма и оценить производительность этой процедуры в различных реализациях.

Эффективный вывод треугольников с перспективной коррекцией текстуры

Каталог Projects\View3d содержит работающий пример и исходный текст. Необходимо ввести понятие текстурной координаты для каждой вершины. Вдоль скан-линии интерполируются величины:

`TexU_Per_Z,TexV_Per_Z,Z,`

`TexU_Per_Z = TexU/Z, TexV_Per_Z = TexV/Z.`

На каждом пикселе скан-линии восстанавливаются значения

`u=TexU_Per_Z * Z и v=TexV_Per_Z * Z.`

В соответствии с полученными текстурными координатами, выводится пиксель из исходного изображения.

Вывод кривой Безье рекурсивным алгоритмом с половинным делением

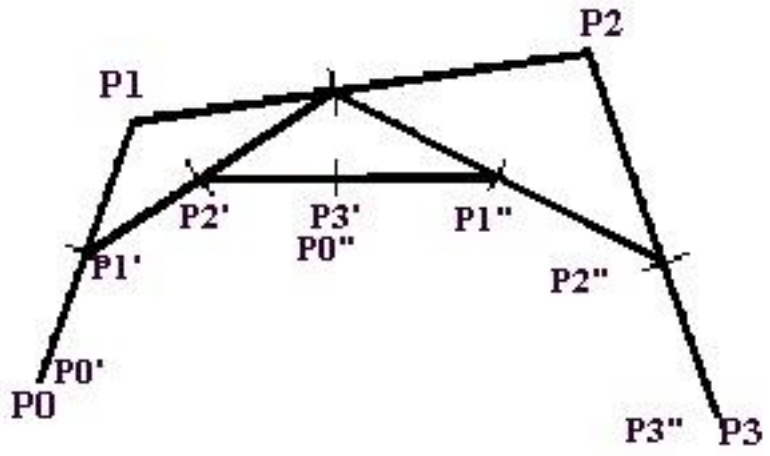

Рис 7.10. Построение кривой Безье рекурсивным алгоритмом.

P0, P1, P2, P3 – исходные базовые вершины. По исходным вершинам рис. 7.10. строятся вершины *P0', P1', P2', P3'* и *P0'', P1'', P2'', P3''*, к которым процедура применяется рекурсивно. При достижении условия окончания рекурсии (вершины почти совпадают) выводится отрезок *P0→P3*.

Алгоритм генерации текстуры огня (алгоритм с фильтрацией и переносом очагов)

Создать цветовую палитру белый–желтый–красный–черный. По размеру изображения создать карту огня. Каждый пиксель карты – яркость горения. Добавлять на каждом кадре в нижнюю строку карты источник горения случайным образом. На каждом кадре сдвигать копированием карту на строку вверх. После сдвига применить усредняющую фильтрацию по восьми соседним точкам. После этого построить изображения, используя точки карты горения как индексы в палитре.

Построение данных и палитры – как в предыдущем описании. На каждом кадре обрабатывается карта по вертикальным столбцам снизу вверх. Каждая точка переносится вверх на следующую строку в одном из трех направлений: строго вверх, вверх и налево или вверх и направо. При этом интенсивность точки случайным образом уменьшается. Эффективный код переноса:

```
nDir = (nSeed&1)-((nSeed&2)>>1);
nSeed = (((UInt32)nSeed)>>2)| (((UInt32)nSeed)<<30);
 (*(cpCur+nDir)) = cValue-(nSeed&7);
```

Искажение изображения в стиле Quake

Построить карту переноса (искажений) для всех пикселей изображения. Карта представляет собой смещение по вертикали, откуда надо брать пиксель. Возможен вариант со смещениями по вертикали и горизонтали. Карта смещений строится с использованием обычных тригонометрических функций (sin, cos) Карта динамически прокручивается по вертикали, создавая иллюзию волнения изображения.

