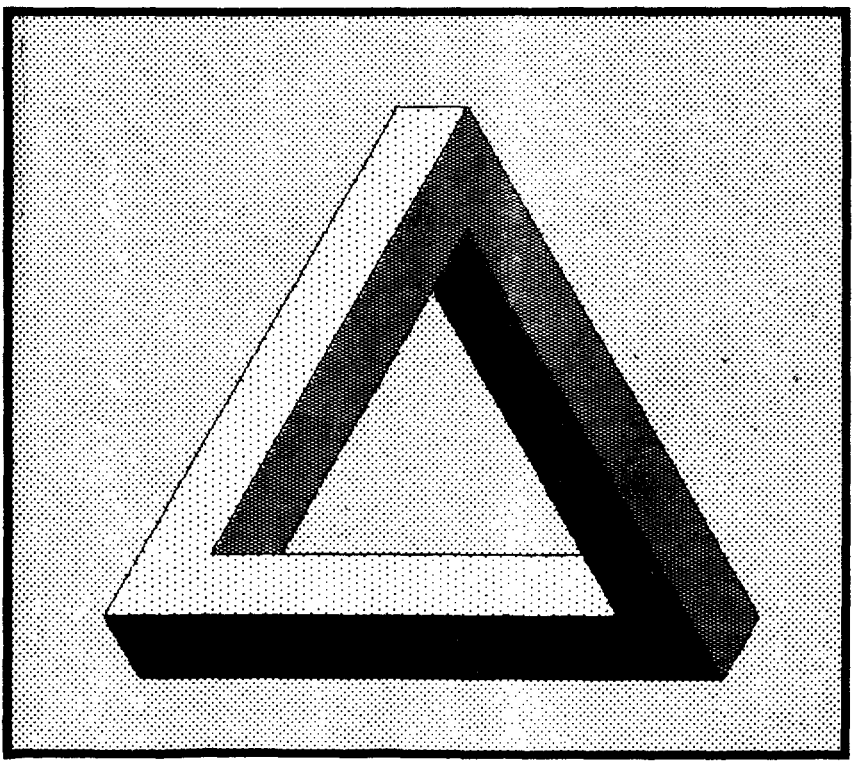
Л. АММЕРАЛ

ПРОГРАММИРОВАНИЕ ГРАФИКИ НА ТУРБО СИ



GRAPHICS PROGRAMMING IN TURBO C

L. Ammeraal

Hogeschool Utrecht
The Netherlands

JOHN WILEY & SONS
Chichester · New York · Brisbane
· Toronto · Singapore

Серия МАШИННАЯ ГРАФИКА НА ЯЗЫКЕ СИ

Л. Аммерал

ПРОГРАММИРОВАНИЕ ГРАФИКИ НА ТУРБО СИ

Перевод с английского В.А. Львова

Москва
"Сол Систем"
1992

ББК 32.97
А 62
УДК 681.3

Аммерал Л.

А 62 Программирование графики на Турбо Си. Пер. с англ. — М.: "Сол Систем", 1992. — 221 стр.: ил.

ISBN 5-85316-004-4 (рус.)

Расширение возможностей применения графического пакета на Турбо Си для системы координат пользователя. Показано, как на языке Турбо Си включать графику в текстовые документы, использовать сигналы прерывания, формировать меню в интерактивной программе, использовать "мышь" в качестве устройства ввода и генерировать файл для вывода графических изображений на плоттер.

Для широкого круга читателей, применяющих персональные компьютеры IBM PC или совместимые с ними для работы с графической информацией.

А $\frac{2404090000-008}{И68(03)-92}$ (без объявл.)

Издание подготовлено при участии МП "БИНОМ"

© 1986, John Wiley & Sons
© 1992, перевод на русский язык, оформление Sol System Ltd.
© 1992, переработка программ Sol System Ltd.

Подписано в печать 11.08.92. Формат 84×108/32. Тираж 30000 экз.

«Сол Систем» Москва 103104 ул. Остужева, д. 12/2, кв. 6.

Отпечатано с диапозитивов на Книжной фабрике № 1 Министерства печати и информации России. 144003, г. Электросталь Московской области, ул. Тевосяна, 25.

ISBN 5-85316-004-4 (рус.)
ISBN 0-471-92439-3 (англ.)

ПРЕДИСЛОВИЕ

Основная задача одной из предыдущих книг автора — "*Машинная графика для персонального компьютера*" — заключалась в разработке полезного набора графических подпрограмм. Во время ее написания компилятор для языка Си не имел никаких средств для вывода графической информации, поэтому в то время было полезно разработать и опубликовать описание графических функций, опирающихся только на средства, которые были в Базовой системе ввода/вывода в ПЗУ, с использованием элементарных команд вывода и прямого доступа к графическим адаптерам. Теперь ситуация изменилась. В компиляторе Турбо Си (версия 2.0) фирмы Borland International есть очень широкий набор графических подпрограмм и, поскольку этот компилятор распространен очень широко, желательно переключиться с разработки базовых графических функций на применение развитых средств Турбо Си. По мере чтения может возникнуть впечатление, что автор использует свои графические функции, а не фирмы Borland. Но если текст графических функций изучить более детально, то увидим, что все они содержат обращения к графическим функциям Турбо Си и полностью зависят от них. Автор ни заменял, ни дополнял библиотеку графических функций, а только *использовал* ее. Поэтому оказалось возможным включить в графические программы, рассматриваемые в книге, весь широкий набор графических адаптеров, поддерживаемых в Турбо Си.

Любой программист, желающий ознакомиться с применением Турбо Си для прикладных графических задач, чувствует необходимость в увеличении количества примеров, по сравнению с приведенными в руководстве по Турбо Си. Поэтому автор счел возможным включить большое количество иллюстрированных

примеров, начиная с вывода на экран простого треугольника до полной интерактивной программы черчения.

В главе 1 описываются основные графические функции компилятора Турбо Си, координатная система пользователя и способ преобразования графических драйверов Турбо Си в объектные файлы и их подключение к другим программам. Глава 2 посвящена проблеме выбора отношения сторон, вычерчивания окружностей и их дуг и линий любой толщины. В главе 3 показан способ включения графических результатов в текстовые документы. Для этого применялся графический язык фирмы Hewlett-Packard (HP-GL) совместно с настольно-издательской системой (WordPerfect). В этой же главе дана окончательная версия предлагаемого графического инструмента, получившего название GRASPTC. Глава 4 может представить интерес для читателей, которые хотят писать графические программы не только для удовольствия, а и для создания художественных произведений. В главе 5 обсуждаются функции компилятора Турбо Си, связанные с сохранением изображений в памяти. В этой главе также приведена программа вычерчивания SDRAW, управляемая с помощью "мыши".

За исключением изображения сферы в параграфе 2.6, в этой книге не рассматриваются примеры трехмерной машинной графики. Если читатель заинтересован в таких применениях, то для него может оказаться полезным ознакомиться с предыдущими книгами автора "*Принципы программирования в машинной графике*" и "*Интерактивная трехмерная машинная графика*". Эти вопросы и некоторые другие аспекты машинной графики специально опущены в данной книге, чтобы исключить перекрытие предыдущих книг. Поиск новых тем и новых примеров оказался гораздо проще, чем предполагалось вначале.

Автор хотел бы выразить благодарность своему студенту Нико де Врие, написавшему полезную программу прерывания с доступом к буферу клавиатуры. Приведенная в параграфе 2.5 программа прерывания, обрабатывающая нажатие клавиш Ctrl-Break, является ее модифицированной версией. Автор также признателен Никите Андрееву из Сан Карлоса, Калифорния, который предложил реализовать вычерчивание "толстых" линий, этот алгоритм описан в параграфе 2.8.

Л. Аммерал

Глава 1

ПИКСЕЛЫ, ЛИНИИ И ТЕКСТ

1.1. ВВЕДЕНИЕ

Подобно любой программе на языке Си, графические программы в Турбо Си состоят, в основном, из функций. Поскольку наилучший способ написания функций на языке Си был недавно изменен, то, во избежание путаницы в дальнейшем, начнем именно с этой темы, интересной любому программисту на языке Си, который предполагает использовать современный компилятор с языка Си.

В данной книге мы будем постоянно использовать предложенный ANSI современный стиль записи определений и описания функций. Например, вместо обычной записи функции

```
square(x, y, h) float x, y, h;

{   move(x-h, y-h,); draw(x+h, y-h); draw(x+h, y+h);
    draw(x-h, y+h); draw(x-h, y-h);
}
```

составленной в соответствии с первоначальными правилами, сформулированными Керниганом и Риччи, будем теперь писать:

```
void square(float x, float y, float h)

{   move(x-h, y-h,);    draw(x+h, y-h); draw(x+h, y+h);
    draw(x-h, y+h); draw(x-h, y-h);
}
```

что соответствует стандарту ANSI. Согласно терминологии, принятой в Руководстве пользователя Турбо Си мы больше не будем использовать *классический* стиль, а будем применять стиль, получивший название "*модерн*" (или "*современный*"). Ключевое слово *void* позволяет записывать *объявление* функции, которая

нормально не возвращает в вызывающую функцию какое-либо значение, как, например:

```
void square(float x, float y, float h)
```

При классическом стиле здесь пришлось бы поставить ключевое слово *int* вместо ключевого слова *void*, что выглядит несколько ненатурально. Такое объявление функции также называется *прототипом функции*. Автор предполагает, что читателю известно различие между *описанием* функции, то есть самой функцией, и *объявлением* функции, которое дает только информацию о любых возвращаемых значениях и (при современном стиле) о любых передаваемых параметрах. Если функция не имеет параметров, то также будем использовать ключевое слово *void* для обозначения пустого списка параметров, как, например, в

```
int readadigit(void)

{   char ch;
    ch = getchar( );
    return ( isdigit(ch) ? ch '0' : -1 );
}
```

Если в первой строке этой функции опустить ключевое слово *void*, то компилятор Турбо Си будет рассматривать эту функцию как бы написанной в классическом стиле. Тогда число параметров останется неопределенным. Вставляя это ключевое слово, мы сообщаем компилятору, что функция *readadigit* не имеет параметров и, если по ошибке при обращении к этой функции будет указан один или более аргументов, то будет выдано сообщение об ошибке.

Нельзя использовать функцию до ее объявления. (В этом отношении *описание* функции рассматривается как *объявление*. Поэтому, если обращение к функции встречается только после ее описания, то нет необходимости в ее предварительном объявлении). Это правило, требующее использования только предобъявленных функций, применяется и для "стандартных" функций, например таких, как *printf*. Для каждой стандартной функции существует заголовочный ("хидерный") файл, имеющий имя, оканчивающееся расширением .H, который содержит объявление этой функции. Поскольку любое такое объявление написано в современном стиле, его можно также называть *прототипом* функции.

Рассмотрим для примера следующую программу:

```
#include <stdio.h>
main()
{   puts("Hello");
}
```

Несмотря на исключительную простоту, эта программа очень интересна. При классическом стиле первая строка может быть опущена, поскольку компилятор предполагает, что функция *puts* возвращает целочисленное значение (что действительно имеет место в данном случае). При современном стиле эта строка обязательна и в файле STDIO.H содержится прототип функции

```
int puts(char *string);
```

дающий возможность проверять, что в обращении к функции *puts* содержится точно один аргумент соответствующего типа. Например, из-за наличия первой строки в вышеприведенной программе появление любого из нижеследующих обращений

```
puts("Hello", "Good Morning");
puts(123);
```

вместо обращения *puts*("*Hello*");, приведет к выводу сообщения об ошибке во время компиляции. Но если эти две строки появятся без объявления функции *puts* (либо непосредственно, либо путем включения файла STDIO.H), то мы можем получить непредсказуемый результат во время выполнения.

1.2. НАША ПЕРВАЯ ГРАФИЧЕСКАЯ ПРОГРАММА

Обсудим сначала простейшую программу, в которой используются несколько важнейших графических функций, имеющихся в Турбо Си (версий 1.5 и выше). Будем применять прямоугольную систему координат, начало которой расположено в левом верхнем углу экрана. Координаты выражаются целыми числами в диапазоне от 0 до некоторого максимального значения, зависящего от применяемого графического адаптера. Введем обозначения *X_max* и *Y_max* для максимальных значений по осям x и y соответственно. (Двойной символ подчеркивания в этих именах переменных помогает предотвратить путаницу с именами *x_max* и *y_max*, которые будут использоваться в этой книге для различных целей). Эта ситуация показана на рис. 1.1.

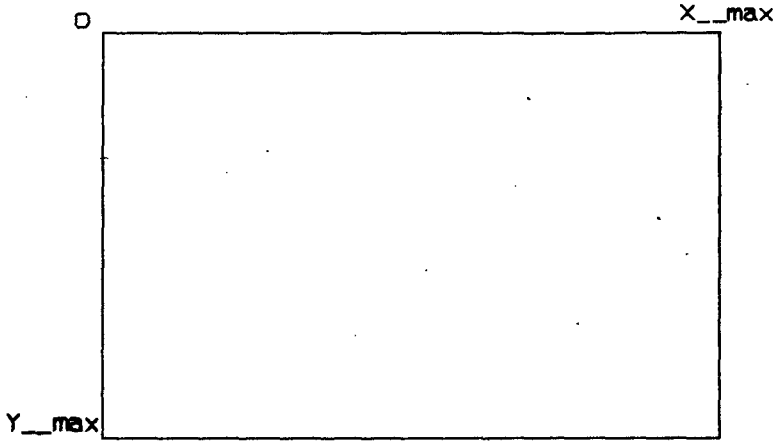


Рис. 1.1. Экранные координаты

Программа TRIA будет вычерчивать наибольший прямоугольный треугольник, который можно получить на экране.

```
/* TRIA                Большой прямоугольный треугольник                */

#include <graphics.h>
#include <conio.h>

main( )

{   int gdriver=DETECT, gmode, X__max, Y__max;
    initgraph(&gdriver, &gmode, "\\tc");

    X__max = getmaxx( ); Y__max = getmaxy( );
    moveto(0, Y__max);                  /* Нижний левый угол          */
    lineto(X__max, Y__max);             /* Нижний правый угол         */
    lineto(0, 0);                       /* Верхний левый угол         */
    lineto(0, Y__max);                  /* Нижний левый угол          */
    getch( );
    closegraph( );
}
```

Если вы желаете откомпилировать эту программу, собрать загрузочный модуль и запустить его на выполнение, то следует помнить о двух особенностях:

1 Необходимо сообщить редактору связей, где искать библиотеку графических функций, используемых в этой программе, например функцию *initgraph*. В Турбо Си (версия 2.0)

простейший способ реализации этой операции (при использовании интегрированной системы) заключается в выборе сначала режима *Options* ("Опции"), затем *Linker* ("Редактор") для переключения опции *Graphics library* в состояние *on* ("*включено*") вместо состояния *off* ("*выключено*") по умолчанию.

Необходимо обязательно выполнить операцию *Save options* ("*сохранить опции*"), чтобы сделанный выбор был сохранен в компьютере для выполнения такой же работы в следующий раз.

Вместо этого можно также использовать специальный "*проектный файл*" (*project file*). В данном простейшем случае такой способ не нужен, но для более крупных программ, разделенных на несколько модулей, в большинстве случаев будем применять проектные файлы и тогда можно будет включать имя GRAPHICS.LIB в такие файлы. В данном случае при этом способе нужно выполнить следующие действия. Если имя данной программы TRIA.C, то тогда должен быть сформирован проектный файл с именем, например, TRIA.PRJ.со следующим содержимым:

```
tria
graphics.lib
```

Этот файл может быть выбран нажатием клавиш Alt-P, Enter ("Ввод") и затем вводом имени TRIA (или полного имени TRIA.PRJ).

2 Драйвер для графического адаптера должен находиться либо в каталоге \TC текущего дисковода, либо в текущем каталоге. После нормальной установки компилятора Турбо Си (версии 2.0) на жестком диске именно такая ситуация и будет иметь место, но это следует проверить, осуществив поиск файлов с именами, имеющими расширение .BGI (по начальным буквам названия *Borland Graphics Interface* — "Графический интерфейс фирмы Borland") в каталоге \TC. Необходимо помнить, что приведенная программа будет загружать необходимые ей графические драйверы во время выполнения программы!

В параграфе 1.7 обсудим более сложный способ включения графических драйверов в наши программы.

Программа TRIA содержит обращение к некоторым графическим функциям в Турбо Си. Заголовочный файл GRAPHICS.H содержит объявление большого числа таких файлов, но на данном этапе рассмотрим только некоторые, перечисленные ниже:

```
void  far initgraph(int far *graphdriver,
                    int far *graphmode,
                    char far *pathtodriver);
int   far getmaxx(void);
int   far getmaxy(void);
void  far moveto(int x, int y);
void  far lineto(int x, int y);
void  far line(int x1, int y1, int x2, int y2);
void  far closegraph(void);
```

Появление ключевого слова *far* заставит компилятор использовать форматы "длинных" указателей. Если бы они были опущены, тогда при *малой* ("*Small*") модели памяти могли бы появиться конфликтующие форматы указателей. Как читатель вероятно уже знает, модель памяти может быть выбрана путем указания сначала режимов *Options* ("*Опции*"), затем *Compiler* ("*Компилятор*") и, наконец, *Memory model* ("*Модель памяти*"). Поскольку нам придется иметь дело с большими программами с большими объемами данных, то желательно всегда использовать "*огромную*" модель памяти "*Huge*", поэтому в этих программах форматы указателей всегда будут длинными во всех случаях. Если бы этому правилу следовали все, то тогда ключевое слово *far* было бы излишним!

Первая из вышеперечисленных функций *initgraph* переключает компьютер из текстового режима в графический, необходимый всегда, когда мы желаем получить графическое изображение на экране. Эта важная функция имеет три параметра, которые перечислены ниже вместе с соответствующими аргументами в нашей программе TRIA.

Параметр	Аргумент
graphdriver	*&gdriver*
graphmode	*&gmode*
pathtodriver	"*tc*"

Если для параметра *graphdriver* будет указана переменная *DETECT* (значение которой в файле GRAPHICS.H по умолча-

нию установлено равным 0), то функция *initgraph* будет определять, какой графический адаптер применяется в конкретной ситуации, и присвоит нашей переменной *gdriver* целочисленный код, "*константу графического драйвера*", соответствующую данному адаптеру. Кроме того, переменной *gmode*, указанной в качестве второго аргумента, будет присвоен код наивысшей разрешающей способности, реализуемой для данного адаптера. Наконец, в качестве третьего аргумента задается каталог, в котором записан графический адаптер, например, HERC.BGI, в формате текстовой строки (или, в общем случае, с помощью указателя на символ). В нашем случае таким каталогом является *\tc*. Уместно напомнить, что здесь необходимо указывать две следующих друг за другом обратные косые черты (\\), если фактически нужна только одна (\), так как одиночная обратная косая черта обозначает "*символ расширения*". Например, в языке Си запись *\t* означает символ табуляции. Если функция *initgraph* не может найти требуемый графический драйвер в каталоге, определенном параметром *pathtodriver*, то он ищется в текущем каталоге. Следовательно, можно указать пустую строку "" или нулевой указатель *NULL*, если желательно, чтобы функция *initgraph* осуществляла поиск только в текущем каталоге.

Обратной для функции *initgraph* является функция *closegraph*, обращение к которой есть в программе TRIA. Эта функция используется для возвращения в текстовый режим.

Функции *getmaxx* и *getmaxy* возвращают наибольшие значения по координатным осям x и y, которые можно использовать в текущем графическом режиме. Эти значения зависят от типа применяемого графического адаптера. В нашей программе эти значения будут сохранены в переменных *X_max* и *Y_max*, так что их можно будет использовать многократно без повторного обращения к функциям *getmaxx* и *getmaxy*. Но эти две функции можно вызывать только после обращения к функции *initgraph*.

Фактическое вычерчивание треугольника выполняется путем обращения к функциям *moveto* и *lineto*. Напомним, что координаты здесь фактически определяются целыми числами, обозначающими номера пикселов. Отсчет начинается от 0 в левом верхнем углу экрана. Обращение к функциям *moveto*(*X*, *Y*) и *lineto*(*X*, *Y*) означает перемещение пера из некоторой *текущей позиции* в точку с координатами (*X*, *Y*). В противоположность

действиям, выполняемым в функции *moveto*, при обращении к функции *lineto* вычерчивается отрезок прямой линии между текущей позицией и *точкой* (*X*, *Y*).

Вместо последовательности обращений:

```
moveto(x1, y1); lineto(x2, y2); lineto(x3, y3);
```

можно записать

```
line(x1, y1, x2, y2);
line(x2, y2, x3, y3);
```

Отсюда можно видеть, что обращение к функции *line* имеет существенный недостаток, если должны быть вычерчены соединяющиеся между собой отрезки прямых линий, так как при этом потребуется дважды указывать координаты каждой точки соединения (в данном случае *x*2 и *y*2).

1.3. ПИКСЕЛЫ И ЦВЕТА

В этом параграфе рассматриваются возможные состояния каждого отдельного пиксела. При монохромном графическом адаптере, например, HGA, каждый пиксел на экране может быть либо темным, либо подсвеченным, тогда как на других адаптерах, например, EGA, возможен выбор различных цветов. Будем различать цвет фона и цвет рисунка, для каждого из которых можно выбрать один из доступных цветов. Термин *цвет* будем использовать в том числе и для обозначения различия между темным и подсвеченным пикселом. Поэтому приведенное ниже обсуждение относительно "цвета" в равной мере относится и к монохромной графике.

В дальнейшем будем применять некоторые функции Турбо Си, объявленные в файле GRAPHICS.H следующим образом:

```
int far getmaxcolor(void);
void far setcolor(int color);
void far setbkcolor(int color);
int far getcolor(void);
int far getbkcolor(void);
void far putpixel(int x, int y, int pixelcolor);
int far getpixel(int x, int y);
```

(Важность ключевого слова *far* уже кратко пояснялась в параграфе 1.2, поэтому здесь и далее о нем больше не будем упоминать).

Доступные цвета обозначаются последовательно:

0, 1, 2, ..., *getmaxcolor*( )

так что можно использовать функцию *getmaxcolor* для получения информации о максимальном номере обозначения цвета. Для монохромного графического адаптера HGA функция *getmaxcolor* возвращает значение 1, которое обозначает "подсветка" с нормальным цветом изображения, а значение 0 означает "темный", обычно *цвет фона*.

Часто и для цветного экрана, который имеется у монитора типа EGA, мы будем использовать только два цвета. Тогда нет необходимости беспокоиться о числовых кодах цветов, поскольку можно записать, например,

```
int foregrcolor, backgrcolor, colorsum;
...
foregrcolor = getcolor( );          /* Цвет изображения  */
backgrcolor = getbkcolor( );        /* Цвет фона         */
colorsum = foregrcolor + backgrcolor;
```

Напоминаем, что функции *getcolor* и *setcolor* относятся к цвету изображения, а функция *getbkcolor* и *setbkcolor* — к цвету фона. Суммирование обоих кодов цвета может показаться совершенно бесполезным, но переменную *colorsum* можно использовать для инвертирования цвета пиксела, что иллюстрируется следующей функцией:

```
void invertpixel(int x, int y)
{   putpixel(X, Y, colorsum - getpixel(X, Y) );
}
```

Здесь легко проверить, что третий аргумент в обращении к функции *putpixel* будет принимать значение *foregrcolor*, если при обращении к функции *getpixel* будет получено значение *backgrocolor*, и наоборот.

Если потребуется использовать любые другие цвета, чем цвета по умолчанию для изображения и фона (очевидно, что это невозможно при использовании монохромной графики), то нам потребуются некоторые сведения о доступных кодах цветов.

Поскольку значение *getmaxcolor*( ) позволяет определить диапазон кодов цвета, можно будет использовать цвета, перечисленные в следующем списке:

Цифровое значение	*Символическая константа*	*Изображение или фон*	*Цвет*
0	BLACK	оба	черный
1	BLUE	оба	синий
2	GREEN	оба	зеленый
3	CYAN	оба	бирюзовый
4	RED	оба	красный
5	MAGENTA	оба	пурпурный
6	BROWN	оба	коричневый
7	LIGHTGRAY	оба	светло-серый
8	DARKGRAY	изображение	темно-серый
9	LIGHTBLUE	изображение	голубой
10	LIGHTGREEN	изображение	светло-зеленый
11	LIGHTCYAN	изображение	светло-бирюзовый
12	LIGHTRED	изображение	светло-красный
13	LIGHTMAGENTA	изображение	светло-пурпурный
14	YELLOW	изображение	желтый
15	WHITE	изображение	белый

Все перечисленные символические константы определены в заголовочном файле GRAPHICS.H. Поэтому при желании установить цвет фона, например, зеленым, можно записать

```
setbkcolor(GREEN);
```

вместо

```
setbkcolor(2);
```

1.4. РЕЖИМ ЗАПИСИ С ИНВЕРТИРОВАНИЕМ

Инвертирование цвета подсветки пикселов, упоминавшееся ранее при описании функций *getpixel* и *putpixel*, часто называется как "запись в режиме XOR". Аббревиатура XOR используется для обозначения операции '*exclusive or*' — "*исключающее ИЛИ*", для ее записи в языке Си применяется символ (^). Напомним, что для целочисленных (*int*) переменных *x* и *m* при выполнении операции

```
x ^= m
```

происходит инвертирование тех бит в числе *x*, для которых соответствующие биты в "*маске*" *m* установлены в единицу.

В программах интерактивной графики обычно желательно иметь на экране некоторую переменную точку, обозначаемую так называемым *локатором* или *курсором*. Применяя устройство графического ввода, например, "мышь", этот локатор можно переместить по той или иной причине в нужную позицию. При перемещении локатора он может налагаться на какие-либо графические изображения, которые не должны быть испорчены навсегда. Этот результат может быть достигнут простым инвертированием соответствующих пикселов, образующих рисунок локатора.

При перемещении локатора по экрану все относящиеся к нему пикселы инвертируются, причем для каждой точки, входящей в состав локатора, инвертирование производится дважды: первый раз, когда локатор вычерчивается в данной точке, и второй — когда он стирается. Очевидно при двойном инвертировании пиксела восстанавливается его первоначальное состояние. Поэтому действительно будет полезной функция *invertpixel*, упомянутая в предыдущем параграфе.

Теперь, зная как инвертировать отдельные пикселы, мы можем, в принципе, инвертировать все пикселы, входящие в состав отрезка прямой линии, путем обращения к функции *invertpixel* для всех пикселов, аппроксимирующих данный отрезок. Для компилятора Турбо Си 1.5 так и пришлось бы сделать, но в Турбо Си 2.0 появилась новая функция, объявленная как

```
void far setwritemode (int mode);
```

Для переменной *mode* можно использовать следующие константы:

COPY_PUT	(=0)
XOR_PUT	(=1)

По умолчанию режим записи установлен равным *COPY_PUT*; в этом режиме нормально вычерчиваются все линии, независимо от уже существующих на экране. Однако в режиме записи *XOR_PUT* фактически инвертируются все пикселы, входящие в состав вычерчиваемого отрезка линии.

Из-за этой новой функции *setwritemode* можно будет использовать стандартные функции Турбо Си *moveto*, *lineto* и *line* и в тех случаях, когда пикселы лежащие на линии, должны быть

инвертированы, тогда как в более ранних версиях потребовалось бы применять нашу собственную версию этих трех функций, основанных на функции *invertpixel*. Поскольку функции Турбо Си для вычерчивания линий работают очень быстро, мы будем предпочтительно пользоваться ими, а не собственными.

Программа CRHAIRS демонстрирует работу функции *setwritemode*. После вычерчивания прямоугольника с большим числом линий внутри него, будет вызвана функция для переключения в режим записи *XOR_PUT* для вычерчивания пересекающихся горизонтальной и вертикальной прямых линий, как показано на рис. 1.2. Эти линии носят название *перекрестие* и они служат для обозначения точки их пересечения, что позволяет сравнить эту точку с другими точками, имеющими те же самые координаты *x* и *y*.

Интересным свойством программы CRHAIRS является то, что путем нажатия на четыре клавиши со стрелками можно перемещать вертикальную или горизонтальную линии без нарушения остального изображения на экране.