Проявление изображения

Два изображения плавно переходят одно в другое. Цвет на экране вычисляется как

$Cscreen = C_1K_1 + C_2(1.0-K_1)$,

где C_1, C_2 – цвета пикселей первого и второго изображений, K_1 – весовой коэффициент, который анимируется от 0 до 1 попеременно в прямом и обратном направлениях.

Получение уменьшенных вдвое копий изображения

Программа выводит сначала крупное изображение, например, 256×256, затем тут же пересчитывает его уменьшенную копию 128×128, выводит, и т.п. (до 4×4).

Сделать пересчет двумя способами:

- От текущего изображения к уменьшенному вдвое изображению путем осреднения матриц 2×2;
- От исходного изображения к текущему осреднением последовательности увеличивающихся по размеру матриц 2×2, 4×4, 8×8,…

Сравнить производительность получения копии 16×16 из 256×256 обоими методами и качество.

Вывод спрайтов с масштабированием и прозрачностью

На экран вывести спрайт в заданный прямоугольник (меняющий размеры динамически), смешивая выводимые пиксели со статическими пикселями фона динамически меняющимся весовым коэффициентом (0...1).

Быстрая декомпрессия изображения методом RLE (PCX, FLC)

Смотрите также описание на. http://www.dcs.ed.ac.uk/~mxr/gfx/2d/PCX.txt

Формат файла PCX задан в таблице 7.2

Таблица 7.2. Формат файла PCX

Byte offset	Size in bytes	Description
0	1	Constant Flag 10 = ZSoft .PCX

Окончание табл.7.2. Формат файла PCX

Byte offset	Size in bytes	Description
1	1	Version information: 0 = Version 2.5 2 = Version 2.8 w/palette information 3 = Version 2.8 w/o palette information 5 = Version 3.0
2	1	1 = .PCX run length encoding
3	1	Number of bits/pixel per plane
4	8	Picture Dimensions (Xmin, Ymin) - (Xmax - Ymax) in pixels, inclusive
12	4	Horizontal Resolution of creating device
16	48	Color palette setting
64	1	Reserved
65	1	Number of color planes
66	2	Number of bytes per scan line per
68	2	How to interpret palette - 1 = color/BW, 2 = grayscale
70	58	blank to fill out 128 byte header

```
FOR  each  byte,  X,  read from the file
        IF the top two bits of X are  1's then
                count = 6 lowest bits of X
                data = next byte following X
        ELSE
                count = 1
                data = X
```

Упаковка и вывод изображения методом векторной квантизации (VQ)

Метод используется в аппаратной реализации PowerVR. Изображение разбивается на квадраты 2×2. Из всех таких квадратов находят 256 наиболее часто встречающихся (или по другому критерию). Эти 256 квадратов записываются в массив. Исходное изображение заменяется на байтовые индексы в этом массиве (при этом размер изображения

уменьшается вдвое). В итоге получаем таблицу из 256 квадратов + байтовые индексы.

Степень сжатия и качество зависят от исходного размера изображения. Качество также зависит от функции выбора квадратов таблицы-палитры.

Рассчитать скорость распаковки, сравнить с неупакованным изображением, попробовать различные функции, влияющие на качество.

Упаковка и вывод изображения методом DST.

Метод используется в аппаратной реализации S3. Изображение разбивается на элементы 4×4. В каждом таком элементе находится два базовых цвета (RGB). Существует функция, которая по заданным двум цветам строит еще два дополнительных цвета. В простейшем случае это может быть линейная интерполяция. Получаем для каждого элемента четыре цвета. Далее, каждый пиксель элемента 4×4 заменяется на 2-битный индекс в массив из четырех цветов. На выходе получаем последовательность из элементов 4×4, каждый из которых содержит 2 RGB базовых цвета + битовую таблицу индексов размером 4×4×2=32 бита (4 байта).

Степень сжатия и качество не зависят от размера изображения. Качество зависит от функции получения двух дополнительных цветов.

Определить скорость распаковки, сравнить с неупакованным изображением, попробовать различные функции, влияющие на качество.

Искажение изображения.