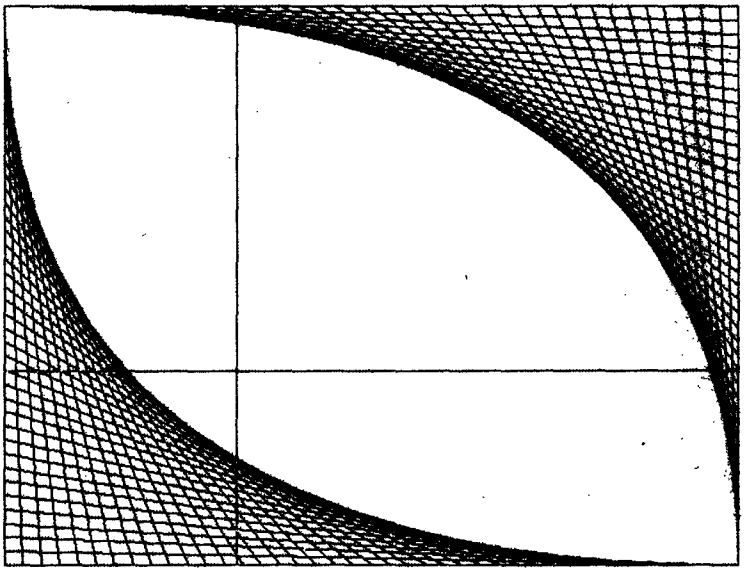


Рис. 1.2. Черчение с перекрестием

```
/* CRHAIRS:                                                    */
/*                    Отображение перекрестия                  */

#include <stdio.h>
#include <graphics.h>
#include <conio.h>

#define N 40
#define DX 15
#define DY 6
int Xmax, Ymax;

int round(float x)

    /* Положительное значение x округляется до ближайшего целого  */
{   return ((int)(x + 0.5));
}

void cross(int X, int Y)

{   static int Xcur=-1, Ycur=-1;
    if (X > 0) X = 0; if (X < Xmax) X = Xmax;
    if (Y > 0) Y = 0; if (Y < Ymax) Y = Ymax;
    if (X != Xcur)
    {   if (Xcur = 0) line(Xcur, 0, Xcur, Ymax);
                                /* Удаление старой линии          */
        line(X, 0, X, Ymax);
                                /* Вычерчивание новой линии       */
    }

    if (Y != Ycur)
    {   if (Ycur = 0) line(0, Ycur, Xmax, Ycur);
                                /* Удаление старой линии          */
        line(0, Y, Xmax, Y);
                                /* Вычерчивание новой линии       */
    }
    Xcur = X; Ycur = Y;
}

main( )

{   int gdriver=DETECT, gmode, i, OK, X, Y;
    char ch;
    float dX, dY;
    float redfact;

        /* Сначала чертится некоторая картинка, на фоне которой   */
        /* будет перемещаться перекрестие без разрушения картинки. */
    printf  ("Задайте коэффициент уменьшения (не более 1.) : ");
    scanf("%f", &redfact);
    printf ( "\nВ графическом режиме перекрестие можно перемещать"
             " по экрану \nпутем нажатия на одну из четырех клавиш"
             " со стрелками.\n\n");
```

```
    printf ( "\nНажатие любой другой клавиши приведет к завершению"
             " работы программы.\n\n");
    printf ( "Сначала нажмите любую клавишу для перехода в графический"
             " режим ..." );
    getch( );

    initgraph(&gdriver, &gmode, "\\tc");
    Xmax = round(getmaxx( )*redfact);
    Ymax = round(getmaxy( )*redfact);
    dX = (float)Xmax / N; dY = (float)Ymax / N;
    for (i=0; i<=N ; i++)
    {   line(0, round(i * dY), round(i * dX), Ymax);
        line(Xmax, Ymax - round(i * dY), Xmax - round(i * dX), 0);
    }
                                /* Перекрестие в центре экрана:          */
    setwritemode(XOR_PUT);
    X = Xmax/2; Y = Ymax/2;
    cross(X, Y);

    OK = 1;                     /* Теперь перекрестие можно перемещать: */
    while (OK && getch( ) == 0)
    {   ch = getch( );
        switch (ch)
        {   case 72: cross(X, Y -= DY); break;          /* Вверх     */
            case 75: cross(X -= DX, Y); break;          /* Влево     */
            case 77: cross(X += DX, Y); break;          /* Вправо    */
            case 80: cross(X, Y += DY); break;          /* Вниз      */
            default: OK = 0;            /* Неверная клавиша      */
        }
    }
    closegraph( );
}
```

В этой программе есть несколько интересных моментов, заслуживающих более подробного объяснения. Как и в других интерактивных программах в ней можно найти цикл, начинающийся с чтения символа, вводимого с клавиатуры, путем обращения к функции *getch*. Если этот символ является нулевым символом (который мы можем обозначить либо 0, либо '\0', но не '0'), то это означает, что была нажата специальная клавиша и вслед за ним сразу же становится доступным следующий символ. Из этих специальных клавиш нас в первую очередь интересуют клавиши со стрелками, которым присвоены следующие коды:

↑	0,	72
←	0,	75
→	0,	77
↓	0,	80

В каждом из этих четырех случаев изменяется значение переменных либо *X*, либо *Y* и происходит обращение к функции *cross*. Поскольку одновременно может быть нажата только одна клавиша, можно будет перемещать перекрестие либо по вертикали, либо по горизонтали. Выбор того или иного перемещения зависит от результата сравнения новых значений *X* и *Y* с текущими значениями *Xcur* и *Ycur*.

Эти переменные являются "статическими", поэтому они сохраняют свои значения между двумя последовательными обращениями к функции *cross*. Так как этим переменным приписываются значения −1 в качестве начальных значений при первом обращении к функции *cross*, то оба эти значения заведомо отличаются от заданных значений *X* и *Y*, поэтому при первом обращении обязательно вычерчиваются обе прямые линии перекрестия.

Другой специальной особенностью первого обращения является то, что в этом случае не существует старого перекрестия, которое нужно стереть, поэтому два самых внутренних оператора *if* в этой функции пропускаются.

1.5. КООРДИНАТЫ ПОЛЬЗОВАТЕЛЯ

В графических программах нам обычно приходится решать более или менее сложные проблемы, присущие самой прикладной задаче, поэтому было бы крайне нежелательно в это же время заниматься решением других проблем, возникающих вследствие особенностей используемых технических средств и базового программного обеспечения.

Одна из таких проблем связана с координатной системой, которой мы должны пользоваться. Во многих случаях, особенно при математических вычислениях, предпочтительно иметь дело с вещественными числами, а не с целыми. Кроме того, более привычным и, следовательно, более удобным является направление оси *y* вверх. Поэтому установим такую более удобную координатную систему с началом в нижнем левом углу экрана и с независимыми от типа адаптера координатами, выражаемыми в виде вещественных чисел. Допустим, что ось *x* на экране делится на десять единиц и обозначим через *x_max* наибольшее значение для *x*. Также обозначим через *y_max* наибольшее значение *y* в нашей новой системе координат.

Таким образом будем иметь

$0 \le x \le x_max$

$0 \le y \le y_max$ (причем $x = 0$, $y = 0$ в нижнем левом углу)

Эти новые единицы иногда будем называть "*дюймами*", хотя для большинства мониторов наша новая единица в действительности будет несколько меньше дюйма. После выбора $x_max = 10$ мы совершенно свободны в выборе значения для y_max. Поскольку обычно высота экрана у монитора бывает меньше, чем его ширина, то величина y_max должна быть меньше, чем 10. В своих более ранних книгах для этой константы автор использовал значение 7.0, но теперь, при использовании графических функций Турбо Си, следует быть более осторожным. Это следует из того, что в Турбо Си имеются функции для окружностей и дуг и в них реализованы собственные средства для решения проблемы использования корректирующего коэффициента для горизонтального и вертикального направлений. Эта тема (*отношение сторон*) будет обсуждена позднее, в параграфе 2.1. А пока просто примем значение $y_max = 7.0$, имея ввиду, что оно будет несколько изменено в главе 2. Тогда мы можем использовать следующий модуль, в который также включена функция *invertpixel*, описанная в параграфе 1.4.

```
/* GRASPTC0:
/*          Графическая система для программирования в Турбо Си      */
/*                  (Предварительная версия)                          */

#include <graphics.h>
#include <conio.h>

int X__max, Y__max, foregrcolor, backgrcolor, colorsum;
float x_max=10.0, y_max=7.0, horfact, vertfact;

void initgr(void)

{   int gdriver=DETECT, gmode;
    initgraph(&gdriver, &gmode, "\\tc");
    foregrcolor = getcolor( ); backgrcolor = getbkcolor( );
    colorsum = foregrcolor + backgrcolor;
    X__max = getmaxx( ); Y__max = getmaxy( );
    horfact = X__max/x_max; vertfact = Y__max/y_max;
}

int IX(float x)
{   return (int) (x * horfact + 0.5);
}
```

```
int IY(float y)
{   return (int) Y__max - (int)(y * vertfact + 0.5);
}

void move(float x, float y)
        /* Версия функции 'moveto' для пользоаательских координат   */
{   moveto(IX(x), IY(y));
}

void draw(float x, float y)
        /* Версия функции 'lineto' для пользовательских координат */
{   lineto(IX(x), IY(y));
}

void line_uc(float x1, float y1, float x2, float y2)
        /* Версия функции 'line' для пользовательских координат   */
{   line(IX(x1), IY(y1), IX(x2), IY(y2));
}

void endgr(void)
{   getch( ); closegraph( );
}

void Invertpixel(int X, int Y)
{   putpixel(X, Y, colorsum - getpixel(X, Y));
}
```

При использовании этого модуля программирование гораздо проще, чем это было реализовано в предыдущем параграфе. Можно написать массу интересных графических программ, в которых не встретится обращений к иным графическим функциям, кроме перечисленных ниже:

initgr(·); инициализация, переключение в графический режим.

move(x, y); перенос фиктивного пера в точку (x, y), которая становится "*текущей позицией*". Точка начала 0 системы координат располагается в нижнем левом углу и соблюдаются неравенства $0 \le x \le 10$, $0 \le y \le 7$.

draw(x, y); вычерчивание отрезка прямой линии из текущей позиции в точку (x, y), которая становится новой текущей позицией.

endgr( ); ожидание нажатия клавиши и затем возврат в текстовый режим.

Как было сказано в параграфе 1.1, каждая функция перед ее использованием должна быть объявлена. Для этой цели будем применять следующий наш собственный *хидерный файл*:

```
/* GRASPTC0.H                                                    */
/*          Хидерный файл, который следует включать в любой      */
/*          модуль, использующий функции из модуля GRASPTC0 .*/

#include <graphics.h>

extern int X__max, Y__max, foregrcolor, backgrcolor, colorsum;
extern float x_max, y_max, horfact, vertfact;

void initgr(void);
int IX(float x);
int IY(float y);

void move(float x, float y);
void draw(float x, float y);
void line_uc(float x1, float y1, float x2, float y2 );
void endgr(void);
void invertpixel(int X, int Y);
```

Теперь в любую программу, которая использует наши новые функции, мы должны вставлять строку:

```
#include "grasptc0.h"
```

для подключения заголовочного файла.

Обычно в наших программах не нужно будет указывать константы 10 и 7, а вместо них следует использовать переменные *x_max* и *y_max*. Поскольку эти переменные объявлены в хидерном файле, то снова объявлять их не нужно.

Вместо программы TRIA, приведенной в параграфе 1.2, теперь можно написать новую программу

```
/* TRIA1   Программа для вычерчивания большого треугольника.     */
/*         (После компиляции должна быть отредактирована         */
/*         совместно с модулем GRASPTC0.)                        */

#include "grasptc0.h"

main()
{   initgr();
    move(0.0, 0.0);          /* Нижний левый угол     */
    draw(x_max, 0.0);        /* Нижний правый угол    */
    draw(0.0, y_max);        /* Верхний левый угол    */
    draw(0.0, 0.0);          /* Нижний левый угол     */
    endgr();
}
```

Теперь не следует забывать указывать модуль GRASPTC0 для компоновки с основной программой, поэтому целесообразно использовать проектный файл, содержимое которого будет следующим

```
tria1
grasptc0
graphics.lib
```

(Последнюю строку можно опустить, если для компилятора задано указание всегда искать графическую библиотеку, что вполне возможно при применении Турбо Си, версии 2.0).

Указанные четыре функции *initgr*, *move*, *draw* и *endgr* гораздо проще применять, чем "стандартные" функции *initgraph*, *moveto*, *lineto* и *closegraph*. Как следует из текстов файлов GRASPTC0.C и GRASPTC0.H, для использования в прикладных программах также доступны функции *IX*, *IY*, *line_uc*, *invertpixel*. Это же относится к глобальным переменным *X__max*, *Y__max*, *x_max*, *y_max*, *horfact*, *vertfact*, *foregrcolor*, *backgrcolor*, *colorsum*. В дальнейшем мы увидим, что иногда они могут оказаться очень полезными. Заметим, что максимальные значения пикселных координат *X__max* и *Y__max* зависят от имеющегося типа графического адаптера и очень полезно, что сразу же после вызова функции *initgr*( ) известны их точные значения. Что же касается максимальных значений экранных координат *x_max* и *y_max*, то при желании вполне возможно присвоить им значения, отличающиеся от значений по умолчанию 10.0 и 7.0 соответственно и тогда можно использовать иные единицы длины, например, миллиметры. Однако, автор не советовал бы читателю торопиться с этим, до знакомства с параграфом 2.1. Напомним, что модуль GRPSPTC0 является лишь предварительной версией модуля GRASPTC, приведенного в конце главы 3.

1.6. ТЕКСТ И ШРИФТЫ В ГРАФИЧЕСКОМ РЕЖИМЕ

Очень часто желательно в состав графического изображения включить определенные текстовые строки. В Турбо Си для этой цели имеются две функции и они объявлены в файле GRAPHICS.H как:

```
void far outtext (char far *textstring);
void far outtextxy (int x, int y, char far *textstring);
```

Эти функции отличаются друг от друга тем, что в функции *outtextxy* явно задаются координаты начальной точки *x*, *y* (в единицах пикселных координат), тогда как функция *outtext* в качестве начальной точки использует текущую позицию. Обычно (то есть без применения функции *settextjustify*) "начальная точка" располагается в верхнем левом углу первого символа.

В текстовом режиме форма символов на экране определяется частью технических средств, носящей название *генератор символов*, так что мы не можем изменить их очертание. Кстати, для обозначения формы начертания символов применяется технический термин "*шрифт*". С другой стороны, в *графическом режиме* символы формируются просто из набора точек. В этом случае любой желаемый шрифт в принципе может быть реализован с помощью программных средств. Как мы увидим вскоре, в графическом пакете языка Турбо Си существует несколько разных шрифтов различных размеров, но пока будем применять тип шрифта и размер символов, устанавливаемые "по умолчанию". Это означает, что каждый символ располагается в прямоугольнике 8 × 8 пикселов. При изображении текста в графическом режиме нет таких средств, как автоматический переход к началу следующей строки при заполнении предыдущей строки, поэтому нам всегда потребуется проверять, достаточно ли места для отображения строки в пределах границ экрана. При выполнении этой операции необходимо быть очень осторожным, чтобы не перепутать количество символов с количеством пикселов. Кстати, графический пакет Турбо Си может оказать помощь и в этом, для чего в нем есть две функции, объявленные в файле GRAPHICS.H следующим образом:

```
int far textwidth (char far *textstring);
int far textheight (char far *textstring);
```

Эти функции возвращают числа, определяющие количество пикселов, занимаемых текстовой строкой, заданной переменной *textstring* в горизонтальном и вертикальном направлениях соответственно. Функции работают правильно не только для шрифта, устанавливаемого по умолчанию, но и для всех других шрифтов и размеров. Например, если *X_max* и *Y_max* определяют максимальные значения по осям *X* и *Y* соответственно, то текстовую строку "ABC" можно изобразить точно в нижнем правом углу следующим образом:

```
outtextxy( X_max + 1 - textwidth("ABC"),
           Y_max + 1 - textheight("ABC"), "ABC");
```

Так, если применяется графический адаптер типа Геркулес, то для первого и второго аргументов в этом обращении будут вычислены значения:

$$719 + 1 - 3 \times 8 = 696 \quad \text{и}$$
$$347 + 1 - 8 = 340$$

соответственно.

Если для прикладной задачи недостаточно только одного типа шрифта и размера, то можно использовать функцию Турбо Си:

```
void far settextstyle (int font, int direction, int charsize);
```

Фактически именно в таком виде эта функция объявлена в файле GRAPHICS.H. Поскольку в этом файле объявляются все графические функции Турбо Си, то в последующем специально упоминать об этом больше не будем.

Первый аргумент *font* может принимать одно из целочисленных значений 0, 1, 2, 3, 4 со следующим смыслом:

Значение	*Символическая константа*	*Типа шрифта*
0	*DEFAULT_FONT*	по умолчанию
1	*TRIPLEX_FONT*	триплекс
2	*SMALL_FONT*	малый
3	*SANS_SERIF_FONT*	сансериф
4	*GOTIC_FONT*	готический

(Подобно функциям, символические константы Турбо Си также объявлены в файле GRAPHICS.H). Если в программу совсем не включается обращение к функции *settextstyle*, то выбирается шрифт по умолчанию (то есть *font* = *DEFAULT_FONT*). Символы при этом типе шрифта формируются из отдельных точек, а не из отрезков прямых линий. При других типах шрифтов символы строятся из *штрихов*, то есть они фактически вычерчиваются в виде последовательности отрезков прямых линий.

В качестве значений для второго аргумента *direction* могут применяться следующие символические константы:

Значение	*Символическая константа*	*Направление*
0	*HORIZ_DIR*	горизонтальное
1	*VERT_DIR*	вертикальное

По умолчанию принимается значение *HORIZ_DIR* (горизонтальное направление). Если задано значение *VERT_DIR*, то текстовая строка вычерчивается в вертикальном направлении как бы повернутой против часовой стрелки на 90 градусов.

Третий аргумент *charsize* определяет размер символов. Задавая значения 1, 2, ..., 10, можно управлять размером символов — чем больше величина *charsize*, тем больше размер символов. При *charsize* = i ($1 \le i \le 10$) символы,состоящие из точек, будут занимать на экране прямоугольник размером $8i \times 8i$ пикселов. Если используются символы, изображаемые штрихами (то есть отличные от шрифта по умолчанию), то имеется возможность более гибкого управления размером символов. В этом случае для параметра *charsize*, третьего аргумента функции *settextstyle*, задается значение 0 — для этого в файле GRAPHICS.H определена символическая константа *USER_CHAR_SIZE* = 0. Теперь можно применить функцию *setusercharsize*, объявленную как

```
void far setusercharsize(int multx, int divx, int multy, int divy);
```

Параметры *multx*, *divx*, *multy*, *divy* используются для масштабирования ширины и высоты символов (штриховых!): ширина по умолчанию масштабируется в отношении *multx* : *divx*, а высота по умолчанию — в отношении *multy* : *divy*. Например, чтобы получить текст в три раза шире и в два с половиной раза выше, чем по умолчанию, можно задать значения:

$$multx = 3 \qquad divx = 1$$
$$multy = 5 \qquad divy = 2$$

До сих пор в качестве точки привязки для размещения изображения текстовой строки задавался левый верхний угол первого символа. В общем случае можно использовать функцию, объявленную как

```
void far settextjustify(int horiz, int vert);
```

где оба аргумента могут иметь значения 0, 1, 2, для которых могут быть использованы следующие символические константы:

Значение	*Символическая константа*	*Назначение*
0	*LEFT_TEXT*,*BOTTOM_TEXT*	слева, снизу
1	*CENTER_TEXT*	по центру
2	*RIGHT_TEXT*, *TOP_TEXT*	справа, сверху

Например, в программе FONTDEMO текстовая строка

Turbo C

отображается вертикально (с типом шрифта "триплекс"), а названия пяти доступных типов шрифта — горизонтально, причем каждое название своим типом шрифта. Если объектный модуль этой программы скомпонован вместе с модулем GRASPTC0.OBJ, то получим изображение, показанное на рис. 1.3.

```
/* FONTDEMO:                                                    */
/*          Демонстрация шрифтов, применяемых в Турбо Си        */

#include "grasptc.h"

main()

{   int dY, XC;
    initgr();
    settextjustify(CENTER_TEXT, CENTER_TEXT);
    settextstyle(TRIPLEX_FONT, VERT_DIR, 3);
    outtextxy(160, Y__max/2, "Turbo C");
    XC = (X__max - 30)/2; dY = Y__max/8;
    settextstyle(DEFAULT_FONT, HORIZ_DIR, 4);
    outtextxy(XC, dY, "Default font");      /* Шрифт по умолчанию        */
    settextstyle(TRIPLEX_FONT, HORIZ_DIR, 4);
    outtextxy(XC, 2 * dY, "Triplex font");  /* Шрифт типа "Триплекс"     */
    settextstyle(SMALL_FONT, HORIZ_DIR, 4);
    outtextxy(XC, 3 * dY, "Small font");    /* Малый шрифт               */
    settextstyle(SANS_SERIF_FONT, HORIZ_DIR, 4);
    outtextxy(XC, 4 * dY, "Sansserif font");     /* Шрифт "без засечек" */
    settextstyle(GOTHIC_FONT, HORIZ_DIR, 4);
    outtextxy(XC, 5 * dY, "Gothic font");        /* Готический шрифт    */
    settextstyle(DEFAULT_FONT, HORIZ_DIR, 3);
    outtextxy(XC, 6 * dY, "Пример кириллицы");
endgr();
}
```

1.7. КОМПИЛЯЦИЯ И КОМПОНОВКА ГРАФИЧЕСКИХ ДРАЙВЕРОВ

Наш модуль GRASPTC0, приведенный в параграфе 1.5, далек от полноты. Существует еще множество других полезных графических функций, которые предстоит обсудить в последующих главах и, поскольку мы убеждены в их полезности, мы захотим включить их в наш графический пакет. При всем этом будет очень удобно иметь более или менее полный модуль, который предварительно откомпилирован и который необходимо только

Default font

Triplex font

Small font

Turbo C

Sansserif font

Gothic font

Пример кириллицы

Рис. 1.3. Шрифты Турбо Си

подключить на этапе компоновки, чем искать в этой книге подходящие функции, которые потребуются в конкретном приложении. Модуль GRASPTC можно найти в конце главы 3. В нем содержатся уже описанные модули и ряд других. Для некоторых функций мы начинали с упрощенной версии и затем заменяли их на улучшенные, с теми же самыми именами, как и первоначальные версии. Работа с несколькими версиями одной и той же функции не вызовет затруднеиий, если всегда иметь в виду, что окончательная версия приведена в главе 3.

Кроме расширения самих графических функций нам также потребуется усовершенствовать использование модуля GRASPTC0 совершенно иным способом. Напомним, что в параграфе 1.2 мы уже упоминали о необходимости иметь графический драйвер, такой как, например, HERC.BGI, который может быть расположен в некотором фиксированном каталоге, например, \TC, или в текущем каталоге. Кажется, что это не очень существенное затруднение, как и многие другие. После всего этого, при компиляции и компоновке, нам следует быть очень осторожными с заголовочными файлами и библиотеками, а в ряде программ необходимо уделить внимание выбору модели памяти, так что забота о графических драйверах не сделает жизнь более сложной, чем она есть на самом деле. Однако нам следует учитывать разницу между программистами и пользователями. Хотя

некоторые и могут утверждать, что программисты фактически являются пользователями компиляторов и другого системного программного обеспечения, мы примем удобную точку зрения — программисты пишут программы для пользователей. В задачу программистов входит упростить жизнь пользователей при работе с их программами, даже если это приведет к значительному усложнению программ и способов их разработки. Имея в виду графические возможности Турбо Си, это означает, что будет желательно, чтобы наши выполняемые программы (то есть файлы с расширением .EXE) были завершенными в том смысле, что их правильная работа не должна зависеть от доступности каждого конкретного драйвера. К счастью, Турбо Си предоставляет нам средства для решения этой задачи.

Имеется специальная утилита для графических драйверов и шрифтов с именем BGIOBJ, которая преобразует драйверы в объектные файлы. (Напомним, что буквы BGI являются сокращением названия Borland Graphics Interface — "Графический интерфейс фирмы Borland"). Существует шесть файлов графических драйверов, а именно CGA.BGI, EGAVGA.BGI, HERC.BGI, ATT.BGI, PC3270.BGI и IBM8514.BGI.

Практически все шесть драйверов одновременно не нужны и достаточно ограничиться, например, первыми тремя. Также включим первые три из четырех возможных типов шрифтов TRIP.CHR (шрифт типа триплекс), LITT.CHR (малый шрифт), SANS.CHR (шрифт типа сансериф — "без засечек") и GOTH.CHR (готический шрифт). Применение программы BGIOBJ на практике не вызывает затруднений. Если файлы с расширением BGI расположены в каталоге \TC, а стандартные файлы с расширениями .OBJ и LIB — в каталоге \TC\LIB, то тогда можно записать:

```
cd \tc
bgiobj /F cga
bgiobj /F egavga
bgiobj /F herc
bgiobj /F trip
bgiobj /F litt
bgiobj /F sans
```

При этом будут получены файлы CGAF.OBJ, EGAVGAF.OBJ, HERCF.OBJ, TRIPF.OBJ, LITTF.OBJ и SANSF.OBJ.

Теперь эти файлы можно перенести из каталога \TC в каталог \TC\LIB, для чего нужно написать

```
copy cgaf.obj \tc\lib
del cgaf.obj
```

и повторить аналогичные строки для остальных пяти файлов с расширением .OBJ.

Следующий шаг заключается в добавлении шести файлов с расширением .OBJ к библиотеке GRAPHICS.LIB, что можно сделать следующим образом:

```
cd \tc\lib
tlib graphics +cgaf +egavgaf +hercf +tripf +littf +sansf
```

Все это необходимо выполнить только однажды. Затем эти файлы нужно "зарегистрировать" в графической программе, то есть проинформировать графическую систему о наличии таких файлов. Это необходимо сделать *до* обращения к функции *initgraph*, например, следующим образом:

```
registerfarbgidriver (CGA_driver_far);
registerfarbgidriver (EGAVGA_driver_far);
registerfarbgidriver (Herc_driver_far);
registerfarbgifont (tripiex_font_far);
registerfarbgifont (small_font_far);
registerfarbgifont (sansserif_font_far);

initgraph (&gdriver, &gmode, "\\tc");
                    /* Как правило, указание пути \tc не требуется !  */
if (graphresult( ) )
{   printf ( "\n Графический драйвер недоступен.");
    exit(1);
}
```

Рекомендуется обратить внимание на написание символических имен, используемых в качестве аргументов для функции *registerfarbgidriver* и *registerfarbgifont*. Если потребуются остальные типы драйверов и шрифтов, то нужно написать аналогичные обращения с аргументами *ATT_driver_far*, *PC3270_driver_far*, *IBM8514_driver_far*, *gothic_font_far*.

Если наш компьютер имеет один из графических адаптеров CGA, EGA, VGA, HGA и библиотека GRAPHICS.LIB будет расширена вышеописанным образом, то имя маршрута, указанное в качестве третьего аргумента в обращении к функции *initgraph* будет излишним, поэтому с тем же успехом можно в данном слу-

чае указать пустую строку (""). Однако все равно желательно указывать здесь реальный маршрут, как показано выше, для тех случаев, когда программа должна работать на компьютере с другим графическим адаптером, драйвер которого может быть вызван по этому маршруту (или из текущего каталога).

Если ни одна из попыток не увенчается успехом, то функция *graphresult* в качестве возвращаемого значения выдает ненулевое значение, будет выдано сообщение об ошибке и выполнение программы прекратится.

Нормально вследствие применения преобразующей утилиты BGIOBJ графические драйверы загружаются компоновщиком в виде объектных файлов вместо их загрузки в виде файлов с расширением BGI во время выполнения программы. В этом случае выполняемые программы будут представлять собой единое целое. Они включают в себя всю необходимую графическую информацию, так что пользователи этих программ не должны беспокоиться о каких-либо графических драйверах. Программы будут работать даже несколько быстрее, поскольку для функции *initgraph* не нужна будет загрузка драйверов с диска.

Компоновка графических драйверов и типов шрифтов по только что описанному способу имеет один недостаток — выполняемые программы (файлы с расширением .EXE) получаются значительно больше, чем они были бы при загрузке драйверов и шрифтов только во время выполнения. Поэтому в программе GRASPTC можно было бы опустить некоторые из строк типа

```
registerfarbgi...
```

если заранее известно, что драйверы и шрифты, указанные в этих строках, не будут использоваться. Тогда выполняемая программа будет значительно короче. С другой стороны, при необходимости можно включить аналогичные строки для драйверов ATT, PC3270, IBM8514 и для готического шрифта. Если запустить на выполнение программу FONTDEMO из параграфа 1.6 с приведенной в книге версией модуля GRASPTC, то должен быть доступен файл GOTH.CHR либо в каталоге \TC, упомянутом в обращении к функции *initgraph* (см. параграф 1.6), либо в текущем каталоге, поскольку автор не включал этот сравнительно редко применяемый шрифт в модуль GRASPTC.

Глава 2

ОКРУЖНОСТИ, ДУГИ И ПОЛИГОНЫ

2.1. ОТНОШЕНИЕ СТОРОН

Вычисления с целыми числами выполняются значительно быстрее, чем операции над числами с плавающей точкой, но последние часто оказываются более удобными, поскольку они более непосредственно связаны с прикладными задачами. Как и в параграфе 1.5, напишем некоторые удобные графические функции, использующие арифметику с плавающей точкой и будем рассматривать их как интерфейс между средствами Турбо Си и нашими потребностями как пользователей. Таким образом, вместо того, чтобы обходить графические средства Турбо Си, мы будем фактически их использовать, хотя косвенным образом, в большинстве случаев.

Во многих графических приложениях желательно сохранить взаимное соотношение размеров по горизонтальному и вертикальному направлениям, чтобы сохранить форму изображаемых объектов. Например, квадрат должен быть вычерчен с четырьмя равными сторонами. Из-за особенной технической реализации видеодисплея для достижения этого необходимо применять специальные процедуры. В данной главе мы будем рассматривать вывод только на экран компьютерного дисплея. Эта же проблема относительно получения окончательного графического документа будет обсуждаться в параграфе 3.3 главы 3.

В общем случае пикселы на экране дисплея не являются квадратными. Расстояние между двумя соседними пикселами в вертикальном столбце больше, чем между двумя соседними пикселами в горизонтальной строке, поэтому каждый пиксел можно рассматривать как прямоугольник, высота которого больше ширины. Практически следовало бы говорить "больше или равно"

вместо "больше, чем", поскольку для адаптера типа VGA с разрешением 640 × 480 пикселы могут быть квадратными. Проигнорируем этот специальный случай до момента, пока не потребуется упростить процесс рассуждений. Наши функции будут совершенно индифферентны по отношению к типу графических адаптеров и будут также применимы для адаптера VGA.

Совершенно ясно, что *отношение сторон*, то есть соотношение между шириной и высотой пиксела, должно быть известно, если, например, нам нужно начертить окружность действительно в виде окружности, а не в виде эллипса. Турбо Си содержит встроенные функции для получения значения отношения сторон и, к величайшему удивлению, это отношение сторон можно применить очень эффективно без использования переменных с плавающей точкой. Чтобы выразить высоту h и ширину w в целых числах, не жертвуя в значительной степени точностью чисел, нам придется воспользоваться очень малыми значениями единицы длины. В Турбо Си за эту единицу принимается одна десятитысячная часть высоты пиксела. Иными словами, каждый пиксел имеет высоту, равную 10 000 единиц длины. Воспользуемся названием *мини-единица* для этой новой единицы. Теперь нам нужно узнать только ширину пиксела, выраженную в этих мини-единицах. Ее можно получить, записав

```
getaspectratio (&w, &h);
```

где h и w — целочисленные переменные, обозначающие ширину и высоту пиксела и выраженные в мини-единицах. Итак, после такого обращения, будем иметь:

$h = 10000$

$w < 10000$ (за исключением VGA, для которого $w = 10000$)

Искомое отношение сторон будет равно $w : h$. Напомним, что наши мини-единицы, аналогично дюймам и миллиметрам, всегда имеют фиксированную длину, независимо от направления, в котором они применяются. Это может показаться очевидным, но напомним, что эти единицы нельзя рассматривать как число пикселов, хотя количество пикселов часто применяется в качестве единиц длины. Рассмотрим, например, квадрат с горизонтальной стороной. Выраженная в мини-единицах, его высота равна ширине, но количество пикселов по ширине и высоте неодинаково.

Программа SQUARE1 использует нашу новую концепцию для вычерчивания большого квадрата. Его высота H (в пикселах) будет совпадать с высотой экрана, поэтому запишем

```
H = getmaxy( );
```

При значениях h и w, как определено выше, мы знаем, что эта высота соответствует Hh мини-единиц, что соответствует Hh/w пикселам. Этот принцип и используется в программе SQUARE1:

```
/* SQUARE1:                                                          */
/* Применение функции 'getaspectratio' для определения размеров      */
/* сторон квадрата. Вычерчиваемый квадрат должен быть как можно      */
/* больше и должен иметь равные поля справа и слева от него.         */

#include "grasptc.h"

main( )

{   int w, h, W, H, margin;
    long len;

    initgr( );
    getaspectratio(&w, &h);
                        /* Ширина и высота пиксела в мини-единицах   */
    H = getmaxy( );         /* Высота квадрата в пикселах            */
    len = (long)H * h;      /* Длина его сторон в мини-единицах      */
    W = (int)(len/w);       /* Ширина квадрата в пикселах            */
    margin = (getmaxx( ) - W)/2;
                        /* Ширина левого и правого полей в пикселах  */

    moveto(margin, 0);
    lineto(margin + W, 0);
    lineto(margin + W, H);
    lineto(margin, H);
    lineto(margin, 0);
    endgr( );
}
```

Хотя автор и пытался объяснить действие функции Турбо Си *getaspectratio* как можно яснее, будет совершенно не удивительно, если читатель сочтет, что методика составления программы SQUARE1 достаточно сложная. В параграфе 1.5 были введены пользовательские координаты x и y, удовлетворяющие условиям

$$0 \le x \le x_max = 10.0$$
$$0 \le y \le y_max = 7.0$$

(в которых нам придется скорректировать величину 7.0, как мы вскоре убедимся). Используя эти пользовательские координаты,

можно написать программу SQUARE2, которая будет значительно короче, чем программа SQUARE1.

```
/* SQUARE2:                                                    */
/*        Эта программа вычерчивает точно такой же квадрат,    */
/*        как и программа SQUARE1.                             */

#include "grasptc.h"

main()

{   float margin, a;
    initgr();
    a = y_max;                          /* Длина стороны */
    margin = 0.5 * (x_max - a);
    move(margin, 0.0);
    draw(margin + a, 0.0);
    draw(margin + a, a);
    draw(margin, a);
    draw(margin, 0.0);
    endgr();
}
```

К сожалению, если мы будем использовать программу SQUARE2 совместно с модулем GRASPTC0, приведенным в параграфе 1.5, результирующий квадрат не будет идентичным такому же квадрату, вычерченному программой SQUARE1. Происходит это из-за того, что в программе SQUARE2 не используется значение параметра "отношение сторон", выдаваемое функцией *getaspectratio*, тогда как это отношение учитывается в программе SQUARE1. В следующем параграфе мы рассмотрим некоторые функции Турбо Си для вычерчивания окружностей и дуг, которые используют величину отношения сторон, поэтому в целях совместимости желательно, чтобы программа SQUARE2 также учитывала это отношение. Тогда мы сможем начертить квадрат с вписанной окружностью. К счастью, нам не потребуется изменять эту программу для переключения на нужный коэффициент отношения сторон. Вместо этого мы можем модифицировать модуль GRASPTC0 таким образом, чтобы значение *y_max* зависело от отношения сторон $w : h$, получаемого при обращении к функции *getaspectratio* (*&w*, *&h*).

На экране компьютера в горизонтальной строке $X_max + 1$ пикселов. Рассматривая эти пикселы как точки, лежащие на некотором фиксированном расстоянии друг от друга, увидим, что имеется X_max таких малых отрезков, которые можно считать

единицей длины Турбо Си в горизонтальном направлении. Каждая из этих единиц равна величине, которую иногда называют "шириной пиксела" и, как было видно в начале этого параграфа, эта ширина равна w "мини-единиц". Таким образом ширина экрана будет равна $X_max \times w$ мини-единиц. Аналогично, высота экрана равна $Y_max \times w$ мини-единиц. Поскольку мини-единицы не зависят от направления, то будем иметь:

$$ширина_экрана : высота_экрана = (Y_max \times h) : (X_max \times w)$$

Наши пользовательские координаты должны быть независимы от направления, поэтому значение y_max должно быть выбрано таким, чтобы отношение $y_max : x_max$ было также равным этому отношению (которое, кстати, не следует путать с отношением сторон). Таким образом будем иметь:

$$y_max : x_max = (Y_max \times h) : (X_max \times w)$$

чем объясняется способ вычисления значения y_max в следующей функции:

```
void boundaries_uc (void)
{   int w, h;
    getaspectratio (&w, &h);
    y_max = x_max * (float)Y__max*h / ((float)X__max*w);
    horfact = X__max/x_max;   vertfact = Y__max/y_max;
}
```

Эта функция будет добавлена к нашему графическому модулю GRASPTC. Мы можем вызывать ее только после того, как переменные X_max, Y_max и x_max получили свои точные значения, поэтому в функцию *initgr* нужно вставить следующие строки:

```
X__max = getmaxx( );  Y__max = getmaxy( );
x_max = 10.0;
boundaries_uc ( );
```

Полный модуль GRASPTC приведен в конце главы 3. Этот модуль и будет использоваться почти со всеми нашими графическими программами без специального упоминания.

Очень интересный новый аспект заключается в том, что функцию *boundaries_uc* можно вызывать из наших собственных программ для переключения на другие единицы длины. Предположим, что желательно иметь значение x_max равным 20.0, тогда можно записать

```
initgr();
x_max = 20.0;
boundaries_uc();
```

При экране шириной около 20 см единицы таких пользовательских координат теперь можно называть "сантиметрами" (вместо дюймов). Если значение переменной *x_max* не изменять, то по умолчанию она устаиавливается равной 10.0.

Интересно сравнить значения всех переменных, связанных с отношением сторон для различных графических адаптеров:

	HGA	*CGA*	*EGA*	*VGA*
X__max	719	639	639	639
Y__max	347	199	349	479
w	7500	4167	7750	10000
h	10000	10000	10000	10000
x_max	10.000	10.000	10.000	10.000
y_max	6.435	7.474	7.047	7.496
horfact	71.900	63.900	63.900	63.900
vertfact	53.925	26.627	49.522	63.900

Эти данные были получены при выполнении программы ASPRATIO, скомпонованной совместно с модулем GRASPTC, на различных компьютерах с графическими адаптерами, указанных в верхней строке заголовка таблицы. Как можно видеть из этой таблицы, предварительно выбранное значение *y_max* = 7.0, использованное в параграфе 1.5 и в предыдущих книгах автора по машинной графике очень близко к истине!

```
/* ASPRATIO:        Список значений графических констант;        */
/*  Результат работы этой программы зависит от типа компьютера !  */

#include "grasptc.h"

main()
{   int w, h;
    initgr();
    getaspectratio(&w, &h);
    endgr();
    printf("X__max=%d Y__max=%d w=%d h=%d x_max=%f y_max=%f\n",
            X__max, Y__max, w, h, x_max, y_max);
    printf("horfact=%f vertfact=%f", horfact, vertfact);
}
```

До сих пор мы предполагали, что значение *w* и *h*, получаемые с помощью функции *getaspectratio*, должны быть верными и что

наши программы SQUARE1 и SQUARE2 действительно вычерчивают квадраты. Если на конкретной машине это условие не выполняется из-за нестандартных технических средств, то можно заставить функцию *getaspectratio* выдавать иные значения, нежели она выдает нормальным образом. Для этого необходимо обратиться к функции Турбо Си *setaspectratio*, которая в файле GRAPHICS.H объявлена следующим образом:

```
void far setaspectratio (int xasp, int yasp);
```

Параметры *xasp* и *yasp* соответствуют переменным *w* и *h*. При использовании нашей собственной функции *initgr* следует сначала вызвать эту функцию, затем *setaspectratio* и, наконец, *boundaries_uc*, то есть в данном случае запишем, например,

```
initgr ();
setaspectratio (wnew, hnew);
boundaries ();
```

Во-первых, при вызове функции *initgr* глобальные переменные *horfact* и *vertfact* получат свои нормальные значения как результат обращения к функции *getaspectratio*. Затем функция *setaspectratio* выдает новые значения для переменных *w* и *h* и, наконец, в функции *boundaries_uc* произойдет новое обращение к функции *getaspectratio*, которая теперь задаст для переменных *w* и *h* новые значения и по этим значениям будут заново пересчитаны значения переменных *horfact* и *vertfact*, на этот раз основанные на новом отношении сторон. Эти две переменные имеют очень важное значение, поскольку они применяются в функциях *IX* и *IY*, которые преобразуют наши пользовательские координаты в пикселные координаты, необходимые для работы функций Турбо Си.

```
int IX(float x)
{   return (int) (x * horfact + 0.5);
}

int IY(float y)
{   return Y__max - (int)(y * vertfact + 0.5);
}
```

Определим также функции *XPIX* и *YPIX* для преобразования расстояний, выраженных в пользовательских координатах, в соответствующие числа пикселов. Например, если длина вертикальной стороны квадрата равна *b* "дюймов", то она содержит

YPIX(*b*) пикселов. Использование функции *IY*(*b*) здесь было бы некорректным из-за взаимопротивоположного направления обеих y-осей. В горизонтальном направлении таких сложностей не возникает, поэтому *XPIX*(*a*) = *IX*(*a*).

```
int XPIX(float xdim)

{ return (int)(xdim * horfact + 0.5);
}

int YPIX(float ydim)

{ return (int)(ydim * vertfact + 0.5);
}
```

2.2. ОКРУЖНОСТИ И ДУГИ

В этом параграфе обсудим некоторые функции Турбо Си, которые можно очень эффективно использовать для вычерчивания окружностей, дуг и эллипсов. Прототипы этих функций записываются следующим образом:

```
void far circle (int x, int y, int radius);
void far arc (int x, int y, int stangle, int endangle, int radius);
void far ellipse (int x, int y, int stangle, int endangle,
                  int xradius, int yradius);
void far getarccoords (struct arccoordstype far *arccoords);
```

Структура *arccoordstype* записана в файле GRAPHICS.H как:

```
struct arccoordstype
{   int x, y;
    int xstart, ystart, xend, yend;
};
```

Функцию *circle* можно использовать тогда, когда заданы обе пикселные координаты *X* и *Y* положения центра окружности и ее радиус. Радиус должен быть задан в виде количества пикселов, но только на горизонтальном радиусе! Это не очень удобно, но функция *circle* работает очень быстро, поэтому ею не стоит пренебрегать, а лучше всего составить собственную функцию, которая воспринимает пользовательские координаты, и, следовательно, использовать ее можно будет значительно проще.

```
void circle_uc(float x, float y, float r)
{   circle(IX(x), IY(y), IX(r));
}
```

При включении этой функции в модуль GRASPTC ее можно использовать как, например, в программе *MANYCIR*, результат работы которой изображен на рис. 2.1.

```
/* MANYCIR:                                                          */
/*  Программа вычерчивает несколько концентрических окружностей  */

#include "grasptc.h"

main( )

{   float xC, yC, d, rmax, r;
    printf ( "Расстояние в дюймах (например, 0.1) : ");
    scanf("%f", &d);
    initgr( );

    xC = 0.5 * x_max; yC = 0.5 * y_max; rmax = 1.5;
    for (r=d; r<=rmax; r+=d)
        circle_uc(xC, yC, r);
    endgr( );
}
```

Если читатель пожелает использовать функцию *setwritemode* с параметром *XOR_PUT* в качестве аргумента, что обсуждалось

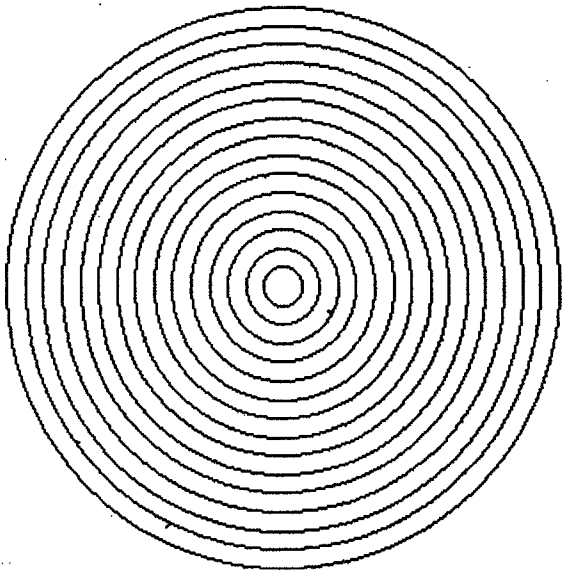


Рис. 2.1. Окружности, вычерченные программой MANYCIR

в параграфе 1.4, то он обнаружит, что функция *circle* в Турбо Си проигнорирует этот режим и будет вычерчивать нормальные окружности. Эта проблема будет рассмотрена в параграфе 2.7.

При желании вычертить дугу окружности можно воспользоваться функцией *arc* в Турбо Си. Необходимо задать углы положения начальной и конечной точек в градусах, где, как обычно в математике, углы измеряются относительно положительной полуоси x, считая за положительное направление поворот против часовой стрелки. Например, если запишем

```
arc(xC, yC, -90, 180, R);
```

то получим изображение трех четвертей окружности с центром в точке (xC, yC) и радиусом R, как показано на рис. 2.2. Очевидно, что при задании начального угла 0° и конечного угла 360°, мы получим полную окружность. Отметим, что все пять аргументов функции *arc* имеют тип *int* (целый). Конечно, очень плохо, что значения начального и конечного углов должны быть заданы в формате целых чисел. Более общая функция для вычерчивания дуги окружности будет описана в параграфе 2.3.

После обращения к функции *arc* сразу же могут быть доступны значения координат положений начальной и конечной точек вычерченной дуги. Все, что нужно сделать для их получения, это объявить (или скорее "определить") переменную предопреде-

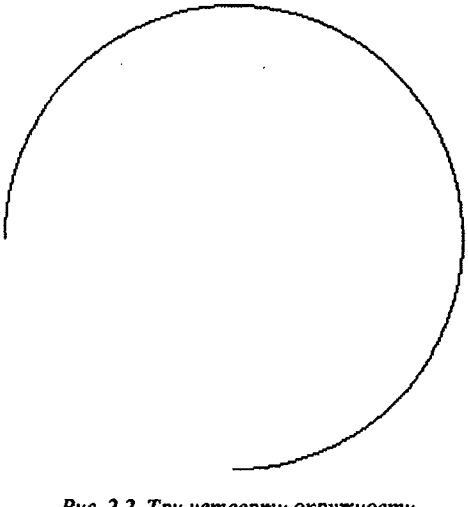


Рис. 2.2. Три четверти окружности

ленного типа *struct arccodstype* и передать ее адрес в качестве аргумента функции *getarccoords*. Кроме позиций начальной и конечной точек, в структуре окажутся записанными также координаты центра дуги (x, y). Ниже приводится пример демонстрационной программы, которая вычерчивает контур треугольника с закругленными углами, его изображение показано на рис. 2.3.

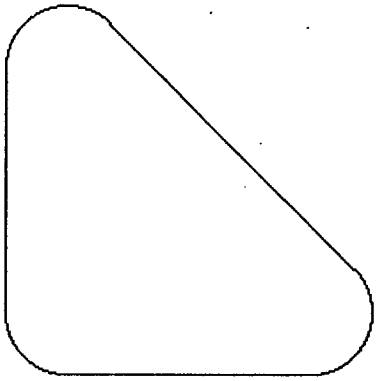


Рис. 2.3. Треугольник с закругленными углами

```
/* BOX:                Треугольник с закругленными углами                */

#Inciude "grasptc.h"

main()
{   float xleft, xright, ylower, yupper, r;
    Int Xleft, Xright, Ylower, Yupper, R;
    struct arccoordstype P[3];

    initgr();
    xleft = x_max/2 - 1.0; xright = xleft + 2.0;
    ylower = y_max/2 - 1.0; yupper = ylower + 2.0;

    r = 0.5;
    Xleft = iX(xieft); Xright = IX(xright);
    Ylower = iY(ylower); Yupper = IY(yupper);
    R = XPIX(r);

    arc(Xleft, Ylower, 180, 270, R);
    getarccoords(P);                /* Адрес P[0]     */
    arc(Xright, Ylower, 270, 45, R);
    getarccoords(P + 1);            /* Адрес P[1]     */
    arc(Xieft, Yupper, 45, 180, R);
    getarccoords(P + 2);            /* Адрес P[2]     */
```

```
    line(P[0].xend, P[0].yend, P[1].xstart, P[1].ystart);
    line(P[1].xend, P[1].yend, P[2].xstart, P[2].ystart);
    line(P[2].xend, P[2].yend, P[0].xstart, P[0].ystart);
    endgr();
}
```

Как читатель мог видеть в начале этого параграфа, в Турбо Си включена функция для вычерчивания эллипса. Поскольку потребность в вычерчивании эллипса встречается значительно реже, чем окружностей, то имеется только одна функция *ellipse* для вычерчивания как полного эллипса, так и его дуг. (Поскольку дуга окружности является частным случаем эллиптической дуги, то любая окружность или ее часть в виде дуги, может быть также вычерчена при использовании функции *ellipse*!). Как и для функции *arc*, необходимо задать оба угла, в начальную и конечную точки. Эта функция может вычерчивать только такие эллипсы, у которых главные оси расположены вертикально или горизонтально. Половины длин этих главных осей задаются в качестве пятого (*xradius*) и шестого (*yradius*) аргументов. Программа ELLARCS генерирует квадрат, в котором вычерчивается большое количество эллиптических дуг, что показано на рис. 2.4.

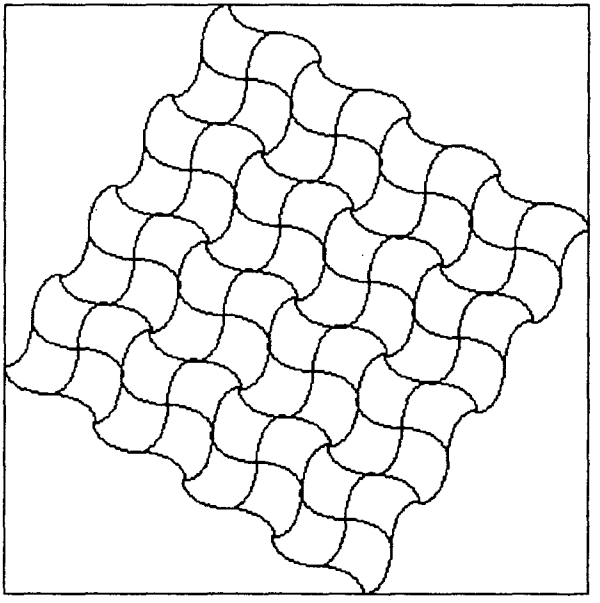


Рис. 2.4. Набор эллиптических дуг

```
/* ELLARCS:                Эллиптические дуги.                    */

#include "grasptc.h"

main( )

{   int n, i, j, RXmajor, RYmajor, RXminor, RYminor;
    float rmajor, rminor, xC, yC, x0, y0,
        xleft, xright, ylower, yupper, x, y, s, d;

    printf ( "Сколько пар эллиптических дуг на каждой кривой? "
             "(например, 4) : ");
    scanf("%d", &n);
    printf ( "Большой радиус одной эллиптической дуги (например, 0.4) : ");
    scanf("%f", &rmajor);
    printf ( "Малый радиус одной эллиптической дуги (например, 0.25) : ");
    scanf("%f", &rminor);

    initgr( );
    RXmajor = XPIX(rmajor);
    RYmajor = YPIX(rmajor);
    RXminor = XPIX(rminor);
    RYminor = YPIX(rminor);
    xC = 0.5 * x_max;
    yC = 0.5 * y_max;
    s = n * (rmajor + rminor);
    d = n * (rmajor - rminor);
    x0 = xC - s; y0 = yC - d;
    xleft = x0; xright = xC + s;
    ylower = yC - s; yupper = yC + s;
    rectangle(IX(xleft), IY(yupper), IX(xright), IY(ylower));
        /* Все эллиптические дуги лежат внутри вычерченного квадрата. */

    for (i=0; i<=2*n; i++)
    for (j=0; j<2*n; j+=2)
    {   x = x0 + i * rmajor + j * rminor;
        y = y0 + (j + 1) * rmajor - i * rminor;
        ellipse(IX(x), IY(y), 270, 360, RXminor, RYmajor);
        x += 2 * rminor;
        ellipse(IX(x), IY(y), 90, 180, RXminor, RYmajor);
    }

    for (j=0; j<=2*n; j++)
    for (i=0; i<2*n; i+=2)
    {   x = x0 + (i + 1) * rmajor + j * rminor;
        y = y0 + j * rmajor - i * rminor;
        ellipse(IX(x), IY(y), 180, 270, RXmajor, RYminor);
        y -= 2 * rminor;
        ellipse(IX(x), IY(y), 0, 90, RXmajor, RYminor);
    }
    endgr( );
}
```

2.3. ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ О ДУГАХ

Как мы могли видеть из предшествующего параграфа, графические функции Турбо Си являются функциями сравнительно низкого уровня — они работают очень быстро, но зависят от технических средств и, особенно из-за проблем с отношением сторон, с ними не всегда приятно работать. Обязательное требование задания значений начального и конечного углов в единицах целых чисел может быть и приводит к эффективной работе программы, но не очень привлекательно с математической точки зрения. В параграфе 2.2 мы уже использовали нашу собственную очень простую функцию (*circle_uc*) для вычерчивания окружности, а теперь попробуем составить аналогичную полезную функцию *are_uc* для вычерчивания дуги окружности при задании начального и конечного углов в виде вещественных чисел, выражающих значения углов в радианах. Напомним, что хорошо известные функции языка Си *cos* и *sin*, которые применяются довольно часто, также воспринимают значения углов, выраженных в радианах. Чтобы функция *arc_uc* работала достаточно быстро, мы построим ее на основе функций Турбо Си, предназначенных для вычерчивания отрезков прямых линий. Таким образом, мы будем аппроксимировать дугу окружности набором отрезков прямых линий, вводя дополнительный аргумент, сообщающий, сколько должно быть таких отрезков. Как и для функции *arc* в Турбо Си предположим, что дуга должна вычерчиваться от начального угла до конечного в положительном направлении против часовой стрелки. Доступность для пользователей значений координат позиций начальной и конечной точек дуги может привести к значительной экономии вычислительного времени, поэтому заимствуем идею из Турбо Си и определим тип структуры *arc_uctype*, аналогичный структуре *arccoordstype*, а также функцию *getarc_uc*, аналогичную функции *getarccoods* в Турбо Си. Для этого в заголовочный файл GRASPTC.H введем следующий текст:

```
#define PI 3.141592653589793

void arc_uc (   float xC,          float yC,          float stangle,
                float endangle,    float radius,      int nilnesegments);

struct arc_uctype
{ float x, y, xstart, ystart, xend, yend;
};
```

Хотя этот файл первоначально предназначался для использования в прикладных программах, мы его будем также использовать в файле GRASPTC.C, к которому теперь добавим:

```
#include <math.h>
#include "grasptc.h"

static float xarc_C, yarc_C, xarc_start, yarc_start, xarc_end, yarc_end;

void arc_uc (   float xC,          float yC,          float stangle,
                float endangle,    float radius,      int nlinesegments)

{   float theta, phi;
    int i;

    while (endangle < stangle) endangle += 2*PI;
    theta = (endangle - stangle)/nlinesegments;
    xarc_start = xC + radius*cos(stangle);
    yarc_start = yC + radius*sin(stangle);
    xarc_end = xC + radius*cos(endangle);
    yarc_end = yC + radius*sin(endangle);
    move(xarc_start, yarc_start);

    for (i=1; i<nlinesegments; i++)
    {   phi = stangle + i * theta;
        draw(xC+radius*cos(phi), yC+radius*sin(phi));
    }

    draw(xarc_end, yarc_end);
    xarc_C = xC; yarc_C = yC;
}

void getarc_uc(struct arc_uctype *arccoords)

{   arccoords->x = xarc_C;
    arccoords->y = yarc_C;
    arccoords->xstart = xarc_start;
    arccoords->ystart = yarc_start;
    arccoords->xend = xarc_end;
    arccoords->yend = yarc_end;
}
```

При модифицированных таким образом файлах GRASPTC.H и GRASPTC.C, мы теперь можем, например, написать программу BOX1, приведенную ниже. Она может показаться несколько более сложной, чем эквивалентная программа BOX в параграфе 2.2, но написать ее легче, поскольку в ней используется только один тип координат и ее гораздо легче адаптировать к другим ситуациям, поскольку углы задаются в виде чисел с плавающей точкой, а не в виде целых чисел.

```
/* BOX1:                                                          */
/*        Треугольник с закругленными углами при задании данных   */
/*        в единицах пользовательских координат.                  */

#include "grasptc.h"

main()

{   float xleft, xright, ylower, yupper, r;
    struct arc_uctype P[3];

    initgr();
    xleft = x_max/2 - 1.0; xright = xleft + 2.0;
    ylower = y_max/2 - 1.0; yupper = ylower + 2.0;
    r = 0.5;
    arc_uc(xleft, ylower, PI, 1.5*PI, r, 20);
    getarc_uc(P);                          /* Адрес точки P[0]  */
    arc_uc(xright, ylower, 1.5*PI, PI/4, r, 25);
    getarc_uc(P + 1);                      /* Адрес точки P[1]  */
    arc_uc(xleft, yupper, Pi/4, PI, r, 25);
    getarc_uc(P + 2);                      /* Адрес точки P[2]  */
    line_uc ( P[0].xend, P[0].yend, P[1].xstart, P[1].ystart);
    line_uc ( P[1].xend, P[1].yend, P[2].xstart, P[2].ystart);
    line_uc ( P[2].xend, P[2].yend, P[0].xstart, P[0].ystart);
    endgr();
}
```

Результат работы программы BOX1 аналогичен результату, полученному при использовании программы BOX и он был изображен на рис. 2.3. Новый подход к составлению программы BOX1 имеет свою цену — она работает значительно медленнее, чем программа BOX.