По формулам:

$x'=x+k_1\sin(k_2*x+k_3)$,

$y'=y+\cos(k_4*y+k_5)$.

Фильтрация изображения.

Формула:

$A'=k_1+k_2|A-D|+|B-C|$, где A, B, C, D – значения цветов в квадрате 2×2, A и D – на главной диагонали.

Простейшие wavelet(ы). Преобразование Хаара.

Сделать программу эффективного преобразования Хаара для изображения.

Прямое преобразование для матрицы 4×4 выполняется по формуле:

$$V' = \frac{1}{4}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & 1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} V$$

Обратное преобразование выполняется, соответственно, обратной матрицей. Сделать прямое преобразование, квантизацию полученных значений (отбрасывание младших N бит) и обратное. Вывести полученные изображения для разных значений N.

Алгоритмы умножения, деления

Алгоритм умножения. Необходимо реализовать побитное умножение, и умножение с какой-нибудь разрядной оптимизацией, например 4, 8, 1, т.е. за одну итерацию умножать не бит на бит, а сразу цепочку битов на цепочку битов (из 4, 8, 6 и т.д.).

Для каждой разрядности сделать сравнение по производительности, обязательную проверку правильности вычисленного результата, оптимизации по памяти и производительности.

Алгоритм деления. Необходимо реализовать несколько целочисленных алгоритмов деления:

- алгоритм с умножением;
- итерационный алгоритм Ньютона–Рафсона;
- простейший алгоритм с вычитанием;
- одноразрядный безвозвратный алгоритм деления “в столбик” (алгоритм CRT).

Для всех алгоритмов провести исследование производительности с одинаковыми исходными данными, определить, при каких исходных данных какой алгоритм более эффективный, проверить правильность полученного результата.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Бредли Д.* Программирование на языке ассемблера для персональной ЭВМ фирмы IBM. М.: Радио и связь, 1988.

2. *Скэнлон Л.* Персональные ЭВМ IBM PC и XT. Программирование на языке ассемблера. М.: Радио и связь, 1991.

3. *Григорьев В.Л.* Микропроцессор i486. Архитектура и программирование: В 4 кн. М.: Микан, 1993.

4. *Пильщиков В.Н.* Программирование на языке ассемблера IBM PC. М.: Диалог-МИФИ, 1997.

5. *Пивоваров В.И.* Ассемблер. Программирование и анализ корректности машинных программ. Киев: Ирина, 2000.

6. *Юров В.* Assembler: Учеб. СПб.: Питер, 2001.

7. *Юров В.* Assembler: Справ. 2-е изд., СПб.: Питер, 2005.

8. *Магда Ю.С.* Assembler для процессоров Intel Pentium. СПб.: Питер, 2006.

9. *Чириков С.В.* Алгоритмы компьютерной графики (Методы растрирования кривых): Учеб. пособие. СПб.: Изд–во СПбИТМО(ТУ), 2001. 120 с.

10. *Bresenham J.E.* Algorithm for Computer Control of Digital Plotter// IBM System Journal. 1965.

11. *Koren I.* Digital Computer Arithmetic. Fast division/ University of Massachusetts. 2004.

12. Atkins D.E. High-Radix Division Using Estimates of the Divisor and Partial Remainders// IEEE Transactions on Computers. 1968.

13. AMD 3DNow! Technology manual/ Advanced Micro Devices (AMD). 1998.

14. ARM Developer Suite. Version 1.2. Assembler Guide/ ARM Limited. 2001.

15. Intel Processor Identification and the CPUID Instruction. Application note/ Intel. 2000.

16. Intel Architecture. MMX Technology. Programmer’s Reference Manual/ Intel. 1996.

17. Vector Floating-point Processing Unit (VFPU). User’s manual/ Sony Computer Entertainment Inc. 2005.

18. *Knuth D.E.* Seminumerical Algorithms. The art of computer programming. Vol.2. Seminumerical algorithms. Reading, Mass// Addison-Wesley, 1968.