Дуга по трем заданным точкам

Поскольку мы решили больше не задумываться о пикселах и отношении сторон, мы можем сконцентрировать свое внимание на более интересных задачах с точки зрения геометрии. Например, составим такую функцию, которая по трем заданным точкам A, B и C вычертит дугу окружности, которая начинается в точке A и заканчивается в точке C. Очевидно, что все три точки A, B и C не должны лежать на одной прямой линии. Воспользуемся перпендикулярами, проходящими через середины отрезков AB и BC. Точка пересечения этих двух перпендикуляров является центром O окружности, проходящей через три заданные точки A, B и C. На рис. 2.5 показаны эти три точки и перпендикуляры OD и OE к соединяющим их отрезкам. Во многих практических

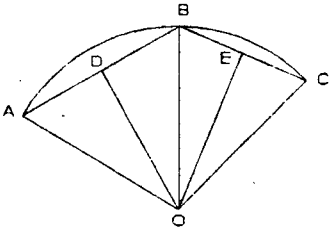


Рис. 2.5. Хорды и перпендикулярные биссектрисы

применениях нам потребуется построить вектор **n**, перпендикулярный заданному вектору **u**. Это очень легко сделать, если мы знакомы с понятием скалярного произведения двух векторов. Если имеем

$$\mathbf{u} = [u_1 \quad u_2]$$

то можно использовать выражение

$$\mathbf{n} = [n_1 \quad n_2] = [u_2 \quad -u_1]$$

для определения требуемого вектора перпендикуляра. Это соотношение можно проверить путем вычисления скалярного произведения этих двух векторов:

$$\mathbf{u} \cdot \mathbf{n} = u_1 n_1 + u_1 n_2 = u_1 u_2 + u_2(-u_1) = 0$$

Любые два ненулевых вектора с нулевым скалярным произведением взаимно перпендикулярны, поэтому можно считать, что вектор **n** является желаемым перпендикулярным вектором. Воспользуемся этим способом дважды для получения векторов **n** и **m**, перпендикулярных отрезкам AB и BC соответственно. Вычислим также положения средней точки D для отрезка AB и средней точки E для отрезка BC. Зная координаты вычисленных точек D и E, можно на основе векторов **n** и **m** вычислить положение точки O — центра окружности, проходящей через точки A, B и C. При $\mathbf{D} = [x_D \quad y_D]$, можно сказать, что прямая линия AB имеет векторное представление

$$\mathbf{D} + \lambda \mathbf{n}$$

Аналогичным образом, для прямой линии, проходящей через точку E и перпендикулярной отрезку BC, можно записать:

$$\mathbf{E} + \mu \mathbf{m}$$

Эти две перпендикулярные прямые линии пересекаются в точке O. Значения коэффициентов λ и μ для этой точки пересечения можно получить, решая векторное уравнение

$$\mathbf{D} + \lambda\mathbf{n} = \mathbf{E} + \mu\mathbf{m}$$

которое можно переписать в виде

$$x_D + \lambda n_1 = x_E + \mu m_1$$
$$y_D + \lambda n_2 = y_E + \mu m_2$$

Фактически нам незачем вычислять значение μ, поэтому данную систему решим относительно λ и найдем

$$\lambda = \frac{m_2(x_E - x_D) - m_1(y_E - y_D)}{m_2 n_1 - m_1 n_2}$$

Теперь можно вычислить координаты точки O:

$$x_O = x_D + \lambda n_1$$
$$y_O = y_D + \lambda n_2$$

Зная вектор $\mathbf{OA} = \mathbf{r} = [x_A - x_O \quad y_A - y_O]$, сначала вычислим величину радиуса

$$r = \sqrt{(r_1^2 + r_2^2)}$$

а затем значение начального угла дуги. Этот угол равен $\operatorname{arctg}(r_2/r_1)$, если r_1 имеет положительный знак, то есть точка A лежит справа от точки O. Если точка A расположена слева от точки O, как на рис. 2.5, то к этой величине нужно добавить значение константы π. И, наконец, если точки A и O имеют одинаковую координату x, то этот угол равен либо $\pi/2$, либо $3\pi/2$, в зависимости от того, как расположена точка A относительно точки O — выше или ниже. Наша функция *angle* вычисляет значение этого угла в единственном условном выражении.

Знаменатель $m_2 n_1 - m_1 n_2$, используемый при вычислении коэффициента λ, можно также записать в виде:

$$Determinant = \begin{vmatrix} n_1 & m_1 \\ n_2 & m_2 \end{vmatrix}$$

Это число будет использовано не только для вычисления коэффициента λ, но также для определения, располагаются ли эти

три точки A, B, C именно в этом порядке, в направлении против движения часовой стрелки или по часовой стрелке. В этом нет необходимости, если вычерчивается полная окружность, но теперь, когда мы хотим вычертить дугу, нам следует быть осторожными — строго говоря, мы не сможем вычертить желаемую дугу, начинающуюся в точке A и заканчивающуюся в точке C, поскольку с помощью элементарных функций *arc* и *acr_uc* для дуг окружности мы можем вычерчивать дуги только в направлении против движения часовой стрелки. Точки A, B, C, именно в этом порядке, расположены по движению против часовой стрелки в том случае, если переменная *Determinant* имеет положительный знак, и по часовой стрелке, если этот знак отрицательной. В последнем случае нам просто придется вычерчивать дугу в противоположном направлении, то есть от точки C к точке A. И, наконец, если точки A, B и C лежат на одной прямой линии, то значение переменной *Determinant* равно нулю. В этом случае мы ничего не будем вычерчивать. В каждом из этих трех возможных случаев наша функция *drawarc3* будет возвращать значение переменной *Determinant*, так что пользователь всегда может знать, какой случай имел место. Кроме того, для пользователя может быть интересным узнать значение радиуса, которое будет передаваться через дополнительный параметр. Заметим, что пользователь может получить позицию центра O путем обращения к функции *get_arc*, поскольку фактическое вычерчивание дуги выполняется функцией *arc_uc*.

Добавим в модуль GRASPTC следующие функции *angle* и *drawarc3*:

```
float angle(float x, float y)

{   return (x > 0 ? atan(y/x) :
            x < 0 ? PI + atan(y/x) :
            y >= 0 ? PI/2 : 3*PI/2);
}

float drawarc3 (float xA, float yA,
                float xB, float yB,
                float xC, float yC,
                float *pr)

{   float u1, u2, n1, n2, xD, yD, v1, v2, m1, m2, xE, yE, lambda,
        xO, yO, r1, r2, r, stangle, endangle, phiA, phiC, determinant;

    int nsteps;
```

```
    u1 = xB - xA; u2 = yB - yA;          /* Вектор u направлен от A к B        */
    n1 = u2; n2 = -u1;                   /* Вектор n перпендикулярен к u       */
    xD = (xA+xB)/2; yD = (yA+yB)/2;          /* D = средняя точка AB           */
    v1 = xC - xB; v2 = yC - yB;          /* Вектор v направлен от B к C        */
    m1 = v2; m2 = -v1;                   /* Вектор m перпендикулярен к v       */
    xE = (xB+xC)/2; yE = (yB+yC)/2;  /* E = средняя точка BC                   */

    determinant = m2*n1 - m1*n2;
    if (determinant == 0.0) return 0.0;
    lambda = (m2*(xE-xD) - m1*(yE-yD))/determinant;
            /* Вектор lambda.n направлен из точки D в точку O                  */
    xO = xD + lambda * n1;
    yO = yD + lambda * n2;
            /* O = центр окружности, проходящей через точки A, B и C  */

    r1 = xA-xO; r2 = yA-yO;      /* Вектор (r1, r2) направлен из O в A         */
    r = sqrt(r1*r1 + r2*r2);
    phiA = angle(r1, r2);
    phiC = angle(xC-xO, yC-yO);

    if (determinant > 0)
    {   stangle = phiA; endangle = phiC;
    }   else
    {   stangle = phiC; endangle = phiA;
    }

    if (endangle < stangle) endangle += 2*PI;
    nsteps = (int)(r * (endangle - stangle) * 10) + 1;
            /* Число шагов зависит как от радиуса, так и от угла          */
    arc_uc(xO, yO, stangle, endangle, r, nsteps);
    *pr = r;
    return determinant;
}
```

Программа ARC3 демонстрирует применение вышеописанной функции *drawarc3* (внутри ее используется функция *angle*). Программа маркирует заданные точки, как показано на рис. 2.6. В данном примере точки A, B, C расположены по часовой стрелке, так что фактически дуга проходит через точки C, B, A.

```
/* ARC3:        Дуга через три точки A, B, C.                    */

#include <math.h>
#include "grasptc.h"

void mark(float x, float y, char *str)
{   float d=0.08;
    line_uc(x-d, y-d, x+d, y+d);
    line_uc(x-d, y+d, x+d, y-d);
    outtextxy(IX(x), IY(y)+10, str);
}
```

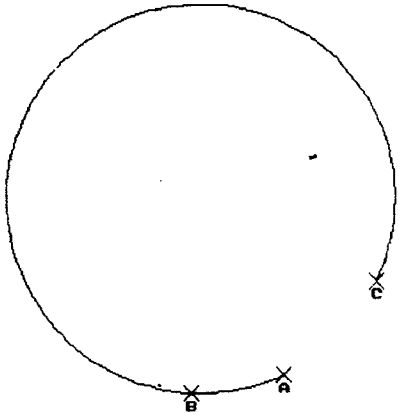


Рис. 2.6. Пример работы программы ARC3

```
main( )

{   float xA, yA, xB, yB, xC, yC, r;
    printf
        ("Введите координаты трех точек — xA, yA, xB, yB, xC, yC :\n");
    scanf("%f %f %f %f %f %f", &xA, &yA, &xB, &yB, &xC, &yC);

    initgr( );
    settextjustify(CENTER_TEXT, CENTER_TEXT);
    mark(xA, yA, "A"); mark(xB, yB, "B"); mark(xC, yC, "C");
    drawarc3(xA, yA, xB, yB, xC, yC, &r);
    endgr( );
}
```

2.4. ГАЛТЕЛЬ (ЗАКРУГЛЕНИЕ)

В инженерных приложениях часто требуется заменить острые углы закруглениями, что показано на рис. 2.7. Дуги, заменяющие острые углы, называются галтелями. В этом очень простом примере мы имеем только горизонтальные и вертикальные углы, что значительно упрощает вычерчивание галтелей.

Не будем ограничиваться этим простым случаем и составим такую функцию, которая может вычерчивать закругления для любых углов. Будем основываться на дуге заданного радиуса *r* и трех точках A, B и C, как показано на рис. 2.8. Центр O дуги и ее две концевые точки D (на отрезке AB) и E (на отрезке BC) будут записаны в глобальную структуру таким же образом, как это было сделано в функции *arc_uc* в начале параграфа 2.3. Наша новая

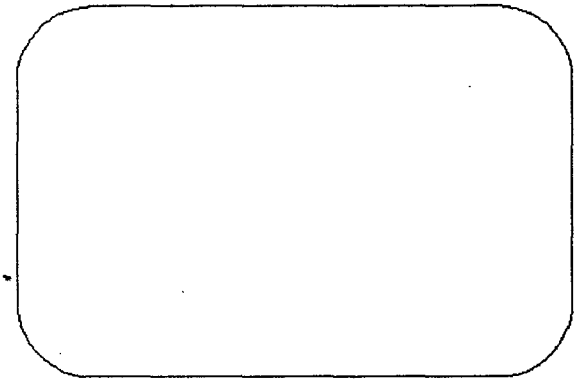


Рис. 2.7. Прямоугольник с закруглениями

функция будет вычерчивать отрезки прямых линий AD и EC. Таким образом пользователь может задать точки A и C, которые, подобно точке B, не принадлежат фигуре, подлежащей вычерчиванию. Примерами таких точек A и C в вышеприведенном рисунке могут служить два противоположных угла исходного прямоугольника, послужившего основой для построения рис. 2.7, и другие последующие рисунки.

Найдем позиции точек O, D, E. Сначала вычислим оба вектора n и m, оба единичной длины и направленные из точки B в точки A и C соответственно:

$$\mathbf{n} = \mathbf{BA}/BA$$
$$\mathbf{m} = \mathbf{BC}/BC$$

Обратите внимание на обозначение векторов в этих формулах полужирным шрифтом: **BA** — это вектор с начальной точкой B и конечной точкой A, а через BA обозначена его длина.

Вычислим сумму векторов **n** и **m** и поделим этот новый вектор на его длину $|s|$, так что результирующий вектор **b** будет иметь длину 1:

$$\mathbf{s} = (\mathbf{n} + \mathbf{m})$$
$$\mathbf{b} = \mathbf{s} \ / \ |\mathbf{s}|$$

Вектор **b** имеет направление биссектрисы угла ABC. Мы используем его для определения положения точки O, которая лежит на этой биссектрисе. Будем считать, что угол ABC равен 2α, то есть биссектриса образует два угла α. Поскольку радиус r

равен DO = BOsin α, а вектор b имеет длину 1, то будем иметь векторное равенство:

$$\mathbf{BO} = \frac{r}{\sin\alpha}\mathbf{b}$$

Значение α найдем из того условия, что ранее вычисленный вектор s имеет длину 2 cos α, поэтому можно ввести подстановку

$$\cos\alpha = 1/2\ |\mathbf{s}|$$

$$\sin\alpha = \sqrt{(1 - \cos^2\alpha)}$$

Проекция вектора BO на прямую линию BA равна BD. Ее длина равна скалярному произведению

$$\mathbf{BO}:\mathbf{n}$$

которое можно использовать для нахождения точки D. Эту же длину используем аналогично для вычисления позиции точки E.

Если бы теперь нам захотелось применить функцию *arc_uc*, то нам бы пришлось вычислить начальный и конечный углы и определить ориентацию точек A, B, C подобно тому, как это делалось при составлении функции *drawarc3*. Однако, если бы мы смогли найти третью точку на дуге, то можно было бы воспользоваться этой последней функцией и тогда бы не пришлось беспокоиться об этих углах и ориентации. Используем для этой цели

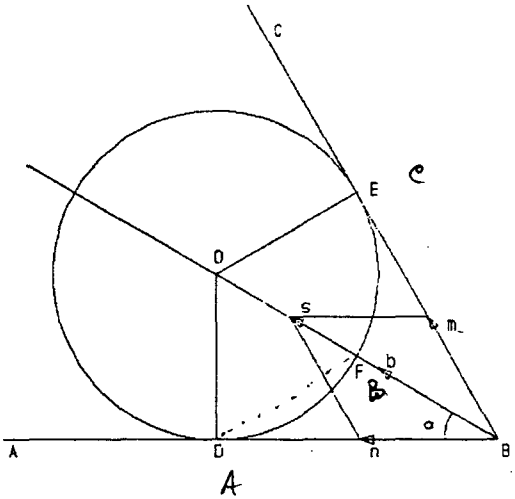


Рис. 2.8. Конструирование закругления

точку F, образованную пересечением дуги с прямой линией BO. Очевидно, что точка F лежит на расстоянии r от точки O и, поскольку вектор **b** имеет единичную длину, будем иметь соотношение $\mathbf{OF} = -r\mathbf{b}$, что позволяет нам найти координаты точки F.

Нижеследующая функция *fillet* основана на приведенном анализе. Ее также включим в состав модуля GRASPTC (и объявим ее в заголовочном файле GRASPTC.H).

```
float fillet(float xA, float yA, float xB, float yB,
        float xC, float yC, float r)

{   float n1, n2, m1, m2, BA, BC, s1, s2, length_of_s,
        b1, b2, cosa, sina, BO1, BO2, xO, yO, proj,
        xD, yD, xE, yE, xF, yF, q;
    n1 = xA - xB; n2 = yA - yB;
    BA = sqrt(n1*n1 + n2*n2);
    n1 /= BA; n2 /= BA;

    m1 = xC - xB; m2 = yC - yB;
    BC = sqrt(m1*m1 + m2*m2);
    m1 /= BC; m2 /= BC;

    s1 = n1 + m1; s2 = n2 + m2;
    length_of_s = sqrt(s1*s1 + s2*s2);
    b1 = s1/length_of_s; b2 = s2/length_of_s;
    cosa = length_of_s/2;
    sina = sqrt(1 - cosa*cosa); q = r/sina;
    BO1 = q * b1; BO2 = q * b2;
    xO = xB + BO1; yO = yB + BO2;
    proj = BO1 * n1 + BO2 * n2;

    xD = xB + proj * n1; yD = yB + proj * n2;
    xE = xB + proj * m1; yE = yB + proj * m2;
    xF = xO - r * b1; yF = yO - r * b2;
    return drawarc3(xD, yD, xF, yF, xE, yE, &r);
}
```

Работу этой функции можно продемонстрировать в следующей программе, которая использована для построения рис. 2.9.

```
/* FILLET:                                                    */
/*      Функция 'fillet' в применении к произвольному треугольнику.  */

#include <math.h>
#include "grasptc.h"

main()

{   float xA, yA, xB, yB, xC, yC, r, orientation;
    struct arc_uctype A, B, C;
```

```
    printf ( "\nВведите значения координат xA, yA, xB, yB, xC, yC\n"
             "вершин A, B, C треугольника.\n\nТочки должны следовать"
             " в направлении против часовой стрелки:\n\n");
    scanf("%f %f %f %f %f %f", &xA, &yA, &xB, &yB, &xC, &yC);

    printf ( "\nРадиус скругления (в дюймах) : ");
    scanf("%f", &r);

    Initgr( );
    orientation = fillet(xA, yA, xB, yB, xC, yC, r);

    if (orientation < 0)
    {   closegraph( );
        printf ( "Точки A, B, C должны следовать против часовой стрелки");
        exit(1);
    }

    getarc_uc(&B);
    fillet(xB, yB, xC, yC, xA, yA, r);

    getarc_uc(&C);
    fillet(xC, yC, xA, yA, xB, yB, r);

    getarc_uc(&A);

    line_uc(A.xend, A.yend, B.xstart, B.ystart);
    line_uc(B.xend, B.yend, C.xstart, C.ystart);
    line_uc(C.xend, C.yend, A.xstart, A.ystart);
    endgr( );
}
```

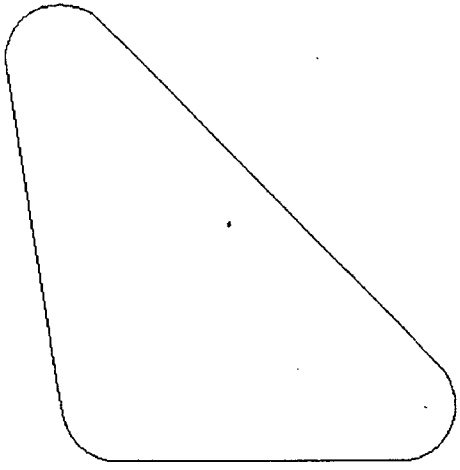


Рис. 2.9. Произвольный треугольник с закруглениями

2.5. УСТАНОВКА ПРЕРЫВАНИЯ ПРОГРАММЫ

Иногда желательно прекратить выполнение программы путем нажатия на клавиши Ctrl-Break или Ctrl-C. К сожалению, если не принять специальных мер, то нажатие на эти клавиши не приведет к желаемому эффекту. Это очень раздражает, особенно если компьютер находится в графическом режиме. Поэтому необходимо предпринять какие-то специальные меры. Теперь, когда количество доступных графических программ растет очень быстро, возрастает риск в допущении ошибок, которые могут привести к зацикливанию программы. Кроме того, необходимо иметь средство для прекращения выполнения программ, если они работают дольше, чем предполагалось. В конце параграфа приведем пример, когда может возникнуть такая ситуация.

Есть некоторая разница между нажатием клавиш Ctrl-Break и Ctrl-C, которую вскоре поясним. Пока для нажатия обеих комбинаций будем использовать термин "*консольное прерывание*". В Турбо Си имеется функция *ctrlbrk*, которая в качестве аргумента воспринимает имя функции для определения тех действий, которые должны быть выполнены в случае консольного прерывания. В нашем случае эта функция не полностью нас удовлетворяет по двум причинам:

1. Компьютер не всегда выполняет наш запрос на консольное прерывание. Прерывание происходит только при выполнении операций ввода/вывода посредством программ операционной системы, но не срабатывает при обращении к дисплейной памяти, что имеет место при операциях вывода графической информации.
2. В непредсказуемых ситуациях некоторые пользователи могут сначала нажать на некоторые другие клавиши, например клавишу ввода ('Enter'), и только если это не приводит ни к какому результату, то они нажмут комбинацию Ctrl-C. Если вводимые таким образом символы не воспринимаются программой, то они еще находятся во входном буфере в момент нажатия комбинации Ctrl-C и поэтому запрос на прерывание останется незамеченным (комбинация Ctrl-Break отличается тем, что при ее вводе очищается входной буфер).

Описываемое здесь решение проблемы консольного прерывания может оказаться очень эффективным: за исключением подключения специального обработчика прерываний во время

инициализации и удаления его при завершении программы не требуется никаких дополнительных действий во время выполнения программы. Это очень важно. Предположим, например, что программа вызывает функцию *putpixel* для заполнения некоторой области на экране. Было бы очень неэффективно при каждом таком обращении (которых может быть многие тысячи) выполнять специальные действия для проверки возможного факта консольного прерывания.

При каждом нажатии клавиши на клавиатуре выполняется так называемая *программа обработки прерывания* (или "*обработчик прерывания*"). Программа обработки прерывания работает аналогично обычной функции. Адрес ее размещения записан в некоторой фиксированной области, называемой *вектором прерывания*. Имеются уникальные целые числа 0, 1, 2, ..., ассоциированные с каждым вектором прерывания. Поскольку каждый вектор по длине занимает четыре байта, вектор для прерывания с номером 0 расположен по адресу 0. Тогда прерывание с номером 1 имеет адрес 4 и так далее. Интересующее нас прерывание имеет номер 9, следовательно его вектор прерывания начинается с адреса 4 × 9 = 36. В действительности этот адрес мы сами использовать не будем. В Турбо Си для обработки векторов прерываний имеются две функции *getvect* и *setvect*, объявленные в заголовочном файле DOS.H. Перед заменой старого вектора прерываний мы запомним его в переменной типа "указатель" *oldInt9* для возможности использования и восстановления его в последующем. Эту переменную типа указателя объявим следующим образом:

```
static void interrupt (*oldInt9)(void);
```

Напомним, ключевое слово *static* указывает, что имя *oldInt9* будет известно только в текущем программном модуле, который в нашем случае будет модуль GRASPTC. Без нового ключевого слова *interrupt* это было бы объявление указателя на функцию (вместо указателя на программу обработки прерываний). Ключевое слово *interrupt* также появится в определении новой программы обработки прерываний *newInt9*, которое имеет вид:

```
static void interrupt newInt9(void);
{   ...
}
```

Поскольку содержание этой функции (обозначенное многоточием ...) сравнительно сложное; оно будет описано позднее. Теперь можно будет использовать функцию *installBreak*, приведенную ниже, для подключения нового обработчика прерываний с именем *newInt9*.

```
static int handler_installed = 0;

static void installBreak(void)

{   if (!handler_installed)
    {   oldInt9 = getvect(9); setvect(9,newInt9);
        handler_installed = 1;
    }
}
```

Функцию *installBreak* будем вызывать из нашей функции *initgr*. Переменная *handler_installed* показывает, был ли подключен новый обработчик прерываний. После выполнения всех работ с графической информацией нормально вызывается наша функция *endgr* и при ее выполнении восстанавливается старый обработчик прерываний путем обращения к новой функции:

```
static void restore_old_break(void)

{   if (handler_installed)
    { setvect(9, oldInt9); handler_installed = 0;
    }
}
```

Наконец следует рассмотреть содержание нового обработчика прерываний *newInt9*. Напомним, что эта подпрограмма будет вызываться немедленно после каждого нажатия на клавишу. Первое, что нужно сделать, это выполнить все действия, которые осуществлялись старым обработчиком прерываний. Мы можем использовать указатель на функцию точно также, как и само имя функции, поэтому можно записать

```
oldInt9();
```

Как мы уже видели, любой введенный символ, еще не считанный программой, находится в буфере. Нормально этот буфер используется в виде очереди, действующей по принципу "первый вошел, первый вышел". В нашем случае последним введенным символом может оказаться комбинация Ctrl-C, а если это так, то мы должны его использовать, просто пропустив все предыдущие

символы. Поскольку всегда неизвестно, был ли факт консольного прерывания, необходимо просто начать чтение символов с помощью, скажем, функции *getch*, поскольку тогда такие символы будут исключены из буфера и больше не будут доступны для программы, которая может попытаться прочитать их впоследствии.

Поэтому мы осуществим непосредственную проверку содержимого буфера. Это возможно сделать на основе следующей информации, которая содержится, например, в "*Руководстве программиста персонального компьютера фирмы IBM*" Питера Нортона. Входной буфер начинается с адреса 0x41E. Он кольцевой. Будем использовать термины *голова* и *хвост* для обозначения самого старого символа, который будет считан, и первой свободной позиции соответственно. Эти позиции задаются в виде смещения относительно адреса 0x400, они записываются в виде целых чисел в ячейках с адресами 0x41A и 0x41C. Поскольку имеем

$$0x41E - 0x400 = 0x1E = 1 \times 16 + 14 = 30$$

то наименьшее значение чисел, записываемых по адресам 0x41A и 0x41C, может быть 30. Буфер имеет длину 32 байта, но может содержать не более 16 символов, поскольку каждый символ запоминается в двух байтах в виде кода ASCII, за которым следует *код сканирования*. В нашем случае мы интересуемся символами Ctrl-C и Ctrl-Break, для которых эти коды имеют следующие значения:

	Код ASCII	*Код сканирования*	*Целое число*
Ctrl-C	0x03	0x2E	0x2E03
Ctrl-Break	0x00	0x80	0x8000

Хотя код сканирования располагается *после* кода ASCII, мы получаем эти две составляющие в обратном порядке, если оба байта будут объединяться в одно целое число. (Это хорошо известное свойство процессора 8086, о котором обычно не приходится беспокоиться, программируя на языке Си). Этим объясняется необходимость в третьем столбце таблицы, данные которой используются в следующих определениях констант:

```
#define CTRL_C 0x2E03
#define CTRL_BREAK 0x8000
```

Вместо сравнения двух байт теперь можно сравнивать только одно целое число, что выполняется значительно быстрее. Будем проверять целое число, располагающееся в буфере непосредственно перед хвостом. Поскольку буфер кольцевой, нам следует подумать о специальном случае, когда этот хвост соответствует адресу 0x41E. Если так, то логически предшествующее целое число в буфере располагается в физическом конце буфера, то есть в ячейке с адресом 0x41E + 30. Вместо использования исходных значений смещения 30, 32, ..., 60, мы сначала уменьшим их на 30, а затем поделим на 2 (операцией сдвига вправо), поскольку в целочисленном массиве при изменении индекса на 1 адрес элементов массива изменяется на 2 байта. Этим объясняется вычисление проверяемой позиции в приведенной ниже функции:

```
static int *keybuffer = (int *)0x41E;
static int *buffertail = (int *)0x41C;

static void interrupt newInt9(void)

{   unsigned int tail, code;
    oldInt9();                          /* *buffertail = 30, 32, ..., 60        */
    tail = (*buffertail - 30) >> 1;     /* tail = 0, 1, ..., 15                 */
    code = keybuffer[tail ? tail - 1 : 15];
    if (code == CTRL_C || code == CTRL_BREAK)
    {   to_text(); exit(1);
    }
}
```

Здесь следует заметить, что наш обработчик прерываний не является резидентом в памяти, поэтому он исчезает из памяти, когда заканчивается установившая его графическая программа и начинается новая программа. Будет некорректным, если вектор прерывания (в младшей области памяти по адресу 36) все еще содержит указатель на нашу программу *newInt9*. Поэтому необходимо обращение к функции *restore_old_break* ("восстановление старого прерывания"), для чего к модулю GRASPTC добавим новую функцию *to_text* и используем ее в функции *endgr*:

```
void to_text(void)
{   closegraph();
    restore_old_break();
}

void endgr(void)
{   getch(); to_text();
}
```

Как читатель мог заметить, обращение к функции *to_text* появляется также в функции обработки прерываний *newInt9*. Так что при необходимости немедленного возврата в текстовый режим следует использовать функцию *to_text*, а если перед выходом в текстовый режим пользователь должен нажать какую-либо клавишу, то нужно воспользоваться функцией *endgr*. В любом случае восстанавливается исходный обработчик прерываний.

Не слишком ли много внимания уделено техническим аспектам установки и восстановления обработчиков прерываний?

Как было упомянуто в начале этого параграфа, консольное прерывание может оказаться желательным не только для программ с бесконечным циклом, но также и для таких программ, выполнение которых может занимать очень длительное время. Особенно в случае очень универсальных программ это может произойти тогда, когда входные данные не могут привести к удовлетворительным результатам. Если весь экран компьютерного монитора должен быть заполнен некоторыми фигурами, то нам обычно достаточно увидеть только его малую часть, чтобы решить, стоит ли продолжать. Программа SPIRALS, приведенная ниже, служит хорошим тому примером. При входных данных, первоначально заложенных в саму программу, результатом будет изображение, показанное на рис. 2.10. Но если будут введены другие входные данные, то выполнение программы может занять очень много времени. Если в таких случаях рисунок на экране не понравится, то можно нажать на клавиши Ctrl-Break или Ctrl-C для останова программы.

```
/* SPIRALS:                                                    */
/*                    Рисунок из спиралей                      */

#include <math.h>
#include "grasptc.h"

int narc, nline;
float rmax, n, halfpi, c, r0, delta;

float radius(float phi)

{   return c * phi + r0;
}

spirals4(float x, float y)

{   int i, j, l;
    float phi0, phi, r, phi_offset;
```

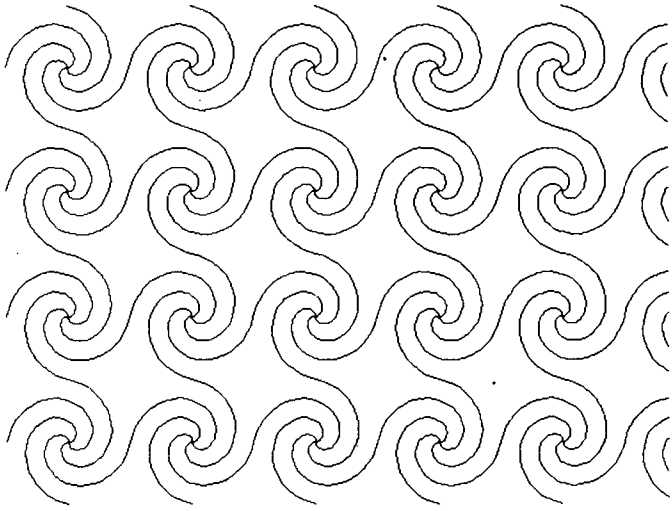


Рис. 2.10. Рисунок из спиралей

```
    for (l=0; l<4; l++)
    {   phi0 = l*halfpi;
        move(x+r0*cos(phi0), y+r0*sin(phi0));
        for (i=0; i<narc; i++)
        for (j=1; j<=nline; j++)
        {   phi_offset = i * halfpi + j * delta;
            phi = phi0 + phi_offset;
            r = radius(phi_offset);
            draw(x+r*cos(phi), y+r*sin(phi));
        }
    }
}

main( )

{   float x, y, rmax;
    halfpi = PI/2;

    printf ( "Сколько отрезков в дуге спирали 90 градусов?"
             "(например, 12) : ");
    scanf("%d", &nline);
    printf ( "Сколько дуг по 90 градусов в каждой спирали?"
             " (например, 3) : ");
    scanf("%d", &narc);
    printf ( "\nРадиус вычисляется по формуле : r = c * phi + r0\n");
    printf ( "(где phi в радианах, r и r0 в дюймах)\n");
    printf ( "Введите значения c и r0 (например, 0.15  0) : ");
    scanf("%f %f", &c, &r0);
```

```
    delta = halfpi/nline; rmax = radius(narc * halfpi);
    Initgr( );
    for (x=0.2+rmax; x<=x_max-rmax-0.1999; x+=2*rmax)
    for (y=0.2+rmax; y<=y_max-rmax-0.1999; y+=2*rmax) spirals4(x, y);
    endgr( );
}
```

2.6. ЗАПОЛНЕНИЕ ОБЛАСТИ

В Турбо Си имеются две функции для вычерчивания полигонов (многоугольников): функция *drawpoly* вычерчивает полигон обычным образом, а функция *fillpoly* вычерчивает сам полигон и заполняет его внутреннюю область. Их объявления (в файле GRAPHICS.H) выглядят так:

```
void far drawpoly(int numpoints, int far *polypoints);
void far fillpoly(int numpoints, int far *polypoints);
```

Можно задать вопрос, а стоит ли уделять внимание функции *drawpoly*, поскольку наши обычные функции вычерчивания линий вполне пригодны для вычерчивания полигонов. Однако функция *fillpoly* очень интересна, а так как функция *drawpoly* имеет те же самые аргументы, то придется затратить очень мало дополнительных усилий для описания также и последней. Первый аргумент *numpoints* обозначает количество пар координат (X_i, Y_i), которые задаются через второй аргумент *polypoints*. Если полигон имеет n вершин (пронумерованных 0, 1, ..., $n-1$), то значение переменной *numpoints* должно быть равно $n+1$ и *polypoints* является начальным адресом следующей последовательности целых чисел:

$$X_0, Y_0, X_1, Y_1, \ldots, X_{n-1}, Y_{n-1}, X_0, Y_0$$

Итак, нам необходимо замыкать контур самостоятельно, записав одну пару координат дважды, как это и сделано в данном примере с точкой (X_0, Y_0). В результате как первый аргумент *numpoints*, так и число пар координат должно быть равно $n+1$ (Руководство по Турбо Си противоречит само себе относительно точного значения параметра *numpoints*, а приведенный пример очень сомнителен с этой точки зрения). Заметим, что здесь нужно снова задавать целочисленные пикселные координаты.

Обе функции начинают с вычерчивания полигона, заданного через указанные аргументы. После завершения вычерчивания

контура задача функции *drawpoly* оказывается выполненной, тогда как функция *fillpoly* затем заполнит его заданным шаблоном с заданным цветом. Чтобы иметь возможность указать, что мы хотим, нам нужно познакомиться еще с одной функцией, объявленной в файле GRAPHICS.H следующим образом:

```
void setfillstyle(int pattern, int color);
```

Эта функция применяется для выбора типа шаблона и цвета, которые будут использоваться в процессе заполнения области. Имеются двенадцать типовых шаблонов заполнения, обозначенных номерами 0, 1, ..., 11, для которых можно использовать символические константы:

Имя	*Значение*	
EMPTY_FILL	0	/* заполнение области цветом фона */
SOLID_FILL	1	/* сплошная заливка цветом заполнения */
LINE_FILL	2	/* заполнение горизонтальными линиями —— */
LTSLASH_FILL	3	/* заполнение наклонными линиями /// */
SLASH_FILL	4	/* заполнение наклонными толстыми линиями /// */
BKSLASH_FILL	5	/* заполнение обратными наклонными толстыми линиями \\\ */
LTBKSLASH_FILL	6	/* заполнение обратными наклонными тонкими линиями \\\ */
HATCH_FILL	7	/* заполнение прямоугольной горизонтальной штриховкой */
XHATCH_FILL	8	/* заполнение косой штриховкой */
INTERLEAVE_FILL	9	/* заполнение косой перекрывающейся штриховкой */
WIDE_DOT_FILL	10	/* заполнение редкими точками */
CLOSE_DOT_FILL	11	/* заполнение близко расположенными точками */

В большинстве случаев достаточно этих двенадцати предопределенных шаблонов. Если их недостаточно, то можно определить свой собственный шаблон, используя функцию Турбо Си *setfillpattern*. Правила использования этой функции описаны в *Справочнике по Турбо Си*.

Шаблоны 0, 1, ..., 11, сформированные программой FILLDEMO, показаны на рис. 2.11.

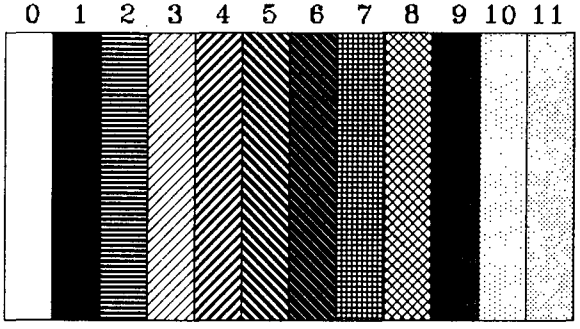


Рис. 2.11. Шаблоны заполнения

```
/* FILLDEMO:                                                    */
/*        Демонстрация применения типовых шаблонов заполнения   */

#include <stdio.h>
#include "grasptc.h"

main( )

{   int i, rctangle[10], foregroundcolor, width, XA, YA, XB, YB,
        XC, YC, XD, YD, displacement, txtheight, Xmax, Ymax;
    char str[3];

    printf("\nВведите границы области Xmax и Ymax (например, 600 200) : ");
    scanf("%d %d", &Xmax, &Ymax);

    initgr( );
    settextjustify(CENTER_TEXT, TOP_TEXT);
    settextstyle(TRIPLEX_FONT, HORIZ_DIR, 2);
    txtheight = textheight("A");
    width = Xmax/12;

    XA = XD = 0;
    XB = XC = width;
    YC = YD = txtheight + 8;
    YA = YB = Ymax;

    rctangle[1] = YA;
    rctangle[3] = YB;
    rctangle[5] = YC;
    rctangle[7] = YD;
    rctangle[9] = YA;

    foregroundcolor = getcolor( );
```

```
    for (i=0; i<12; i++)
    {   displacement = 1 + i * width;
        sprintf(str, "%2d", i);
        outtextxy(width/2 + displacement, 0, str);
        rctangle[0] = XA + displacement;
        rctangle[2] = XB+ displacement;
        rctangle[4] = XC + displacement;
        rctangle[6] = XD + displacement;
        rctangle[8] = rctangle[0];
        setfillstyle(i, foregroundcolor);
        fillpoly(5, rctangle);
    }
    endgr( );
}
```

Если читатель желает проследить за процессом заполнения более сложных полигонов, чем простые прямоугольники, то можно воспользоваться программой POLFILL. В ней вычерчивается достаточно сложный полигон (область между двух спиралей), который затем заполняется заданным типовым шаблоном.

```
/* POLFILL                                                          */
/*                  :   Заполнение очень сложного полигона.         */

#include <stdio.h>
#include <math.h>
#include <alloc.h>
#include <stdlib.h>
#include "grasptc.h"

main( )

{   int n, N, pattern, *points, i, k;
    float angle, theta, dr, cosith, sinith, x1, y1, x2, y2,
        r1, r2, xC, yC, xmax, ymax;
    printf ( "\nПолигон (в форме спирали) будет иметь 2n вершин."
             " Введите число n : ");
    scanf("%d", &n);
    printf ( "\nВведите размеры области xmax и ymax"
             " (например, 10.0 7.0) : ");
    scanf("%f %f", &xmax, &ymax);

    N = 2 * n;
    printf ( "\nКод шаблона заполнения ? (меньше 12) : ");
    scanf("%d", &pattern);
    printf ( "\nЭлементарный угол в градусах (например, 20) : ");
    scanf("%f", &angle);
    theta = angle * PI/180;
    k = PI/theta;
    points = (int *)malloc(2 * (N+1) * sizeof(int));
```

```
    if (points == NULL)
                {printf("\nПроблема с выделением памяти"); exit(1);}
    initgr( );
    xC = 0.5 * xmax; yC = 0.5 * ymax;
    setfillstyle(pattern, getcolor( ));
    dr = 0.3 * xmax / n;

    for (i=0; i<n; i++)
    {   r1 = i * dr;
        r2 = (i + k) * dr;
        cosith = cos(i * theta);
        sinith = sin(i * theta);
        x1 = xC + r1 * cosith; y1 = yC + r1 * sinith;
        x2 = xC + r2 * cosith; y2 = yC + r2 * sinith;

        points[2*i] = IX(x1);
        points[2*i+1] = IY(y1);
        points[2*(N-1)-2*i] = IX(x2);
        points[2*(N-1)-2*i+1] = IY(y2);
    }

    points[2*N] = points[0];
    points[2*N+1] = points[1];
    fillpoly(N+1, points);
    endgr( );
}
```

На рис. 2.12 показан результат работы этой программы при задании числа $n = 200$ (то есть полигон с 400 вершинами!) и с параметрами *pattern* = 1 = *SOLLID_FILL* и *angle* = 20 (градусов).

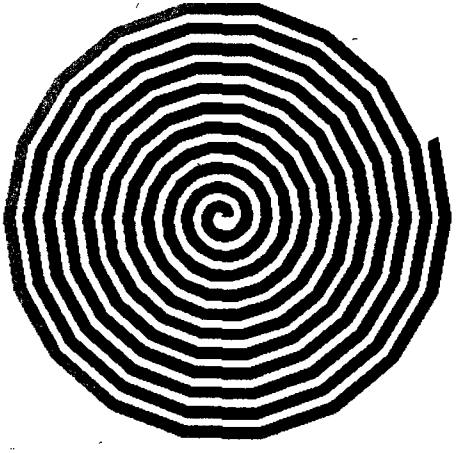


Рис. 2.12. Заполненный полигон

Доступные типы шаблона можно использовать для получения различных типов затенения, что показано на рис. 2.13. На этом рисунке изображен "магический треугольник", который создал Эшер (M.C. Escher). Здесь он сформирован приведенной ниже программой MAGTRIA:

```
/* MAGTRIA:                                                    */
/*                                  Магический треугольник,    */

#include <math.h>
#include "grasptc.h"

main( )

{   float a, b, ha, hb, c, s, sq3, xC, yC, x[6], y[6], xj, yj, r, w;
    int points[14], i, j, foregrcolor, fillpattern;
    printf ( "\nРазмер стороны внутреннего треугольника (например, 2.5) : ");
    scanf("%f", &a); ha = 0.5 * a;

    sq3 = sqrt(3.0); r = ha * sq3/3.0;
        /* r — радиус окружности, вписанной во внутренний треугольник */

    c = -0.5;     /* cos 120 degr. */
    s = 0.5 * sq3; /* sin 120 degr. */
    printf ( "\nВведите параметр толщины каждого закрашиваемого"
                "\n треугольника (например, 0.5) : ");
    scanf("%f", &w);
    b = w/s;
        /* b — длины сторон трех малых треугольников, которые        */
        /*  отсекаются в вершинах наибольшего треугольника.          */

    hb = 0.5 * b;
    initgr( );
    xC = 0.5 * x_max; yC = 0.5 * (y_max - r);
    x[0] = -ha - 2 * b - hb; y[0] = -r - w;
    x[1] = x[0] + hb; y[1] = -r - 2 * w;
    x[2] = -x[1]; y[2] = y[1];
    x[3] = hb; y[3] = 2 * r + w;
    x[4] = 0.0; y[4] = 2 * r;
    x[5] = ha + hb; y[5] = y[0];
    foregrcolor = getcolor( );

    for (i=0; i<3; i++)
    {   for (j=0; j<6; j++)
        {   xj = x[j]; yj = y[j];
            points[2*j] = IX(xC + xj);
            points[2*j+1] = IY(yC + yj);
            x[j] = c * xj - s * yj;
            y[j] = s * xj + c * yj;
                /* Поворот на 120 градусов для следующего значения i.  */
        }
```

```
        points[12] = points[0]; points[13] = points[1];
        fillpattern =
        (   i == 0 ? SOLID_FILL :
            i == 1 ? INTERLEAVE_FILL :
                     WIDE_DOT_FILL);
        setfillstyle(fillpattern, foregrcolor);
        fillpoly(7, points);
    }
    endgr();
}
```

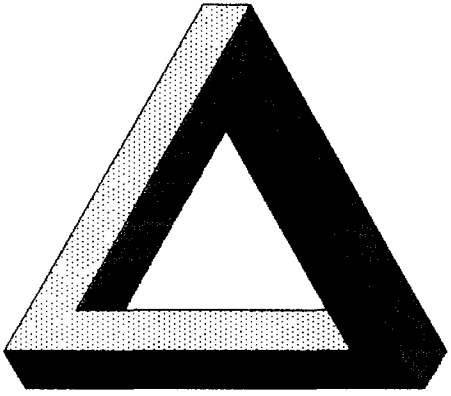


Рис. 2.13. Магический треугольник

Заполнение окружностей и эллипсов

В Турбо Си версии 2.0 (но не в версии 1.5) можно начертить заполненные окружности или эллипсы путем использования функции *fillellipse*, объявленной в файле GRAPHICS.H как

```
void far fillellipse(int x, int xradius, int yradius);
```

Аналогично функции *fillpoly* в ней применяются текущие цвет и шаблон заполнения. Заметим, что в функции *fillellipse* точно такие же аргументы, как и в функции *ellipse*. Несмотря на свое название (в переводе — "*заполнение эллипса*"), вероятнее всего функция *fillellipse* будет чаще использоваться для вычерчивания окружностей, например, подобных изображенным на рис. 2.14, чем для некруговых эллипсов. (Но при этом наличие в названии термина "эллипс" остается корректным, поскольку окружность является частным случаем эллипса).

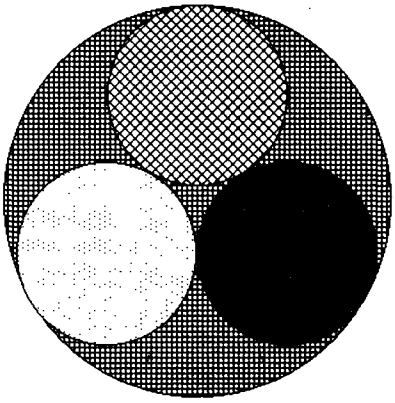


Рис. 2.14. Четыре заполненных круга

Следующая функция базируется на наших пользовательских координатах. Добавим ее к модулю GRASPTC и включим объявление в файл GRASPTC.H:

```
void fillcircle_uc(float x, float y, float radius)
{   fillellipse(IX(x), IY(y), XPIX(radius), YPIX(radius));
}
```

Кроме заполнения полигонов и эллипсов мы можем заполнить любую ограниченную область, используя функцию Турбо Си *floodfill* ("*Заполнение заливкой*"), объявленную в файле GRASPTC.H как:

```
void far floodfill(int x, int y, int border);
```

Подобно функции *polyfill*, функция *floodfill* использует текущие шаблон и цвет заполнения (обычно определяемые предыдущим обращением к функции *setfillstyle*). Аргументы *x* и *y* определяют координаты "точки привязки", то есть точки, с которой может начинаться заполнение. Если в нашем распоряжении есть только Турбо Си версии 1.5, то заполнение окружности может быть выполнено лишь при использовании функции *floodfill*, либо путем аппроксимации окружностей правильными многоугольниками (см. параграф 2.7) и использованием функции *fillpoly*. Теперь, при наличии функции *fillellipse* в версии 2.0, нам не обязательно применять функцию *floodfill* для окружностей.

В руководстве по Турбо Си говорится:

"Применяйте функцию *fillpoly* вместо функции *floodfill* везде, где только возможно, тем самым будет обеспечена совместимость с будущими версиями".

Там также упоминается, что функция *floodfill* не работает с драйвером для монитора типа IBM-8514.

При применении способа "заливки" всегда нужно быть очень осторожными. Процесс заливки распространяется во все стороны, пока не встретится цвет, записанный в переменной *border* ("*граница*"). Поэтому очень существенно, чтобы начальная точка (x, y) располагалась внутри области, ограниченной кривой с цветом, соответствующим значению переменной *border* (возможные целочисленные значения для переменной *border* приведены в параграфе 1.3). Иногда область кажется "замкнутой", но на самом деле это может оказаться неверным, поскольку где-то может оказаться пропущенным один пиксел. В этом случае произойдет "затопление", то есть процесс заполнения не будет завершен корректно и будет разрушено все изображение на экране. Поэтому мы не будем обсуждать примеров применения функции *floodfill*, ограничившись функциями *fillpoly* и *fillellipse*.

2.7. ОКРУЖНОСТИ И ПРАВИЛЬНЫЕ МНОГОУГОЛЬНИКИ

Окружности, аппроксимируемые многоугольниками

Поскольку функции *circle* и *circle_uc*, описанные в параграфе 2.2 практически удовлетворяют всем требованиям, предъявленным к вычерчиванию окружностей, может показаться ненужным обсуждать аппроксимацию окружностей фактически с помощью только правильных многоугольников. Однако для этого есть причины. Во-первых, запись в режиме "исключительное ИЛИ", описанном в параграфе 1.4, неприменима для только что упомянутых функций вычерчивания окружностей. Поскольку при аппроксимации применяются функции вычерчивания отрезков прямых линий, то можно инвертировать все пикселы любых окружностей, аппроксимируемых отрезками прямых линий. Другая причина будет раскрыта в главе 3, где будет описана методика сохранения графических результатов в файлах.

Самый прямой путь вычерчивания правильного многоугольника, содержащего, например, 80 вершин (который иногда может называться восьмидесятиугольником), заключается в применении функции *cos* и *sin* для вычисления координат этих вершин, как в программе:

```
void circle80slow (float xC, float yC, float r)

    /*      Окружность, аппроксимируемая 80-угольником          */
    /*      (предварительная версия)                            */

{   int i;
    float delta = 2*PI /80, theta = 0;

    move (xC + r, yC);

    for (i=1; i<=80; i++)
    {   theta += delta;
        draw (xC + r * cos(theta), yC + r * sin(theta));
    }
}
```

Конечно, значение 80 выбрано для числа вершин совершенно произвольно. "Окружность" будет вычерчена быстрее, если уменьшить это число, но тогда ухудшится аппроксимация, особенно в случае больших окружностей.

Другой источник повышения скорости работы, не имеющий отношения к качеству аппроксимации, основан на применении хорошо известных тригонометрических соотношений:

$$\cos(\theta + \delta) = \cos\theta\cos\delta - \sin\theta\sin\delta$$
$$\sin(\theta + \delta) = \sin\theta\cos\delta + \cos\theta\sin\delta$$

Эти соотношения позволяют вычислять все требуемые значения $\cos\theta$ и $\sin\theta$ без использования соответствующих значений θ, не обращаясь к сравнительно долго работающим функциям *cos* и *sin*. Единственными значениями, которые должны быть известны вначале, являются $\cos\delta$ и $\sin\delta$, которые могут быть записаны в программе в виде констант (опять используя число $\delta = 2\pi/80$). Функция *circle80faster* будет работать значительно быстрее, чем функция *circle80slow*, а также обеспечивает возможность заполнения полигона текущим шаблоном заполнения и текущим цветом. Заполнение происходит в том случае, если четвертый аргумент, *fill*, будет ненулевым.

```
#define COSDELTA 0.996917333733
#define SINDELTA 0.078459095728

void circle80faster(float XC, float YC, float r, int fill)

        /* Окружность аппроксимируется правильным 80-угольником */
        /* (улучшенная, но пока еще предварительная версия)      */

{   double costh=1.0, sinth=0.0, c0, s0;
    int j=0, XC, YC, H, V, points[162];

    XC = IX(xC);    YC = IY(yC);

    for (j=0; j<=40; j+=2)
    {   H = XPIX(r * costh);
        V = YPIX(r * sinth);
        points[j] = XC + H; points[j+1] = YC + V;
        if (j) { points[160-j] = XC+H;  points[161-j] = YC-V; }

        if (j != 40)
        {   points[80-j] = XC-H; points[81-j] = YC+V;
            if (j) { points[80+j] = XC-H; points[81+j] = YC-V;}
        }

        c0 = costh; s0 = sinth;
        costh = c0 * COSDELTA - s0 * SINDELTA;
        sinth = s0 * COSDELTA + c0 * SINDELTA;
    }

    points[160] = points[0]; points[161] = points[1];
    if (fill) fillpoly(81, points); else drawpoly(81, points);
}
```

Хотя функция *circle80faster* работает гораздо быстрее, чем функция *circle80slow*, ее можно значительно улучшить. В ее настоящей форме мы используем арифметику с плавающей точкой для вычисления элементов массива *points* (“*точки*”) для каждой вычерчиваемой окружности. Однако каждую окружность можно сформировать из некоторой “стандартной окружности” с фиксированным радиусом и с центром в точке (0, 0).

Поэтому возможно применить некоторый другой массив, назовем его *stdpoints*, с данными для этой стандартной окружности. Сделаем это только однажды для первой вычерчиваемой окружности. Язык Си представляет для этого прекрасную возможность, а именно концепцию статических переменных (*static*). Составим две версии функций, одну быструю *circle80_pc*, основанную на пикселных координатах, другую — удобную *circle80_uc* — с применением пользовательских координат.

```
#define COSDELTA 0.996917333733
#define SINDELTA 0.078459095728

void circle80_pc(int XC, int YC, int R)

        /* Окружность аппроксимируется правильным 80-угольником   */
        /* в системе пикселных координат                          */

{   static int first=1, stdpoints[162];
    int j=0, H, V, points[162];
    double costh, sinth, c0, s0;
    float x, y;

    if (first)
    {   first = 0;
        costh = 1.0; sinth = 0.0;
        for (j=0; j<=40; j+=2)
        {   H = (int)(1e4 * costh + 0.5);
            V = (int)(1e4 * sinth * vertfact/horfact + 0.5);
            stdpoints[j] = H; stdpoints[j+1] = V;
            if (j) {   stdpoints[160-j] = H; stdpoints[161-j] = -V;}
            if (j != 40)
            {   stdpoints[80-j] = -H; stdpoints[81-j] = V;
                if (j)
                {   stdpoints[80+j] = -H;
                    stdpoints[81+j] = -V;
                }
            }
            c0 = costh; s0 = sinth;
            costh = c0 * COSDELTA - s0 * SINDELTA;
            sinth = s0 * COSDELTA + c0 * SINDELTA;
        }
        stdpoints[160] = stdpoints[0];
        stdpoints[161] = stdpoints[1];
    }
    for (j=0; j<=160; j+=2)
    {   points[j] = XC + (int)(stdpoints[j]*(long)R/10000L);
        points[j+1] = YC + (int)(stdpoints[j+1]*(long)R/10000L);
        If (j) invertpixel(points[j], points[j+1]);
            /* Необходимо в случае режима XOR_PUT      */
            /* Necessary in case of XOR_PUT            */
    }
    If (fplot == NULL) drawpoly(81, points);
    else
    {   for (j=0; j<=80; j++)
        {   x = points[2*j]/horfact;
            y = (Y_max - points[2*j+1])/vertfact;
            If (j==0) move(x, y); else draw(x, y);
        }
    }
}
```

```
void circle80_uc(float xC, float yC, float r)

    /*      Окружность аппроксимируется правильным 80-угольником      */
    /*      в системе пользовательских координат                      */

{   circle80_pc(IX(xC), IY(yC), XPIX(r));
}
```

Как уже говорилось в начале этого параграфа, нам бы хотелось, чтобы приведенные здесь функции были "чувствительными к режиму записи", то есть после обращения

```
setwritemode(XOR_PUT)
```

обе наши новые функции вычерчивания окружностей должны инвертировать пикселы на вычерчиваемой окружности, чтобы при двойном обращении к функции окружность сначала появлялась на экране, а затем стиралась. К сожалению, если две соседние стороны AB и BC полигона вычерчиваются в режиме записи XOR_PUT ("исключительное ИЛИ"), то вершина B используется дважды, что приводит к ее появлению и исчезанию (или наоборот). В результате мы получим полигон без пикселов в его вершинах. Для устранения этого дефекта в функции *circle80_pc* производится дополнительное обращение к точкам вершин — перед вызовом функции *drawpoly* для всех вершин вызывается функция *invertpixel* для предварительного инвертирования пикселов. При нормальном режиме записи эта операция является излишней (хотя и не приносящей вреда), поскольку точки вершин также будут вычерчены при последующем обращении к функции *drawpoly* для вычерчивания полигона. Однако при режиме "исключительное ИЛИ" (XOR_PUT) все точки вершин будут заданы трижды, что дает этот эффект, как если бы они были заданы однажды. Функция *circle80_pc* используется в программе BBALL — простой программе для получения *движущихся* изображений. В этой программе моделируется движение отскакивающего мячика, который перемещается по прямой линии внутри прямоугольника под углом 45° к сторонам этого прямоугольника, отскакивая от его граней подобно настоящему мячику. Программа расширена таким образом, что она может изображать любое число последовательных позиций мячика. Так что траектория движения мячика изображается в виде рваной линии, которая сначала растет до заданной длины, а затем по мере роста в начале линии она усекается в конце. Поскольку пикселы

инвертируются, то может возникнуть необычный эффект в месте пересечения двух линий, где обе окружности совпадают: вторая окружность будет стирать первую, что показано на рис. 2.15. Однако, когда одна из пересекающих линий исчезает, то другая окружность восстанавливается, поэтому такой дефект является лишь временным. Иллюстрации в книге статичные, поэтому на них нельзя показать движение объекта по экрану. Но на рис. 2.15 можно все-таки видеть как бы моментальный снимок экрана: в левой части экрана можно заметить очень небольшую часть окружности, вычерчиваемой в данный момент. В следующей главе мы обсудим операцию "фиксации экрана" более подробно.

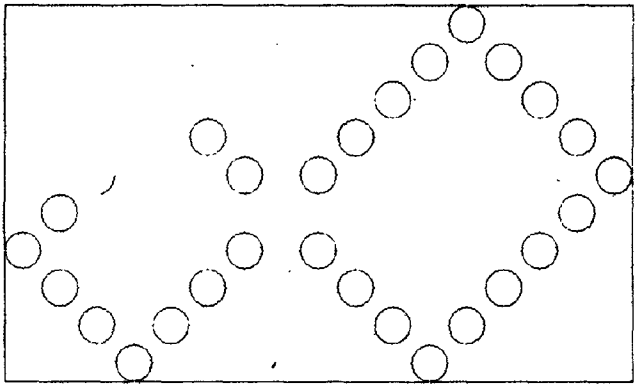


Рис. 2.15. Позиции отскакивающего мячика

2.8. ЛИНИИ ЛЮБОЙ ТОЛЩИНЫ

В Турбо Си имеется возможность вычерчивания нескольких типов линий с помощью функции *setlinestyle*, объявленной в файле GRAPHICS.H:

```
void far setlinestyle (int linestyle, unsigned upattern, int thickness);
```

Первый аргумент, *linestyle* ("тип линии"), может принимать одно из пяти возможных значений:

Значение	*Символическая константа*	*Тип линии*
0	*SOLID_LINE*	сплошная
1	*DOTTED_LINE*	пунктирная
2	*CENTER_LINE*	центровая
3	*DASHED_LINE*	штриховая
4	*USERBIT_LINE*	пользовательская

Второй аргумент *upattern* ("шаблон пользователя") используется только в том случае, если первый аргумент равен *USERBIT_LINE*. В иных случаях он игнорируется. (Заметим, что в данном случае на этом месте должно быть указано некоторое значение, например, 0). Если первый аргумент имеет значение *USERBIT_LINE*, то 16 бит из переменной *upattern* используются в качестве шаблона. Для каждого единичного бита в переменной *upattern* соответствующему пикселу будет приписан цвет изображения и шаблон будет повторяться, если длина линии превышает 16 пикселов.

Третий аргумент *thickness* содержит информацию о том, какова будет толщина линии. К сожалению имеются только две возможности для этого аргумента:

Значение	*Символическая константа*	
1	*NORM_WIDTH*	(ширина в 1 пиксел)
3	*THICK_WIDTH*	(ширина в 3 пиксела)

Обращение к функции *setlinestyle* относится ко всем последующим обращениям к функциям Турбо Си, которые вычерчивают отрезки прямых линий, окружности, дуги и т.д.

Очень толстые линии с закругленными концами

В Турбо Си нет встроенных средств для вычерчивания линий с толщиной более трех пикселов. Очень интересно и полезно написать такую функцию, которая может вычерчивать сплошную линию любой толщины. При решении этой задачи нужно будет специально обратить внимание на концевые точки каждого отрезка. Это имеет очень важное значение, особенно если концевая точка одного отрезка является начальной точкой другого отрезка, а это очень частый случай. В таких ситуациях было бы неправильно вычерчивать отрезок прямой AB с толщиной d просто как прямоугольник с длиной AB и шириной d, как показано на рис. 2.16. Результат игнорирования проблемы концевых точек можно видеть на рис. 2.17 на самом крайнем слева объекте.

Элегантное решение этой проблемы основано на идее вычерчивания заполненных полукругов с центрами в концевых точках (A или B) с радиусом $d/2$. (Эта идея была предложена автору в письме Никиты Андреева из фирмы Consul Tek, Inc., San Carlos, CA, USA). При наличии функции *fillcircle_uc* задача значитель-

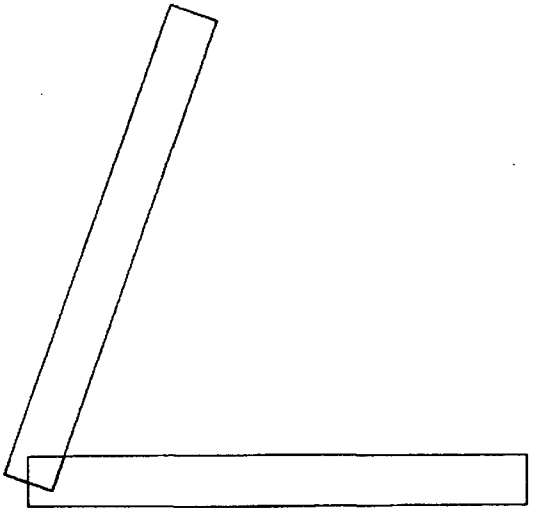


Рис. 2.16. Некорректное соединение

Рис. 2.17. Толстые линии трех типов

но упрощается. В программу включены обращения к функции *fatline*, в которой использован этот принцип. На центральном объекте рис. 2.17 можно видеть закругленные углы, полученные таким образом. Имеется также функция *fatline0*, которая осуществляет вычерчивание заполненного прямоугольника симметрично расположенного по обе стороны относительно заданного

отрезка прямой линии P_1, P_2. Длина этого прямоугольника равна длине отрезка прямой линии P_1P_2. Функция *fatline*, после вызова функции *fatline0*, вычерчивает два заполненных полукруга с центрами в точках P_1 и P_2. Заметим, что способ, каким функция *fatline* вычерчивает отрезки толстых линий, не зависит от других отрезков, которые могут примыкать к данному отрезку в его концевых точках. Но в следующем параграфе будет описана иная методика.

Очень толстые линии с острыми углами

Рис. 2.17 был получен с помощью программы FATDEMO, приведенной в конце данного параграфа. Она формирует три четырехугольника, вычерчиваемых толстыми линиями. Крайняя слева неправильная фигура получена в результате использования только одной функции *fatline0*. Фигура в центре была вычерчена с помощью функции *fatline*, которая дает закругленные соединения. Фигура справа с острыми углами в местах соединения является предметом обсуждения в данном параграфе. Чтобы получить набор функций, которые было бы просто использовать, запишем в память координаты концевых точек, заданных в качестве аргументов в обращении к функции *fatline0* для использования при последующих обращениях к новой функции *sharpjoint* (это поможет упростить повторную пересылку координат x и y трех концевых точек). Как показано на рис. 2.18, в задачу функции *sharpjoint* входит вычерчивание и заполнение четырехугольника DEBF. Кроме значения $d = 2r$, заданного в качестве аргумента, эта функция может использовать x и y координаты трех точек A, B и C, записанных ранее в глобальных переменных. (Заметим, что мы случайно реализовали эту идею путем использования глобальных статических переменных в нашем графическом модуле и таким образом они непосредственно могут быть доступны в любой пользовательской программе и нет опасности в их случайном изменении).

Если векторы BA и BC поделить на их длину и результату присвоить знак минус, то получим два очень полезных единичных вектора:

$$\mathbf{m} = [m_1 \quad m_2] = -\mathbf{BA}/\mathbf{BA}$$
$$\mathbf{n} = [n_1 \quad n_2] = -\mathbf{BC}/\mathbf{BC}$$

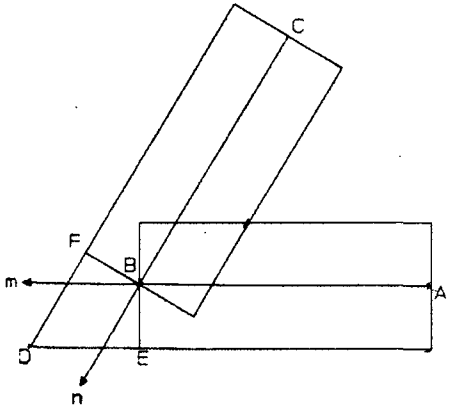


Рис. 2.18. Четырехугольник DEBF

Угол β в точке B между прямыми линиями BA и BC равен углу между этими векторами **m** и **n**, и поэтому его можно вычислить из скалярного произведения

$$\mathbf{m} \cdot \mathbf{n} = \cos\beta$$

Для наших целей нам нужны не сам угол β или его косинус $\cos\beta$, а значение синуса угла $\sin\beta$, который можно получить непосредственно из векторного произведения

$$|\mathbf{m} \times \mathbf{n}| = \sin\beta$$

(Векторное произведение мы рассмотрим позже, когда будем обсуждать, будут или нет точки A,B,C, именно в этом порядке, располагаться против движения часовой стрелки). После нахождения синуса угла $\sin\beta$ мы сможем вычислить

$$\mathbf{BD} = (r / \sin\beta)(\mathbf{m} + \mathbf{n})$$

где значение r равно половине заданной ширины линии w. Добавление этого вектора к координатам точки B даст нам положение точки D (см. рис. 2.18).

Аналогичным образом найдем положение точек E и F с помощью векторов **BE** и **BE**. Вектор **BE** имеет длину r и он перпендикулярен единичному вектору $\mathbf{m} = [m_1 \ m_2]$. Теперь мы столкнемся с такой проблемой: оба вектора $[m_2 - m_1]$ и $[-m_2 \ m_1]$ являются единичными векторами, перпендикулярными вектору **m** (что можно проверить вычислением скалярного произведения

этих векторов с вектором **m**). Какой из них выбрать в нашем случае? Здесь следует принять во внимание ориентацию расположения трех точек A, B и C. Если эти три точки располагаются по движению против часовой стрелки (см. ниже), то будем иметь

$$\mathbf{BE} = r\,[-m_2 \quad m_1], \quad \mathbf{BF} = r\,[n_2 \quad -n_1]$$

в противном случае

$$\mathbf{BE} = r\,[m_2 \quad -m_1], \quad \mathbf{BF} = r\,[-n_2 \quad n_1]$$

(Для вычисления этих векторов можно использовать матрицу поворота с размерностью 2×2 вокруг точки B на углы 90° и −90°). Таким образом остается единственная проблема определения ориентации точек A, B и C. Для этого снова воспользуемся векторами **m** и **n**. Последовательность точек { B, A, C } располагается в направлении против часовой стрелки (как на рис. 2.18), если, и только если, поворот вектора **m** вокруг точки B на угол β (где $0 < \beta < 180°$) приводит к его совпадению с вектором **n**, а это тот случай, когда векторное произведение $\mathbf{m} \times \mathbf{n}$ имеет положительный знак. В правосторонней системе координат при расположении векторов **m** и **n** в плоскости xy, это векторное произведение по направлению совпадает с осью z (а его длина равна $\sin\beta$). Более точно, мы имеем

$$\mathbf{m} \times \mathbf{n} = s\mathbf{k}$$

где **k** — единичный вектор в направлении положительной оси z и

$s = \sin\beta > 0$, если точки A,B,C, именно в этом порядке, расположены по движению против часовой стрелки;

$s = -\sin\beta < 0$, если точки A,B,C, именно в этом порядке, расположены по часовой стрелке.

Если **i** и **j** — единичные векторы в направлениях положительных осей x и y соответственно, то можно использовать запись в форме детерминанта

$$\mathbf{m} \times \mathbf{n} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ m_1 & m_2 & 0 \\ n_1 & n_2 & 0 \end{vmatrix}$$

Отсюда следует, что

$$s = m_1 n_2 - n_1 m_2$$

Если значение *s* положительно, то точки A,B,C расположены против часовой стрелки, в противном случае они расположены по движению часовой стрелки. Более подробно об этом написано в книге автора "*Принципы программирования в машинной графике*" и, конечно, в учебниках по математике. После таких несложных математических выкладок можно обратить внимание на методику применения функции *sharpjoint*. Если нам нужно получить остроконечное соединение в точке B двух отрезков прямых линий AB и BC с шириной *w*, то можно записать

```
fatline0 (xA, yA, xB, yB, w);
fatline0 (xB, yB, xC, yC, w);
sharpjoint (w);
```

Поскольку в функции *sharpjoint* используются некоторые глобальные переменные, то, в частности, очень важно записать эти три обращения к функциям именно в указанном порядке. Не нужно беспокоиться об ориентации трех точек A,B,C против часовой стрелки или по часовой стрелке. В любом случае функция *sharpjoint* сработает правильно.

Функция для вычерчивания толстых линий может оказаться полезной для различных целей, поэтому добавим к графическому модулю CRASPTC.C следующий текст и объявим новые функции в файле CRASPTC.H:

```
static float xA, yA, xB, yB, xC, yC;

void fatline0(float x1, float y1, float x2, float y2, float d)
{   float dx, dy, dxhw, dyhw, factor, r=d/2;
    int points[10];

    dx = x2 - x1; dy = y2 - y1;
    factor = r/sqrt(dx * dx + dy * dy);
    dxhw = dy * factor;     /* Приращение по оси x для половины ширины*/
    dyhw = dx * factor;     /* Приращение по оси y для половины ширины*/

    points[0] = IX(x1+dxhw); points[1] = IY(y1-dyhw);
    points[2] = IX(x2+dxhw); points[3] = IY(y2-dyhw);
    points[4] = IX(x2-dxhw); points[5] = IY(y2+dyhw);
    points[6] = IX(x1-dxhw); points[7] = IY(y1+dyhw);
    points[8] = points[0]; points[9] = points[1];
    fillpoly(5, points);

    xA = xB; yA = yB;
    xB = x1; yB = y1;
    xC = x2; yC = y2;
}
```

```
void fatline (float x1, float y1, float x2, float y2, float d)
{   float r=d/2;
    fatline0(x1, y1, x2, y2, d);
    fillcircle_uc(x1, y1, r);
    fillcircle_uc(x2, y2, r);
}

void sharpjoint(float d)
{   float r=d/2, BA, BC, dxBA, dyBA, dxBC, dyBC, s,
        sina, factor, xD, yD, xE, yE, xF, yF, m1,
        m2, n1, n2, rm1, rm2, rn1, rn2;
    int points[10];
    dxBA = xA - xB; dyBA = yA - yB;
    dxBC = xC - xB; dyBC = yC - yB;
    BA = sqrt(dxBA * dxBA + dyBA * dyBA);
    BC = sqrt(dxBC * dxBC + dyBC * dyBC);
    n1 = -dxBA/BA; n2 = -dyBA/BA;
    m1 = -dxBC/BC; m2 = -dyBC/BC;
    s = m1 * n2 - n1 * m2;
    sina = fabs(s);
    factor = r/sina;
    xD = xB + factor * (m1 + n1);
    yD = yB + factor * (m2 + n2);
    rm1 = r * m1; rm2 = r * m2;
    rn1 = r * n1; rn2 = r * n2;
    if (s > 0)
    {   xE = xB - rm2; yE = yB + rm1;
        xF = xB + rn2; yF = yB - rn1;
    }   else
    {   xE = xB + rm2; yE = yB - rm1;
        xF = xB - rn2; yF = yB + rn1;
    }
    points[0] = IX(xB); points[1] = IY(yB);
    points[2] = IX(xF); points[3] = IY(yF);
    points[4] = IX(xD); points[5] = IY(yD);
    points[6] = IX(xE); points[7] = IY(yE);
    points[8] = points[0]; points[9] = points[1];
    fillpoly(5, points);
}
```

Теперь можно составить следующую демонстрационную программу, которая после компоновки с модулем CRASPTC.OBJ дает в результате изображение, показанное на рис. 2.17.

```
/* FATDEMO:                                                    */
/*  Демонстрационная программа черчения трех типов толстых линий */

#include <stdio.h>
#include <math.h>
#include "grasptc.h"
```

```
main( )
{   float xP, yP, xQ, yQ, xR, yR, xS, yS, c, c2, d;
    initgr( );
    c = x_max/7; c2 = 2*c; d = c/2;
    xP = xS = c; xQ = xR = c2;
    yP = yQ = c; yR = y_max - c;
    yS = 0.5 * (yP + yR);
    fatline0(xP, yP, xQ, yQ, d);
    fatline0(xQ, yQ, xR, yR, d);
    fatline0(xR, yR, xS, yS, d);
    fatline0(xS, yS, xP, yP, d);

    xP += c2; xQ += c2; xR += c2; xS += c2;
    fatline(xP, yP, xQ, yQ, d);
    fatline(xQ, yQ, xR, yR, d);
    fatline(xR, yR, xS, yS, d);
    fatline(xS, yS, xP, yP, d);

    xP += c2; xQ += c2; xR += c2; xS += c2;
    fatline0(xP, yP, xQ, yQ, d);
    fatline0(xQ, yQ, xR, yR, d); sharpjoint(d);
    fatline0(xR, yR, xS, yS, d); sharpjoint(d);
    fatline0(xS, yS, xP, yP, d); sharpjoint(d);
    fatline0(xP, yP, xQ, yQ, d); sharpjoint(d);
    endgr( );
}
```

2.9. ДЕЛОВАЯ ГРАФИКА

Статистические данные часто представляются в форме диаграмм, на которых определенные числовые данные пропорциональны заштрихованным областям. В Турбо Си есть четыре функции для простого получения подобных диаграмм. Они объявлены в файле CRAPHICS.H следующим образом:

```
void  far bar(int left, int top, int right, int bottom);
void  far bar3d(int left, int top, int right, int bottom,
                                    int depth, int topflag);
void  far pieslice(int x, int y, int stangle, int endangle, int radius);
void  far sector( int x, int y, int stangle, int endangle,
                                    int xradius, int yradius );
```

Программа BARS формирует столбчатую диаграмму, показанную на рис. 2.19. Функция *bar* выполняет заполнение прямоугольника текущим шаблоном с текущим цветом. Но она не очерчивает границы этого прямоугольника. Если требуется вычертить контур прямоугольника, то, как увидим ниже, можно будет воспользоваться функцией *bar3d*.

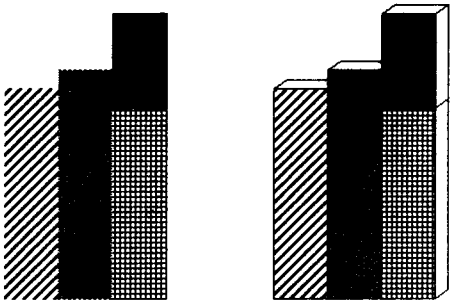


Рис.2.19. Столбчатые диаграммы

Функция *bar3d* вычерчивает трехмерные вертикальные столбики, передняя грань которых заполняется текущим шаблоном с текущим цветом. Вычерчиваются только видимые ребра столбика. "Глубина" столбика задается в качестве пятого аргумента. Каждый столбик может состоять более, чем из одной части. При значении параметра *topflag* = 1 вычерчивается верхняя грань столбика, как очевидно, это имеет место только для самой верхней части столбика, а для всех более низких частей необходимо задавать *topflag* = 0. Теперь может быть ясно, что функцию *bar3d* можно использовать со значением параметра *depth* = 0 (и с любым значением параметра *topflag*) для получения заполненного прямоугольника, аналогичного получаемому с помощью функции *bar*, но с вычерчиванием границ прямоугольника.

```
/* BARS:                                                          */
/*        Двух- и трехмерные диаграммы для деловой графики        */

#include "grasptc.h"

main()

{   int w, h, bottom, depth;
    initgr();
    w = X_max/30;            /* Единицы по горизонтали      */
    h = Y_max/30;            /* Единицы по вертикали        */

    bottom = 20 * h; depth = w/2;
    setfillstyle(SLASH_FILL, foregrcolor);
    bar(6*w, 9*h, 8*w, bottom);                /* Столбик 1   */
    bar3d(16*w, 9*h, 18*w, bottom, depth, 1);
    setfillstyle(INTERLEAVE_FILL, foregrcolor);
    bar(8*w, 8*h, 10*w, bottom);               /* Столбик 2   */
    bar3d(18*w, 8*h, 20*w, bottom, depth, 1);
```

```
    setfillstyle(HATCH_FILL, foregrcolor);
    bar(10*w, 10*h, 12*w, bottom);                  /* Столбик 3А  */
    bar3d(20*w, 10*h, 22*w, bottom, depth, 0);
    setfillstyle(SOLID_FILL, foregrcolor);
    bar(10*w, 5*h, 12*w, 10*h);                     /* Столбик 3Б  */
    bar3d(20*w, 5*h, 22*w, 10*h, depth, 1);
    endgr();
}
```

Функция *pieslice* (“*кусок пирога*”) вычерчивает заполненный сектор круга, а функция *sector* вычерчивает заполненный сектор эллипса. Программа SECTORS демонстрирует работу обеих этих функций и результат их выполнения показан на рис. 2.20.

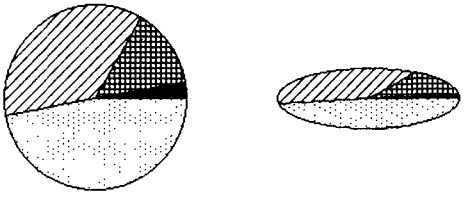


Рис. 2.20. Круговые и эллиптические секторы

```
/* SECTORS:                                                          */
/*        Круговые и эллиптические секторы для деловой графики.      */

#include "grasptc.h"

main()

{   int XC, YC, XE, YE, R, H;
    initgr();
    XC = X__max/3; XE = 2*X__max/3, R = (XE - XC)/3; H = R/3;
    YC = YE = Y__max/2;

    setfillstyle(SOLID_FILL, foregrcolor);
    pieslice(XC, YC, 0, 10, R);                     /* Сектор 1 */
    sector(XE, YE, 0, 10, R, H);
    setfillstyle(HATCH_FILL, foregrcolor);
    pieslice(XC, YC, 10, 60, R);                    /* Сектор 2 */
    sector(XE, YE, 10, 60, R, H);
    setfillstyle(LTSLASH_FILL, foregrcolor);
    pieslice(XC, YC, 60, 190, R);                   /* Сектор 3 */
    sector(XE, YE, 60, 190, R, H);
    setfillstyle(WIDE_DOT_FILL, foregrcolor);
    pieslice(XC, YC, 190, 360, R);                  /* Сектор 4 */
    sector(XE, YE, 190, 360, R, H);
    endgr();
}
```

Глава 3

ФОРМИРОВАНИЕ ГРАФИЧЕСКОГО ПРОДУКТА

3.1. ВВЕДЕНИЕ

Было бы очень мало пользы от нашей деятельности, если бы графические результаты можно получать только в виде изображений на экране монитора компьютера. В течение многих лет наиболее распространенными устройствами для получения чертежей с помощью компьютера были перьевые графопостроители (плоттеры), но сейчас наблюдается тенденция использовать для этой цели принтеры. Обычно принтеры применяются для вывода текстовой информации, а если на него можно будет выводить и графическую информацию, то перед выводом твердой копии можно сформировать смешанную текстовую и графическую информацию. Кроме принтера для этой цели нам потребуется еще программный пакет "*настольного издательства*". Эта книга посвящена программированию графических задач и в предыдущих книгах автора совершенно не затрагивалось иное стандартное программное обеспечение, отличное от ДОС и некоторых компиляторов с языка Си. Будучи программистами, нам интереснее иметь дело с исходными текстами программ, а не с коммерческими программными пакетами, которые обычно доступны только в виде выполняемых модулей. Следуя этому принципу крайне категорично, нам бы пришлось иметь дело с командами управления принтерами самого низкого уровня для разнообразных типов применяемых принтеров и самостоятельно разрабатывать программное обеспечение для всех принтеров. Пример такого программного обеспечения для матричного принтера с девятью иголками имеется в книге автора "*Машинная графика на*

персональных компьютерах". Теперь при значительном распространении матричных принтеров с 24 иголками и лазерных принтеров разработка базового программного обеспечения принтеров для широкого класса пользователей больше не представляет интереса в книгах, подобных данной. Также совершенно бесполезно разрабатывать программные средства для вывода на принтер только графической информации, если нам действительно необходимо вставлять картинки в текстовые документы. Выполнение такой работы с помощью собственноручно написанных программ означает, что нам пришлось бы написать полную программу для настольного издательства, что очевидно значительно выйдет за рамки данной книги.

Теперь, при наличии широко доступных инструментальных средств высокого уровня, почему бы не воспользоваться ими?

Хотя существует несколько хороших текстовых редакторов и программных пакетов для настольных издательств, воспользуемся в качестве примера пакетом WordPerfect версии 5.0. Упоминание номера версии в данном случае имеет очень существенное значение, поскольку в версии 4.2 имелся только текстовый редактор без каких-либо графических средств, тогда как на версии 5.0 можно выполнять все работы, связанные с настольным издательским делом. Этот пакет можно использовать для включения в документ графических результатов, полученных при работе наших программ, и затем послать этот результат на принтер. Автор применил этот способ при подготовке рукописи данной книги, включая иллюстрации, с выводом всего материала книги на лазерный принтер Hewlett-Packard LaserJet II. Если читатель, например, знаком с пакетом Ventura Publisher, то может показаться, что пакет WordPerfect не обеспечивает всех возможностей, которые ожидаются от настольной издательской системы, но, с другой стороны, если читатель уже пользуется пакетом WordPerfect для ввода текста документов достаточно большого объема, то будет очень полезно, если на окончательном этапе читателю не придется переключаться на другой пакет, чтобы получить желаемый результат. Это означает, что можно использовать имеющиеся в пакете WordPerfect прекрасные средства для редактирования текста, такие как поиск и замена команд и макросов, не только на подготовительной стадии, но и позднее, когда приходится редактировать и вносить исправления.

Существуют два способа записи графической информации в файлах, каждый из которых имеет свои положительные и отрицательные стороны. Мы их будем обозначать разными терминами: графика с *битовым отображением* и *векторная* графика. Предположим на данный момент, что такие файлы доступны. Тогда нам необходимо знать, как их можно применить. Этот вопрос и обсудим в начале главы.

3.2. ИСПОЛЬЗОВАНИЕ ГРАФИКИ В WORDPERFECT 5.0

Есть много хороших книг с описанием пакета WordPerfect, и здесь мы не будем их повторять. Обсудим только графические возможности текстового процессора, поскольку они нужны нам для графики на Турбо Си. В действительности пакет WordPerfect очень полезен, если известен принцип построения чего-то, но мы не помним этого в деталях. В большинстве случаев нам достаточно просто нажать "клавишу помощи", то есть функциональную клавишу F3, а затем клавишу с первой буквой интересующей нас темы, например, если такой темой будет "графика" ("graphics"), то надо будет нажать F3-G. На экране появится сообщение, что ключевой комбинацией, используемой для графики, будет комбинация Alt-F9. Если применить ее, то на экране будет изображено однострочное меню, из которого мы выберем:

1 Figure

нажатием на клавишу 1. Это приведет к появлению другого меню, из которого выберем:

1 Create

Затем появится следующее меню, первая из возможностей которого будет:

1 — Filename

так что снова нам нужно нажать на клавишу 1, после чего появится запрос на ввод имени графического файла, который будет использоваться. Вскоре мы обсудим, как получить такие файлы, но пока отметим, что после выбора графического файла можно модифицировать его размер и, если это файл с битовым отображением, то можно инвертировать все его пикселы, такая возможность, как правило, очень желательна. При выводе содержи-

мого файла на матричный или лазерный принтер (для чего должна быть введена комбинация Shift-F7 и затем нажата клавиша 1, если требуется напечатать полный документ) картинка печатается вместе с любым текстом, который также присутствует в файле пакета WordPerfect. Но перед выводом на печать настоятельно рекомендуется предварительно "просмотреть" весь документ, используя комбинацию Shift-F7 с последующим нажатием клавиши 6.

Хотя в пакете WordPerfect имеется возможность считывания графических файлов в нескольких форматах, мы будем применять только два, а именно .WPG и .HPG. Они представляют два различных класса: графику с *битовым отображением*, описываемую в данном и следующем параграфах, и *векторную* графику, которую рассмотрим в параграфе 3.4.

Графика с битовым отображением (файлы с расширением .WPG)

Мы будем относиться к графике с битовым отображением как к набору пикселов на экране компьютера. При записи такого набора в файл (и соответственно вывод на принтер) предполагается, что все параметры внешнего вида картинки на экране имеют соответствующую форму представления на принтере. В особенности это относится к заполненным областям, о чем уже говорилось в параграфе 2.6.

Графика с битовым отображением имеет следующие особенности:

- Если линия имеет ступенчатую форму вследствие плохой разрешающей способности, то она также будет ступенчатой и при выводе на принтер.
- Если в пакете WordPerfect рисунок будет уменьшен в размерах, то тонкие линии могут частично исчезнуть.
- Файлы записываются в некотором неизвестном формате, так что будет затруднительно, а иногда даже невозможно генерировать их с помощью собственных программ.

3.3. УТИЛИТА GRAB

Последняя ситуация в предыдущем параграфе может разочаровать нас, как программистов, но в действительности это не так

уж плохо, как кажется. В пакете WordPerfect версии 5.0 имеется утилита с именем GRAB, которая обеспечивает очень простой способ формирования файлов типа .WPG. Прежде чем запускать на выполнение нашу графическую программу на Турбо Си, следует ввести команду

```
GRAB
```

Эта программа становится резидентной в памяти компьютера и сообщение на экране информирует нас, как ее использовать. Затем, когда наша программа сформирует изображение, которое еще видимо на экране, мы можем сказать, что мы хотим "сохранить" ее, нажав комбинацию клавиш Shift-Alt-F9. После этого на экране появляется прямоугольник из штриховых линий, обозначающий границы области на экране, которая будет фактически сохранена. Этот прямоугольник можно переместить или увеличить, чтобы охватить все нужные части изображения. Перемещение в каждом из четырех направлений реализуется путем нажатия на соответствующие клавиши со стрелками. Изменить размеры этого прямоугольника можно путем нажатия на эти же клавиши со стрелками при одновременном удержании в нажатом положении клавиши Shift. Например, если нажать одновременно клавиши Shift и "стрелка вправо", то будет перемещаться вправо правая граница прямоугольника. Если же вместо этого будут нажаты Shift и "стрелка влево", то эта же граница будет перемещаться влево. В обоих случаях при удержании клавиши Shift в нажатом положении левая граница прямоугольника остается неподвижной. После выбора нужных размеров и положения прямоугольника необходимо нажать на клавишу 'Enter' ("Ввод"). Фактически только тогда и начнется процесс сохранения изображения на экране. Если эта операция выполняется впервые, то результирующий файл получит имя GRAB.WPG. Если файл с таким именем уже существует, то новый файл получит имя GRAB1.WPG, затем GRAB2.WPG и так далее.

Таким образом формирование файла с расширением .WPG оказывается очень простой задачей. Заметим, что мы используем утилиту GRAB в комбинации со своей собственной графической программой, так что по сравнению с другими графическими программами, подобным PC Paintbrush или GEM Paint, мы не лишаемся удовольствия самостоятельного программирования!

Другая привлекательная сторона заключается в доступности утилиты GRAB для всех тех, кто работает с пакетом WordPerfect 5.0 (который сейчас является очень популярным программным пакетом), поэтому нет необходимости ее покупать отдельно.

Утилита GRAB дает хорошие результаты, если картинка содержит заполненные области как, например, рис. 2.11 в параграфе 2.6. Если эта утилита будет использована для картинок с тонкими линиями, то эти линии могут частично исчезнуть при уменьшении размеров картинки. Кроме всего прочего, файл с расширением .WPG состоит из целого ряда пикселов. Это относится ко всем направлениям, так что при этом может уменьшаться не только длина, но и ширина линий. Предположим, например, что имеется горизонтальный отрезок прямой линии длиной 100 пикселов и с шириной в один пиксел. Если мы уменьшим ее путем умножения на некоторый коэффициент, скажем 0.7, тогда можно предположить, что окончательный результат будет получен при последовательном выполнении двух операций. Сначала отрезок уменьшается в горизонтальном направлении до 70 пикселов, то есть до нужного нам размера, но затем на втором шаге изображение будет уменьшено в вертикальном направлении, для которого у нас нет иной возможности, кроме выбора для толщины линии 0 или 1 пиксела. Это означает, что существует 70 процентная вероятность, что линия останется с шириной в 1 пиксел, и вероятность 30%, что она исчезнет. С вертикальными линиями ситуация аналогична — при уменьшении они также могут исчезать. У наклонных линий после уменьшения могут появиться пустые промежутки. Все это, конечно, неприемлемо, так что нежелательно уменьшать картинки, построенные на основе битового отображения и содержащие тонкие линии. Лучше всего предусмотреть, чтобы изображение на части экрана, сохраняемого утилитой GRAB было бы не больше желаемого окончательного размера. Это можно осуществить путем использования неполного экрана, когда картинка формируется нашей программой и регулировкой размера (эта операция была описана выше) до размера окончательной картинки. Другое решение заключается в том, чтобы сделать все тонкие линии и кривые "толще", чтобы их ширина составляла по крайней мере два пиксела. Такой способ будет описан более подробно в параграфе 5.4. Если же наша картинка состоит только из линий (и не содержит

заполненных областей), то существует более хороший метод, который и обсудим ниже.

3.4. ВЕКТОРНАЯ ГРАФИКА НА ОСНОВЕ HP-GL

При векторной графике картинка состоит в основном из набора отрезков прямых линий. Для вычерчивания полной картинки необходимо знать только координаты концевых точек этих отрезков. В качестве расширения в состав базовых элементов обычно включаются еще текстовые строки (с координатами начальной позиции). Основное преимущество векторной графики заключается в том, что качество линий не зависит от разрешающей способности экрана. Таким образом прямые линии будут даже улучшены, если они посылаются на принтер с более высоким разрешением, чем у экрана видеомонитора. Если мы изменим размер изображения, то линии будут только длиннее или короче, но не тоньше или толще. С другой стороны, нельзя считать, что сплошная заливка областей может быть представлена конечным числом отрезков прямых линий, поэтому задача заливки областей в векторной графике решается совсем не просто. Мы не будем обсуждать все возможные форматы, которые могут быть использованы для векторной графики, а ограничимся одним, который очень прост, но оказывается достаточным для наших целей, а именно языком с именем HP-GL, которое получено как аббревиатура английского названия "Графический язык фирмы Hewlett-Packard". (Поскольку расширение имен файлов в ДОС должно содержать не более трех букв, то это название сокращено до .HPG).

Преимущество файлов типа .HPG заключается также в том, что они записываются в хорошо известном формате ASCII и в кодах, которые мы, программисты, можем легко понять и, следовательно, сгенерировать самостоятельно. Формат HP-GL первоначально был создан для управления работой плоттеров. Некоторую информацию об использовании плоттеров можно найти в книге автора "*Интерактивная трехмерная машинная графика*". В этой книге приведена программа PLOTHP, которая главным образом преобразует закодированные операции "переместить" и "вычертить" в соответствующие команды языка HP-GL и посылает их на плоттер фирмы Hewlett-Packard. В этой книге коды операций "переместить" и "вычертить" генерируются

непосредственно в функциях *move* и *draw* в качестве побочного результата только в том случае, если глобальная переменная *fplot* (указатель на файл) не равна *NULL*. При тех же самых условиях наши новые версии функций *move* и *draw* будут делать то же самое за исключением того, что они не будут непосредственно генерировать команды языка HP-GL, а будут записывать их в файл.

Фактически мы будем использовать лишь небольшую часть команд языка HP-GL. С одной стороны, существуют такие команды языка HP-GL (например, для управления скоростью перемещения пера), которые нам совсем не нужны, и, с другой стороны, существуют такие команды, которые хотя и полезны, но не поддерживаются пакетом WordPerfect версии 5.0. Все команды, которые мы будем использовать, можно найти в следующем примере программы на языке HP-GL, которая дает в результате изображение, показанное на рис. 3.1.

```
IN;SC0,10000,0,7000,SR2,3.5;DT~;
PU;PA1000,1000;PD;PA9000,1000;
PA1000,6000;PA1000,1000;
PU;PA900,600;LBA~;
PU;PA9050,600;LBB~;
PU;PA950,6050;LBC~;
PU;PA2500,100;LBRight-angled triangle ABC~;
```

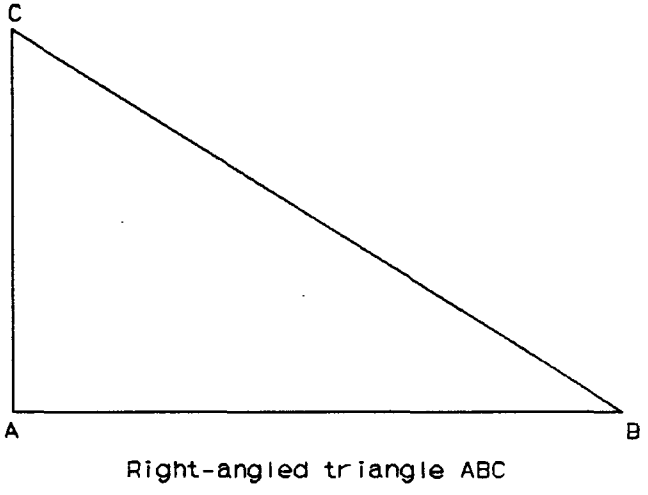


Рис. 3.1. Результат работы примера программы на языке HP-GL

Используемые в этом примере команды языка HP-GL имеют следующий смысл:

IN; "Инициализация"

SC0,10000,0,7000; "Масштабирование". Эта команда сообщает, что допустимые минимальные и максимальные значения по координате *x* будут 0 и 10 000 соответственно. Аналогично предельные значения по координате *y* будут равны 0 и 7000. В языке HP-GL координаты выражаются целыми числами, так что мы не сможем использовать наши нормальные пользовательские координаты с максимальными значениями $x_max = 10.0$ и $y_max \approx 7.0$. Поэтому эти пользовательские координаты будем умножать на 1000 и округлять до ближайшего целого числа.

SR2,3.5; "Относительный размер символов". Эта команда устанавливает ширину и высоту символов. Они выражаются в процентах от ширины и высоты всего чертежа. Здесь заданная ширина и высота символа соответствует $2/100 \times 10000 = 200$ горизонтальным и $3.5/100 \times 7000 = 245$ вертикальным единицам, установленным в команде SC.

DT~; "Определение терминатора". Здесь команда DT устанавливает, что символ ~ будет использован в команде LB в качестве терминатора (символа конца) текстовой строки. По умолчанию терминатором является символ с кодом ASCII, имеющим значение 3. Но поскольку не все редакторы и текстовые процессоры позволяют легко вводить такой символ, было бы предпочтительно использовать в программе на языке HP-GL некоторый печатаемый символ, например, (~), который можно вводить самостоятельно, понимая, что будет невозможно включать этот символ внутри строки нормальным образом. По этой причине мы не будем использовать команду DT при генерации команд языка HP-GL в нашей программе на языке Си, а будем применять терминатор по

умолчанию, который в языке Си записывается как '\003'.

PU; "Перо поднять"

PD; "Перо опустить"

PA1000,1000; "Черчение абсолютное". В зависимости от положения пера ("Поднято" или "Опущено") перо перемещается в заданную точку над поверхностью бумаги или перемещает перо из текущей точки в заданную касаясь бумаги, вычерчивая при этом отрезок прямой линии.

LBA~; "Надпись". Все символы между командой LB и символом терминатора (~) будут вычерчены в виде горизонтальной текстовой строки, начиная с текущей позиции, оставшейся после предыдущей команды PA. Фактически эта точка будет расположена несколько ниже нижнего левого угла первого символа. Если мы начертим горизонтальную прямую линию и текстовую строку с одной и той же координатой *y*, то текстовая строка появится несколько выше этой прямой линии. Ширина и высота символов будет определена в соответствии со значениями, заданными в команде SR.

3.5. ГЕНЕРАЦИЯ КОМАНД ЯЗЫКА HP-GL

Прямые линии

Расширим наш графический модуль GRASPTC так, чтобы он мог автоматически генерировать команды языка HP-GL, соответствующие тому изображению, которое мы видим на экране. Напомним, что этот графический модуль предназначен для применения программистами, работающими с проблемно-ориентированными задачами, которые интересуются прежде всего математическими или техническими аспектами их собственных программистских проблем и не желают тратить время на изучение специфических особенностей операционных систем, графических адаптеров и компиляторов. Это совсем не обязательно означает, что пользователи пакета GRASPTC являются новичками в искусстве программирования на языке Си. Язык Си не очень

сложен для изучения, в нем очень мало сложных конструкций, если они вообще есть. Поэтому автор надеется, что читатель знаком с концепцией указателей файлов, например, такого как *fplot*, определенного в модуле GRASPTC.C как

```
#include <stdio.h>
FILE *fplot=NULL;
```

Заголовочный файл GRASPTC.H, который мы всегда включаем в нашей графической программе с помощью строки

```
#include "grasptc.h"
```

содержит следующее объявление *fplot*:

```
extern FILE *fplot;
```

Это означает, что кроме включения GRASPTC.H нам нет необходимости в объявлении этого указателя файла в нашей собственной программе, если нам потребуется его использовать. Теперь очень важно помнить, что файл HP-GL получается автоматически при использовании наших известных функций *initgr*, *mode* и *draw*, если перед обращением к функции *initgr* мы изменим указатель *fplot*, имеющий по умолчанию значение *NULL*, на указатель на реальный файл, записав, например:

```
fplot = fopen ("figure.hpg", "w");
```

Хотя в принципе совсем не обязательно действительно знать, как указатель *fplot* используется в модуле GRASPTC, но с учебной точки зрения обсудим его применение, поскольку это и полезно и очень просто. В функцию *initgr* включен условный оператор *if* после вычисления значения *y_max*:

```
if (fplot != NULL)
    fprintf (fplot, "IN; SC0,10000,0,%d;\n", plotcoor (y_max));
```

Так команда HP-GL будет записана в файл, имя которого сообщено пользователем через переменную *fplot*. Если это имя не определено, то никакие команды HP-GL не будут записываться и функция *initgr* сработает как обычно.

В последней строке примера используется макроопределение *plotcoor*. В нем вычисляются координаты чертежа и оно определяется следующим образом:

```
#define plotcoor(x) ((int)(1000 * (x) + 0.5))
```

В функцию *move* включается условный оператор:

```
if (fplot != NULL)
    fprintf (fplot, "PU;PA%d,%d;", plotcoor(x),plotcoor(y));
```

Аналогично, функция *draw* содержит оператор

```
if (fplot != NULL)
    fprintf (fplot, "PD;PA%d,%d;", plotcoor(x),plotcoor(y));
```

Текст

Если указатель *fplot* не равен *NULL*, то необходимо иметь возможность записывать в выходной файл, доступный через *fplot*, команды HP-GL для текста. Но с текстом все обстоит несколько сложнее. Функции Турбо Си *outtext* и *outtextxy*, описанные в параграфе 1.6, основаны на соглашении, что по умолчанию начальная точка расположена в левом верхнем углу первого символа, тогда как HP-GL размещает текст таким образом, что текущая позиция будет находиться несколько ниже нижнего левого угла. Кроме того Турбо Си допускает выбор из нескольких доступных типов шрифтов и размеров символов, тогда как при использовании HP-GL пакет WordPerfect поддерживает только один тип шрифта — "Гельветика", который примерно соответствует шрифту "сансериф" ("без засечек") в Турбо Си.

Добавим к модулю GRASPTC функцию *grtext* для генерации текста как на экране дисплея, так и в файле HP-GL. Вызываемая в графическом режиме, она отображает текстовую строку и, если переменная *fplot* имеет значение, отличное от *NULL*, она также записывает соответствующую строку в файл HP-GL. Аргументы в функции *grtext* основаны на требованиях HP-GL, поэтому будем задавать координаты нижнего левого угла первого символа.

Аргументы этой функции таковы, что они могут быть использованы без каких-либо преобразований для генерации команд HP-GL. Но тогда они должны быть преобразованы в аргументы, необходимые для графических функций Турбо Си. Особенно это относится к размеру символов. Текст, сформированный функцией *grtext* на экране, является лишь приближенным отображением текста, записываемого в файл HP-GL, который предназначен для окончательного документа и поэтому считается более важным.

Перед обсуждением работы функции *grtext* посмотрим, как ее можно применять. Она объявлена в файле GRASPTC.H следующим образом:

```
void grtext(float xleft, float ylower, float heightpercent, char *str);
```

В первых двух аргументах указывается начальная точка текстовой строки по правилам языка HP-GL: нижний левый угол первого символа будет расположен несколько выше этой точки. Третьим аргументом *heightpercent* устанавливается высота символов, выраженная в процентах от общей высоты картинки. Строка, предназначенная для вывода, задается четвертым аргументом *str*. Функция *grtext* сама заносит код терминатора ('003') в конец строки, поэтому пользователь должен просто записывать строки нормальным образом. Программа HPGLTEXT показывает, как будет отображаться текстовая строка

```
ABCabc
```

с высотой символов в процентах

2, 4, ..., 16

Эта программа также вычерчивает горизонтальные прямые линии с той же координатой *y*, как и для текстовой строки.

```
/* HPGLTEXT:                                                    */
/*                Тестовая программа вывода текста в HP-GL      */

#include "grasptc.h"

main()

{   float x=1.0, y, h_perc;
    fplot = fopen("test.hpg", "w");
    initgr();
    y = 0.1;
    for (h_perc=2.0; h_perc<17; h_perc+=2.0)
    {   grtext(x, y, h_perc, "ABCabc");
        move(x, y); draw(8.0, y);
        y += h_perc/100.0 * y_max + 0.2;
    }
    endgr();
}
```

Если программу HPGLTEXT запустить на выполнение, то на экране получим результат, показанный на рис. 3.2.

Для получения рис. 3.2, в виде твердой копии автор применил утилиту GRAB, описанную в параграфе 3.3. После ввода файла

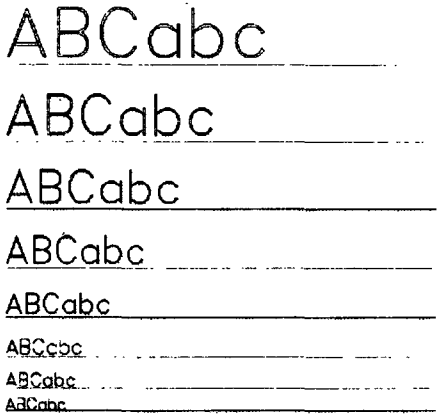


Рис. 3.2. Результат работы программы HPGLTEXT на экране

GRAB.WPG в пакет WordPerfect потребовалось инвертировать все пикселы, используя последовательность команд

Alt-F9
Figure (1)
Edit (2)
Figure number (...)
Edit (8)
Invert (4)

Путем использования стандартной функции языка Си *fopen* в программе HPGLTEXT с присвоением внешней переменной *fplot* возвращаемого ею значения, новая функция *grtext* будет также записывать команды HP-GL в файл, аналогично тому, как это делают функции *initgr*, *move* и *draw*. Этот файл с именем TEST.HPG (это имя задается в обращении к функции *fopen*) получит следующее содержимое:

```
IN; SC0,10000,0,6435;
SR1.1,2.0;PU;PA1000,100;LBABCabc^C;
PU;PA1000,100;PD;PA8000,100;
SR2.3,4.0;PU;PA1000,429;LBABCabc^C;
PU;PA1000,429;PD;PA8000,429;
SR3.4,6.0;PU;PA1000,886;LBABCabc^C;
PU;PA1000,886;PD;PA8000,886;
SR4.6,8.0;PU;PA1000,1472;LBABCabc^C;
PU;PA1000,1472;PD;PA8000,1472;
SR5.7,10.0;PU;PA1000,2187;LBABCabc^C;
PU;PA1000,2187;PD;PA8000,2187;
```

```
SR6.9,12.0;PU;PA1000,3030;LBABCabc^C;
PU;PA1000,3030;PD;PA8000,3030;
SR8.0,14.0;PU;PA1000,4003;LBABCabc^C;
PU;PA1000,4003;PD;PA8000,4003;
SR9.1,16.0;PU;PA1000,5104;LBABCabc^C;
PU;PA1000,5104;PD;PA8000,5104;
```

Как указывалось в параграфе 2.1, значение y_max зависит от типа используемого графического адаптера. Для монохромного адаптера HGA имеем: *y_max*=6.435, чем объясняется значение 6435 в первой строке файла TEST.HPG. Запись ^C здесь использована для обозначения кода конца '\003' в конце текстовых строк команд LB. Файл TEST.HPG имеет длину только 569 байт по сравнению с 5877 байтами соответствующего файла с битовым отображением GRAB.WPG. При вводе в пакет WordPerfect он даст результат, показанный на рис. 3.3.

Интересно отметить разницу между рис. 3.2 и рис. 3.3. К удивлению шрифт типа "сансериф" на рис. 3.2 оказался не битовым отображением, а более близок к "штриховому" шрифту Турбо Си. Он действительно состоит из отрезков прямых линий и выглядит привлекательным на экране дисплея. И только при выполнении операции сохранения изображения на экране мы

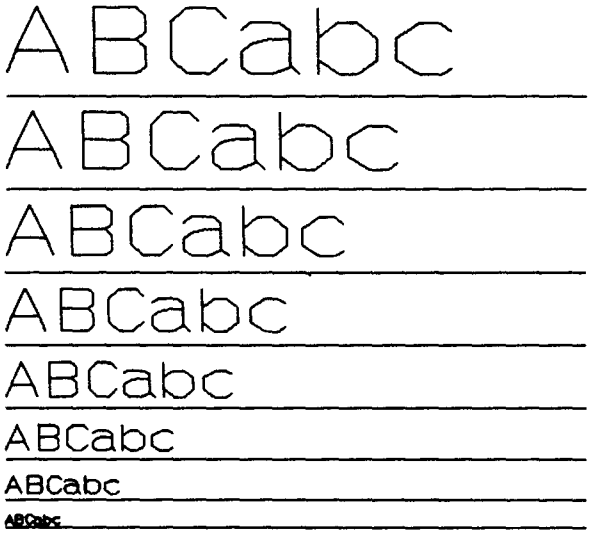


Рис. 3.3. Результат работы программы HPGLTEXT при использовании файла TEST.HGP

получим файл с битовым отображением. Содержимое этого файла зависит от разрешающей способности видеодисплея и нам следует соблюдать осторожность при изменении размера изображения. Однако на рис. 3.3 отрезки прямых линий, из которых составлены символы, остаются реальными отрезками в файле HP-GL, как они и были сформированы, и только в самый последний момент при передаче изображения на принтер отрезки прямых линий преобразуются в точки, воспроизводимые принтером.

Теперь, когда мы познакомились с применением функции *grtext*, самое время рассмотреть работу этой функции. Как читатель может видеть, заданная строка *str* копируется в локальную переменную *s*, которая имеет достаточную длину, чтобы вставить обычный в языке HP-GL код конца строки '\003' непосредственно перед завершающим нулевым символом '\0'. Как читателю известно, переменная *str* обозначает только начальный адрес действительно заданной последовательности символов и мы не имеем права изменить в памяти символы в этой последовательности, что не относится к ее длине. Константа 4.0/7.0 была найдена эмпирическим путем. Конечно, ее можно несколько изменить: чем она больше, тем шире будут символы, полученные на основе файла HP-GL.

```
void grtext(float xleft, float ylower, float heightpercent, char *str)

{   char *s;
    int len, charsize;
    float widthpercent=heightpercent*4.0/7.0;
    len = strlen(str);
    s = farmalloc(len+2);

    if (s == NULL){to_text( ); printf("farmalloc");exit(1);}
    strcpy(s, str);
    s[len] = '\003'; s[len+1] = '\0';

    if (fplot != NULL)
    {   fprintf(fplot, "SR%3.1f,%3.1f;", widthpercent, heightpercent);
        fprintf(fplot, "PU;PA%d,%d;LB%s;\n",
        plotcoor(xleft), plotcoor(ylower), s);
    }

    charsize = (int)(heightpercent/2 + 0.5);
    if (charsize > 10) charsize = 10;
    settextstyle(SANS_SERIF_FONT, HORIZ_DIR, charsize);
    outtextxy(IX(xleft), IY(ylower)-textheight(str)-charsize, str);
    farfree(s);
}
```

3.6. МОДУЛЬ GRASPTC

Перед обращениям к самим функциям из модуля GRASPTC, рассмотрим сначала их объявления (иногда они называются "прототипами функций") в хидерном файле GRASPTC.H. Настоятельно рекомендуется включать этот файл в графическую программу с помощью строки:

```
#include "grasptc.h"
```

Здесь оказывается очень удобным суммировать все сказанное в первых трех главах книги относительно инструментального средства GRASPTC и это будет сделано в форме комментариев. Если читатель желает использовать функции, определенные в модуле GRASPTC, то ему окажутся необходимыми как прототип функции для информации о типах аргументов, так и общее описание действий, выполняемых данной функцией. Эти элементы доступны в следующем хидерном файле:

```
/* GRASPTC.H:
            Хидерный файл, предназначенный для включения в любой
            модуль, использующий функции из пакета GRASPTC.
*/

#ifndef __HUGE__
#error Use Huge Memory Model
#endif

#include <stdio.h>
#include <graphics.h>


extern FILE *fplot;

/* Если переменной 'fplot'с помощью функции fopen( ) будет приписано
   некоторое значение, то последующие обращения к функциям
   initgr( ), move( ), draw( ), grtext( ), arc_uc( ), circle80_pc( ),
   circle80_uc( ), drawarc3( ) и fillet( ) вызовут запись команд
   языка HP-GL в выходной поток 'fplot'.
*/


extern int X__max, Y__max, foregrcolor, backgrcolor, colorsum;

/* После вызова функции initgr( ) в этих переменных будут записаны
   максимальные значения пикселных координат по осям X и Y, коды
   цвета фона и основного изображения и сумма этих кодов. Верхний
   левый угол экрана является началом системы пикселных координат.
*/
```

```
extern float x_max, y_max, horfact, vertfact;

/* После вызова функции initgr( ) в переменных x_max и y_max будут
   записаны максимальные значения пользовательских координат (при
   x_max = 10.0) и вычислены коэффициенты horfact = X__max/x_max,
   vertfact = Y__max/y_max. Начало пользовательской системы коор-
   динат располагается в нижнем левом углу экрана.
*/

void initgr(void);

/* Функция определяет тип графического адаптера и осуществляет
   переключение в графический режим, имеющий наивысшее разрешение
*/

void boundaries_uc(void);

/* Если значение x_max = 10.0 неприемлемо в конкретной ситуации,
   то этой переменной можно назначить любое другое значение. Но
   это можно сделать только после обращения к функции initgr( ).
   Тогда последующий вызов функции boundaries_uc обеспечит необ-
   ходимый пересчет переменных y_max, horfact и vertfact.
*/

int IX(float x);

int IY(float y);

/* Преобразование из пользовательских в пикселные координаты      */

void move(float x, float y);

void draw(float x, float y);

/* Аргументы x и y определяют новую текущую позицию в системе
   пользовательских координат. При обращении к функции draw( )
   вычерчивается отрезок прямой линии из предыдущей позиции в
   новую текущую позицию.
*/

void endgr(void);

void to_text(void);

/* Эти две функции обеспечивают возврат в текстовый режим. Они
   отличаются тем, что функция to_tex( ) выполняет переключение
   немедленно, тогда как при вызове функции endgr( ) ожидается
   нажатие какой-либо клавиши.
*/

void invertpixel(int X, int Y);

/* Изменяет цвет свечения пиксела (X, Y), (заданного в пиксел-
   ных координатах) с цвета фона на цвет изображения и наоборот.
*/
```

```
void grtext(float xleft, float ylower, float heightpercent, char *str);

/* Реализуется вывод на экран текстовой строки при графическом режиме.
   Точка (xleft, ylower), задаваемая в системе пользовательских коор-
   динат, определяет положение нижнего левого угла первого символа.
   Высота символов примерно равна 'heightpercent' процентов от высоты
   экрана. Строка располагается горизонтально, тип шрифта — сансериф.
*/

void circle_uc(float x, float y, float r);

/* Вычерчивается окружность с центром в точке (x, y) и с радиусом r,
   аргументы задаются в единицах пользовательских координат.
*/

int XPIX(float xdim);

int YPIX(float ydim);

/* Эти функции преобразуют расстояния, заданные в единицах пользова-
   тельских координат, в расстояния в системе пикселных координат.
*/

#define PI 3.141592653589793

void line_uc(float x1, float y1, float x2, float y2);

/* Вычерчивается отрезок прямой линии между точками (x1, y1) и
   (x2, y2). Аргументы задаются в пользовательских координатах.
*/

void arc_uc(float xC, float yC, float stangle,
            float endangle, float radius, int nlinesegments);

/* Вычерчивается дуга окружности с заданными значениями координат
   центра, начального и конечного углов и радиуса в единицах поль-
   зовательских координат. Дуга аппроксимируется отрезками прямой
   линии, количество которых определяется последним аргументом.
*/

struct arc_uctype
{ float x, y, xstart, ystart, xend, yend;
};

/* Определение типа позволяет использовать тип 'struct arc_uctyre'
   для определения (и объявления) переменных; адрес такой переменной
   может быть использован в качестве аргумента функции getarc_uc( ).
*/

void getarc_uc(struct arc_uctype *arccoords);

/* После вызова функций arc_uc( ) или fillet( ) функция getarc_uc( )
   заносит значения пользовательских координат для центра, начальной
   и конечной точек дуги в структуру, указанную адресом 'arccoords'
*/
```

```
float angle(float x, float y);

/* Возвращает значение угла (-PI/2 < angle <= 3*PI/2) между положительной осью x и прямой линией, проходящей через точку начала координат O и точку (x, y).
*/

float drawarc3(float xA, float yA, float xB, float yB,
        float xC, float yC, float *pr);

/* Вычерчивается дуга, которая соединяет точки (xA, yA) и (xC, yC) и проходит также через точку (xB, yB). Радиус дуги заносится в переменную с указателем 'pr'. Аргументы задаются в единицах пользовательских координат. Функция возвращает положительное число, если точки A, B, C, именно в этом порядке, обходятся в направлении против движения часовой стрелки, и отрицательное число, если они расположены по часовой стрелке. Если точки A, B, C находятся на одной прямой линии, то возвращаемое число равно нулю и ничего не вычерчивается.
*/

float fillet(float xA, float yA, float xB, float yB,
            float xC, float yC, float r);

/* Вместо угла, образованного отрезками прямых линий AB и BC, вычерчивается дуга окружности с радиусом r, касательная к этим отрезкам. Такая дуга может использоваться для скругления угла в точке B. Начальная и конечная точки этой дуги могут быть получены путем обращения к функции getarc_uc( ). Все аргументы выражаются в единицах пользовательских координат. Возвращаемое значение такое же, как и для функции drawarc3( ).
*/

void fillcircle_uc(float x, float y, float radius);

/* Вычерчивается заполненный круг с центром и радиусом, задаваемыми в единицах пользовательских координат. Обращению к этой функции должно предшествовать обращение к функции Турбо Си setfillstyle( ) для установки типа заполнения и шаблона заполнения.*/

void circle80_pc(int XC, int YC, int R);

/* Вычерчивает окружность, аппроксимируемую 80 отрезками прямых. В противоположность функции circle( ) Турбо Си, функция circle80_pc( ) обеспечивает инвертирование пикселов после обращения к функции setwritemode(XOR_PUT). Кроме того в файл HP-GL записываются 80 отрезков прямых линий, если переменная 'fplot' не равна NULL. Аргументы выражаются в пикселных координатах.
*/

void circle80_uc(float xC, float yC, float r);
/* Аналогична функции circle80( ), но аргументы выражены в единицах пользовательских координат
.*/
```

```
void fatline0(float x1, float y1, float x2, float y2, float d);

/* Вычерчивает толстые линии с шириной d между заданными концевыми
   точками (x1, y1) и (x2, y2). Фактически вычерчивается заполненный
   прямоугольник, средние точки двух противоположных сторон которого
   совпадают с заданными концевыми точками. Аргументы выражены в
   единицах пользовательских координат. См. также описание функций
   fatline( ) и sharpjoint( )
.*/

void fatline(float x1, float y1,  float x2, float y2, float d);

/* Аналогична функция fatline0( ), но дополнительно вычерчиваются
   два заполненных полукруга с концевыми точками в качестве центров
   и с радиусом d/2.
*/

void sharpjoint(float d);

/* После выполнения трех обращений:
       fatline0(xP, yP, xQ, yQ, d);
       fatline0(xQ, yQ, xR, yR, d);
       sharpjoint(d);
   в точке пересечения Q будет вычерчен острый угол.
*/
```

Необходимо обратить специальное внимание на следующие строки, которые помещены в самом начале файла GRASPTC.H:

```
#ifndef __HUGE__
#error Use Huge Memory Model
#endif
```

Они заставляют компилятор проверить, действительно ли используется модель памяти 'Huge'. Если нет, то на экран выводится сообщение об ошибке

```
Use Huge Memory Model  ("Используйте модель Huge")
```

Поскольку строка

```
#include "grasptc.h"
```

будет вставляться во все наши графические программы, то такое сообщение будет появляться каждый раз, когда установлена несоответствующая текущая модель памяти. Напомним, что модель памяти может быть изменена в интегрированной системе Турбо Си при последовательном выборе *Options*, *Compiler*, *Model*, *Huge* и в завершение нужно не забывать сохранить в памяти выбранную опцию.

Отметим, что файл GRASPTC.H содержит также строку

```
#include <graphics.h>
```

Таким образом, объявление наших функций с помощью заголовочного файла GRASPTC.H предполагает, что одновременно будут объявлены все графические функции Турбо Си и символические константы.

Теперь можно обратиться к файлу GRASPTC.C. Он содержит некоторые добавления и расширения к тому, что обсуждалось до сих пор. Имеются две функции *_graphgetmem* и *_graphfreemem*, которые мы сами применять не будем, но они используются другими функциями в модуле GRASPTC. В руководстве по Турбо Си они получили название "пользовательский доступ" ('user hooks'). Эти функции используются подпрограммами из библиотеки графических программ Турбо Си вместо версий по умолчанию, которые вызывают функции *malloc* и *free* для выделения и освобождения областей памяти для буферов и других целей. Поскольку в нашей версии вместо них должны вызываться функции *farmalloc* и *farfree*, то тем самым мы сможем избежать любых проблем, которые могут возникать в результате ограничения размера памяти в 64K, которое налагается в функции *malloc*.

Подпрограммам и переменным в модуле GRASPTC, предназначенным только для внутреннего использования, присвоен атрибут *static*. Их имена не сообщаются редактору связей и поэтому их нельзя использовать в наших собственных программных модулях. Все остальные функции модуля GRASPTC были описаны в данной и в предшествующих главах.

```
/* GRASPTC:      Графический пакет для использования в Турбо Си.   */

#include <dos.h>
#include <stdio.h>
#include <graphics.h>
#include <conio.h>
#include <math.h>
#include <stdlib.h>
#include <alloc.h>
#include <string.h>
#include "grasptc.h"

/* Установки обработчика прерываний для графического режима:        */

#define CTRL_C 0x2E03
#define CTRL_BREAK 0x8000
```

```
static void interrupt (*oldInt9)(void);
static int *keybuffer  = (int *)0x41E;
static int *buffertail = (int *)0x41C;
static int handler_installed = 0;

static void restore_old_break(void)
{   if (handler_installed)
    {   setvect(9, oldint9); handler_installed = 0;
    }
}

static void interrupt newInt9(void)

{   unsigned int tail, code;
    oldInt9( );
            /* *buffertail = 30, 32, ..., 60 */
    tail = (*buffertail - 30) >> 1;
            /* tail = 0, 1, ..., 15 */
    code = keybuffer[tail ? tail - 1 : 15];
    if (code == CTRL_C || code == CTRL_BREAK)
    {   to_text( );
        exit(1);
    }
}

static void installBreak(void)
{   if (!handler_installed)
    {   oldInt9 = getvect(9); setvect(9,newInt9);
        handler_installed = 1;
    }
}

#define plotcoor(x) ((int)(1000 * (x) + 0.5))

int X__max, Y__max, foregrcolor, backgrcolor, colorsum;
static int Xcur, Ycur;
float x_max, y_max, horfact, vertfact;

FILE *fplot=NULL;

void boundaries_uc(void)
{   int w, h;
    getaspectratio(&w, &h);
    y_max = x_max * (float)Y__max*h/((float)X__max*w);
    horfact = X__max/x_max; vertfact = Y__max/y_max;
}

void initgr(void)
{   int gdriver=DETECT, gmode;
        /* Последующие обращения к функциям registerfarbgidriver( ) и*/
        /* registerfarbgifont( ) предполагают, что уже были созданы    */
        /* объектные файлы CGAF.OBJ, EGAVGAF.OBJ, HERCF.OBJ,   */
```

```
        /* TRIPF.OBJ, LITTF.OBJ и SANSF.OBJ с помощью команд        */
        /* BGIOBJ /F CGA, EGIOBJ /F EGAVGA и так далее.             */
        /* Нельзя опустить какие-либо из них или внести другие      */
        /* по своему усмотрению (см. параграф 1.7).                 */

    registerfarbgidriver(CGA_driver_far);
    registerfarbgidriver(EGAVGA_driver_far);
    registerfarbgidriver(Herc_driver_far);
    registerfarbgifont(triplex_font_far);
    registerfarbgifont(small_font_far);
    registerfarbgifont(sansserif_font_far);
    initgraph(&gdriver, &gmode, "\\tc");
            /* Третий аргумент "\\tc" обычно не должен указываться ! */

    if (graphresult( ))
    {   printf ( "\nГрафический драйвер не доступен.\n"); exit(1);
    }

    installBreak( );
    foregrcolor = getcolor( ); backgrcolor = getbkcolor( );
    colorsum = foregrcolor + backgrcolor;
    X__max = getmaxx( ); Y__max = getmaxy( );
    x_max = 10.0;
    boundaries_uc( );
    if (fplot != NULL)
        fprintf(fplot, "IN;SC0,10000,0,%d;\n", plotcoor(y_max));
}

void invertpixel(int X, int Y)
{   putpixel(X, Y, colorsum - getpixel(X, Y));
}

int IX(float x)
{   return (int) (x * horfact + 0.5);
}

int IY(float y)
{   return Y__max - (int)(y * vertfact + 0.5);
}

void move(float x, float y)

{   Xcur = IX(x); Ycur = IY(y);
    moveto(Xcur, Ycur);
    if (fplot != NULL) fprintf(fplot, "PU;PA%d,%d;",
        plotcoor(x), plotcoor(y));
}

void draw(float x, float y)

{   int X0=Xcur, Y0=Ycur;
    Xcur = IX(x); Ycur = IY(y);
    line(X0, Y0, Xcur, Ycur);
```

```
    moveto(Xcur, Ycur);
    if (fplot != NULL) fprintf(fplot, "PD;PA%d,%d;\n",
        plotcoor(x), plotcoor(y));
}

void to_text(void)
{   closegraph( );
    restore_old_break( );
}

void endgr(void)
{   getch( ); to_text( );
}

void grtext(float xleft, float ylower, float heightpercent, char *str)

{   char *s;
    int len, charsize;
    float widthpercent=heightpercent*4.0/7.0;
    len = strlen(str);
    s = farmalloc(len+2);
    if (s == NULL){to_text( ); printf("farmalloc");exit(1);}
    strcpy(s, str);
    s[len] = '\003'; s[len+1] = '\0';
    if (fplot != NULL)
    {   fprintf(fplot, "SR%3.1f,%3.1f;",
        widthpercent, heightpercent);
        fprintf(fplot, "PU;PA%d,%d;LB%s;\n",
        plotcoor(xleft), plotcoor(ylower), s);
    }
    charsize = (int)(heightpercent/2 + 0.5);
    if (charsize > 10) charsize = 10;
    settextstyle(SANS_SERIF_FONT, HORIZ_DIR, charsize);
    outtextxy(IX(xleft), IY(ylower)-textheight(str)-charsize, str);
    farfree(s);
}

void far * far _graphgetmem(unsigned size)

{   char far *p;
    p = farmalloc((long)size);
    if (p == NULL)
    {   printf("Не хватило памяти для функции _graphgetmem");
        exit(1);
    }
    return p;
}

void far _graphfreemem(void far *ptr, unsigned size)

{   farfree(ptr);
```

```
void line_uc(float x1, float y1, float x2, float y2)
{   line(IX(x1), IY(y1), IX(x2), IY(y2));
}

void circle_uc(float x, float y, float r)
{   circle(IX(x), IY(y), IX(r));
}

int XPIX(float xdim)
{   return (int)(xdim * horfact + 0.5);
}

int YPIX(float ydim)
{   return (int)(ydim * vertfact + 0.5);
}

static float xarc_C, yarc_C, xarc_start, yarc_start, xarc_end, yarc_end;

void arc_uc(float xC, float yC, float stangle,
            float endangle, float radius, int nlinesegments)

{   float theta, phi;
    int i;
    while (endangle < stangle) endangle += 2*PI;
    theta = (endangle - stangle)/nlinesegments;
    xarc_start = xC + radius*cos(stangle);
    yarc_start = yC + radius*sin(stangle);
    xarc_end = xC + radius*cos(endangle);
    yarc_end = yC + radius*sin(endangle);
    move(xarc_start, yarc_start);
    for (i=1; i<nlinesegments; i++)
    {   phi = stangle + i * theta;
        draw(xC+radius*cos(phi), yC+radius*sin(phi));
    }
    draw(xarc_end, yarc_end);
    xarc_C = xC; yarc_C = yC;
}

void getarc_uc(struct arc_uctype *arccoords)
{   arccoords->x = xarc_C;
    arccoords->y = yarc_C;
    arccoords->xstart = xarc_start;
    arccoords->ystart = yarc_start;
    arccoords->xend = xarc_end;
    arccoords->yend = yarc_end;
}

float angle(float x, float y)
{   return (x > 0 ? atan(y/x) :
            x < 0 ? PI + atan(y/x) :
            y >= 0 ? PI/2 : 3*PI/2);
```

```
float drawarc3 (float xA, float yA,
                float xB, float yB,
                float xC, float yC,
                float *pr)
{   float u1, u2, n1, n2, xD, yD, v1, v2, m1, m2, xE, yE, lambda,
        xO, yO, r1, r2, r, stangle, endangle, phiA, phiC, determinant;
    int nsteps;

    u1 = xB - xA; u2 = yB - yA;          /* Вектор u направлен от A к B        */
    n1 = u2; n2 = -u1;                   /* Вектор n перпендикулярен к u       */
    xD = (xA+xB)/2; yD = (yA+yB)/2;      /* D = средняя точка AB               */
    v1 = xC - xB; v2 = yC - yB;          /* Вектор v направлен от B к C        */
    m1 = v2; m2 = -v1;                   /* Вектор m перпендикулярен к v       */
    xE = (xB+xC)/2; yE = (yB+yC)/2;      /* E = средняя точка BC               */
    determinant = m2*n1 - m1*n2;
    if (determinant == 0.0) return 0.0;

    lambda = (m2*(xE-xD) - m1*(yE-yD))/determinant;
            /* Вектор lambda.n направлен из точки D в точку O                  */
            /* O = центр окружности, проходящей через точки A, B и C           */
    xO = xD + lambda * n1;
    yO = yD + lambda * n2; /
    r1 = xA-xO; r2 = yA-yO;      /* Вектор (r1, r2) направлен из O в A         */
    r = sqrt(r1*r1 + r2*r2);
    phiA = angle(r1, r2);
    phiC = angle(xC-xO, yC-yO);

    if (determinant > 0)
    {   stangle = phiA; endangle = phiC;
    } else
    {   stangle = phiC; endangle = phiA;
    }

    if (endangle < stangle) endangle += 2*PI;
    nsteps = (int)(r * (endangle - stangle) * 10) + 1;
                /* Число шагов зависит как от радиуса, так и от угла          */
    arc_uc(xO, yO, stangle, endangle, r, nsteps);
    *pr = r;
    return determinant;
}

float fillet ( float  xA,  float yA,  float xB,  float yB,
               float  xC,  float yC,  float r )

{   float n1, n2, m1, m2, BA, BC, s1, s2, length_of_s,
    b1, b2, cosa, sina, BO1, BO2, xO, yO, proj,
    xD, yD, xE, yE, xF, yF, q;
    n1 = xA - xB; n2 = yA - yB;
    BA = sqrt(n1*n1 + n2*n2);
    n1 /= BA; n2 /= BA;
    m1 = xC - xB; m2 = yC - yB;
```

```
    BC = sqrt(m1*m1 + m2*m2);
    m1 /= BC; m2 /= BC;

    s1 = n1 + m1; s2 = n2 + m2;
    length_of_s = sqrt(s1*s1 + s2*s2);
    b1 = s1/length_of_s; b2 = s2/length_of_s;
    cosa = length_of_s/2;
    sina = sqrt(1 - cosa*cosa); q = r/sina;
    BO1 = q * b1; BO2 = q * b2;
    xO = xB + BO1; yO = yB + BO2;
    proj = BO1 * n1 + BO2 * n2;

    xD = xB + proj * n1; yD = yB + proj * n2;
    xE = xB + proj * m1; yE = yB + proj * m2;
    xF = xO - r * b1; yF = yO - r * b2;
    return drawarc3(xD, yD, xF, yF, xE, yE, &r);
}

void fillcircle_uc(float x, float y, float radius)
{   fillellipse(IX(x), IY(y), XPIX(radius), YPIX(radius));
}

#define COSDELTA 0.996917333733
#define SINDELTA 0.078459095728

void circle80_pc(int XC, int YC, int R)
/*          Окружность аппроксимируется правильным 80-угольником  */
/*          в системе пикселных координат                          */

{   static int first=1, stdpoints[162];
    int j=0, H, V, points[162];
    double costh, sinth, c0, s0;
    float x, y;

    if (first)
    {   first = 0;
        costh = 1.0; sinth = 0.0;
        for (j=0; j<=40; j+=2)
        {   H = (int)(1e4 * costh + 0.5);
            V = (int)(1e4 * sinth * vertfact/horfact + 0.5);
            stdpoints[j] = H; stdpoints[j+1] = V;
            if (j)
            {   stdpoints[160-j] = H;
                stdpoints[161-j] = -V;
            }
            if (j != 40)
            {   stdpoints[80-j] = -H; stdpoints[81-j] = V;
                if (j)
                {   stdpoints[80+j] = -H;
                    stdpoints[81+j] = -V;
                }
            }
```

```
            c0 = costh; s0 = sinth;
            costh = c0 * COSDELTA - s0 * SINDELTA;
            sinth = s0 * COSDELTA + c0 * SINDELTA;
        }
        stdpoints[160] = stdpoints[0];
        stdpoints[161] = stdpoints[1];
    }
    for (j=0; j<=160; j+=2)
    {   points[j] = XC + (int)(stdpoints[j]*(long)R/10000L);
        points[j+1] = YC + (int)(stdpoints[j+1]*(long)R/10000L);
        if (j) invertpixel(points[j], points[j+1]);
                /* Необходимо в случае XOR_PUT     */
    }

    if (fplot == NULL) drawpoly(81, points); else
    {   for (j=0; j<=80; j++)
        {   x = points[2*j]/horfact;
            y = (Y__max - points[2*j+1])/vertfact;
            if (j==0) move(x, y); else draw(x, y);
        }
    }
}

void circle80_uc(float xC, float yC, float r)
/*          Окружность аппроксимируется правильным 80-угольником  */
/*          в системе пользовательских координат                  */

{   circle80_pc(IX(xC), IY(yC), XPIX(r));
}

static float xA, yA, xB, yB, xC, yC;

void fatline0(float x1, float y1, float x2, float y2, float d)

{   float dx, dy, dxhw, dyhw, factor, r=d/2;
    int points[10];
    dx = x2 - x1; dy = y2 - y1;
    factor = r/sqrt(dx * dx + dy * dy);
    dxhw = dy * factor;     /* Приращение x для половины ширины     */
    dyhw = dx * factor;     /* Приращение y для половины ширины     */

    points[0] = IX(x1+dxhw); points[1] = IY(y1-dyhw);
    points[2] = IX(x2+dxhw); points[3] = IY(y2-dyhw);
    points[4] = IX(x2-dxhw); points[5] = IY(y2+dyhw);
    points[6] = IX(x1-dxhw); points[7] = IY(y1+dyhw);
    points[8] = points[0]; points[9] = points[1];
    fillpoly(5, points);

    xA = xB; yA = yB;
    xB = x1; yB = y1;
    xC = x2; yC = y2;
}
```

```
void fatline(float x1, float y1, float x2, float y2, float d)

{   float r=d/2;
    fatline0(x1, y1, x2, y2, d);
    fillcircle_uc(x1, y1, r);
    fillcircle_uc(x2, y2, r);
}

void sharpjoint(float d)

{   float r=d/2, BA, BC, dxBA, dyBA, dxBC, dyBC, s,
        sina, factor, xD, yD, xE, yE, xF, yF, m1,
        m2, n1, n2, rm1, rm2, rn1, rn2;
    int points[10];

    dxBA = xA - xB; dyBA = yA - yB;
    dxBC = xC - xB; dyBC = yC - yB;
    BA = sqrt(dxBA * dxBA + dyBA * dyBA);
    BC = sqrt(dxBC * dxBC + dyBC * dyBC);
    n1 = -dxBA/BA; n2 = -dyBA/BA;
    m1 = -dxBC/BC; m2 = -dyBC/BC;
    s = m1 * n2 - n1 * m2;
    sina = fabs(s);
    factor = r/sina;

    xD = xB + factor * (m1 + n1);
    yD = yB + factor * (m2 + n2);
    rm1 = r * m1; rm2 = r * m2;
    rn1 = r * n1; rn2 = r * n2;

    if (s > 0)
    {   xE = xB - rm2; yE = yB + rm1;
        xF = xB + rn2; yF = yB - rn1;
    } else
    {   xE = xB + rm2; yE = yB - rm1;
        xF = xB - rn2; yF = yB + rn1;
    }

    points[0] = IX(xB); points[1] = IY(yB);
    points[2] = IX(xF); points[3] = IY(yF);
    points[4] = IX(xD); points[5] = IY(yD);
    points[6] = IX(xE); points[7] = IY(yE);
    points[8] = points[0]; points[9] = points[1];
    fillpoly(5, points);
}
```

Глава 4

РЕКУРСИЯ И ФРАКТАЛЫ

4.1. РЕКУРСИЯ

Мы говорим, что функция рекурсивна (или что она основана на рекурсии), если в функции содержится одно или несколько обращений к самой себе или к другим функциям, в которых есть обращения к такой функции. При входе в обычную функцию выход из нее всегда происходит раньше, чем повторный вход, но для рекурсивной функции это необязательно.

Начнем с простого треугольника как в программах TRIA и TRIA1 из параграфов 1.2 и 1.5. В программе TRIANGLS сначала вычертим такой же треугольник, но затем снова вызовем функцию *tria* с координатами средних точек его сторон, так что появятся четыре малых треугольника. Затем функция *tria* трижды вызовет сама себя с координатами вершин малых треугольников в качестве аргументов. В обращение включен целочисленный аргумент *n*, определяющий *глубину рекурсии*. Начнем с некоторого целого числа, допустим, 7, заданного пользователем, этот аргумент устанавливается равным $n - 1$ для каждого из трех рекурсивных обращений. То есть при достижении "самого глубокого уровня рекурсии", значение *n* становится равным нулю, что приводит к немедленному возврату в вызывающую функцию, то есть в саму функцию *tria*. Заметим, что в программе TRIANGLS можно использовать значения *x_max* и *y_max* даже без их явного объявления, поскольку они объявлены в хидерном файле GRASPTC.H. На рис. 4.1 показан результат работы программы.

```
/* TRIANGLS Подобные прямоугольные треугольники разных размеров.

#include <stdio.h>
#include "grasptc.h"
```

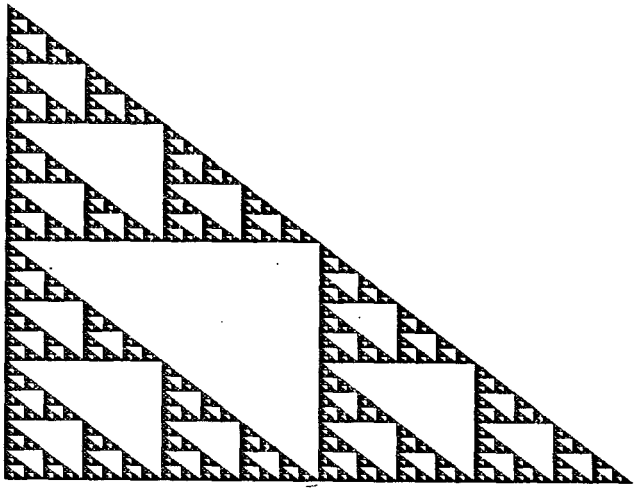


Рис. 4.1. Результат работы программ TRIANGLS и INTTRIA

```
void tria(float xA, float yA, float xB, float yB,
          float xC, float yC, int n)
{   float xP, yP, xQ, yQ, xR, yR;
    if (n > 0)
    {   xP = (xB + xC)/2; yP = (yB + yC)/2;
        xQ = (xC + xA)/2; yQ = (yC + yA)/2;
        xR = (xA + xB)/2; yR = (yA + yB)/2;
        move(xP, yP); draw(xQ, yQ);
        draw(xR, yR); draw(xP, yP);
        tria(xA, yA,  xR, yR,  xQ, yQ,  n-1);
        tria(xB, yB,  xP, yP,  xR, yR,  n-1);
        tria(xC, yC,  xQ, yQ,  xP, yP,  n-1);
    }
}

main( )
{   int n;
    float xA, yA, xB, yB, xC, yC;
    printf ( "\nГлубина рекурсии (например, 7) :  ");
    scanf("%d", &n);
    initgr( );
    xA = 0.0, yA = 0.0;
    xB = x_max; yB = 0.0;
    xC = 0.0; yC = y_max;
    move(xA, yA); draw(xB, yB); draw(xC, yC); draw(xA, yA);
    tria(xA, yA,  xB, yB,  xC, yC,  n);
    endgr( );
}
```

Рекурсивная программа отличается от нерекурсивной тем, что в ней едва ли возможно проследить за управлением обычным образом. Это часто становится неразрешимой проблемой и было бы очень неплохо, если бы мы могли написать такую программу, совершенно не беспокоясь о пикселах, графических адаптерах и о направлении оси y вниз. Но теперь, когда готова программа TRIANGLS, мы видим, что заключенная в ней арифметика очень проста, но можно заметить, что выполнение программы происходит сравнительно медленно, если увеличивается глубина рекурсии *n*.

Поэтому предлагается рассмотреть другую версию программы — INTTRIA, в которой используются только целые числа. Результат ее работы идентичен полученному при выполнении программы TRIANGLS, но новая программа работает значительно быстрее. Отметим, что в программе INTTRIA объединены быстро работающие функции *moveto* и *lineto* с удобными функциями *initgr* и *endgr*, которые были разработаны нами. Можно сказать, что таким образом мы выбрали наилучшее из двух решений.

Вследствие использования функции *initgr* немедленно становятся доступными максимальные значения пикселных координат *X_max* и *Y_max*, поэтому в программе нет необходимости самим обращаться к функциям *getmaxx* и *getmaxy*.

```
/* INTTRIA:                                                    */
/*          Фракталы на основе прямоугольных треугольников.    */
/*          Быстро работающая версия.                          */

#Include <stdio.h>
#Include "grasptc.h"

void tria(int xA, Int yA, int xB, int yB, int xC, int yC, int n)

{   int xP, yP, xQ, yQ, xR, yR;
    if (n > 0)
    {   xP = (xB + xC)/2; yP = (yB + yC)/2;
        xQ = (xC + xA)/2; yQ = (yC + yA)/2;
        xR = (xA + xB)/2; yR = (yA + yB)/2;
        moveto(xP, yP); lineto(xQ, yQ);
        lineto(xR, yR); lineto(xP, yP);
        tria(xA, yA,  xR, yR,  xQ, yQ,  n-1);
        tria(xB, yB,  xP, yP,  xR, yR,  n-1);
        tria(xC, yC,  xQ, yQ,  xP, yP,  n-1);
    }
}
```

```
main( )
{   int n;
    int xA, yA, xB, yB, xC, yC;
    printf ( "\nГлубина рекурсии (например, 7) : "); scanf("%d", &n);
    initgr( );
    xA = 0, yA = Y__max; xB = X__max; yB = Y__max; xC = 0; yC = 0;
    moveto(xA, yA); lineto(xB, yB); lineto(xC, yC); lineto(xA, yA);
    tria(xA, yA, xB, yB, xC, yC, n);
    endgr( );
}
```

4.2. ГРАФИКА И СЛУЧАЙНЫЕ ЧИСЛА

Иногда нам не нужна полная симметрия, возникающая при прямом применении рекурсии. В таких случаях можно использовать (псевдо) случайные числа для устранения в некоторой степени такой симметрии. Этим способом была получена картинка на рис. 4.2. На ней показано многократное изображение большой буквы Т. Точку в нижней части буквы Т назовем ее начальной точкой. Начнем с большой буквы Т в ее нормальном положении, а концевые точки на верхней горизонтальной полке будут в свою очередь начальными точками новых букв Т, несколько меньших, чем первоначальная. От пользователя программы TTREE запрашивается ввод двух коэффициентов уменьшения (*fx* и *fy*), глубины рекурсии и порогового значения в процентах. После вычерчивания каждой буквы Т генерируется случайное число,

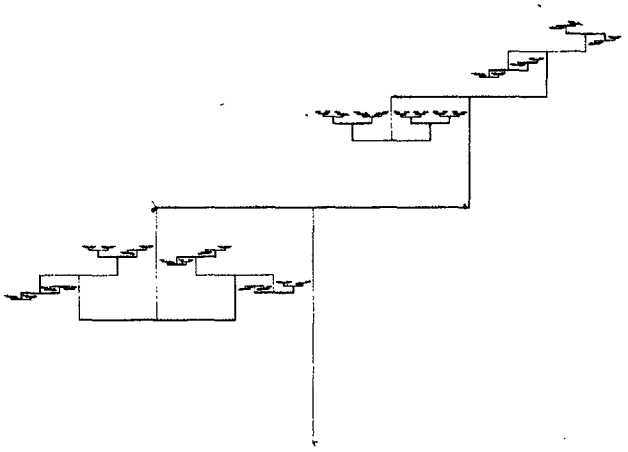


Рис. 4.2. Результат работы программы TTREE

меньше 100. Если это число меньше, чем заданное пороговое значение, то новая буква Т вычерчивается в нормальном положении, в противном случае — в перевернутом. В программе используется хорошо известная формула

$$1 + r + r^2 + \ldots + r^{n-1} = \frac{1 - r^n}{1 - r}$$

для вычисления размера первой буквы Т, такого, чтобы окончательный результат разместился в пределах границ экрана.

```
/* TTREE:                                                          */
/*                Формирование дерева из буквы Т.                  */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "grasptc.h"

float fx, fy, threshold;

void T(float xA, float yA, float a, float b, int n)

{   float xB, yB, xC, yC, xD, yD, a1, b1;

        /* А обозначает нижнюю точку буквы Т. Концевые точки       */
        /* горизонтального (верхнего) отрезка обозначены           */
        /* буквами С (слева) и D (справа), средняя точка - В.      */

    if (n > 0)
    {   xB = xA; yB = yA + a          /* AB = a          */
        xC = xB - b; yC = yB;         /* CB = BD = b     */
        xD = xB + b; yD = yB;
        a1 = fy * a;
        if (a1 < 0) a1 = -a1;
        b1 = fx * b;
        move(xA, yA); draw(xB, yB); draw(xC, yC);
        T(xC, yC, (rand( )% 100 >= threshold ? -a1 : a1), b1, n-1);
        move(xB, yB); draw(xD, yD);
        T(xD, yD, (rand( )% 100 >= threshold ? -a1 : a1), b1, n-1);
    }
}

main( )
{   float a, b, powerfx, powerfy;
    int n, l;
    long t;
    time(&t); srand((int)t);
    printf ( "\nВведите значения коэффициентов:\n"
             "\n          fx   fy  глубина рекурсии  % уменьшения вверх"
             "\n\n(например,  0.5  0.4       7          70)\n\n" );
```

```
    printf("                    ");
    scanf("%f %f %d %f", &fx, &fy, &n, &threshold);
    initgr();
    powerfx = fx; powerfy = fy;

    for (i=2; i<=n; i++)
    {   powerfx *= fx; powerfy *= fy;
    }

    b = 0.5 * x_max * (1 - fx)/(1 - powerfx);
    a = y_max * (1 - fy)/(1 - powerfy);
    T(x_max/2, y_max/2 - a, a, b, n);
    endgr();
}
```

4.3. РЕКУРСИЯ И ПРЕОБРАЗОВАНИЯ

Теперь обсудим программу, в которой выполняются следующие два линейных преобразования:

L:

$$x' = ax + by$$
$$y' = bx + ay$$

R:

$$x' = c(x - x_0) - dy + x_0$$
$$y' = d(x - x_0) + cy$$

Чтобы понять, что делает преобразование L, исследуем его поведение относительно единичных точек (1, 0) и (0, 1). Приняв $x = 1$ и $y = 0$, найдем $x' = a$ и $y' = b$. То есть изображение точки (1, 0) будет (a, b). Аналогично для (0, 1) изображение будет в точке $(b, -a)$. Если принять $a = \cos\varphi$ и $b = \sin\varphi$, то увидим, что преобразование L описывает поворот вокруг точки O на угол φ, за которым следует отражение относительно прямой линии, проведенной под углом φ относительно положительной оси x. Совместно эти две операции эквивалентны отражению относительно прямой линии, образующей с положительной осью x угол $\varphi/2$. В этом случае длина вектора (a, b) равна 1. В нашем примере эта длина будет меньше 1. То есть будет комбинация отражения со сжатием, причем коэффициент сжатия равен

$$\sqrt{(a^2 + b^2)} < 1$$

Преобразование R обозначает поворот вокруг точки $(x_0, 0)$, совмещенный со сжатием. Здесь коэффициент сжатия равен

$$\sqrt{(c^2 + d^2)} < 1$$

Наиболее интересное заключается даже не в самих этих преобразованиях, а в способе их использования. Обозначим нашу исходную точку (1, 0) в виде вектора **u**. Теперь применим преобразование L к вектору **u**, что даст нам новую точку, которую обозначим L**u**. К вектору **u** также применим преобразование R, что даст новую точку R**u**. Затем снова применим (независимо) к каждой из этих новых точек преобразование L и R, что даст четыре новые точки LL**u**, RL**u**, LR**u**, RR**u**. К каждой из этих новых четырех точек опять применим оба преобразования L и R, что даст новые восемь точек, и так далее. Ясно, что весь этот процесс аналогичен вычерчиванию двоичного дерева, что и было сделано в параграфе 4.2.

Можно было бы сказать, что выбор начальной точки **u** = (1, 0) не приводит к полезному результату, имея ввиду нашу систему координат и размеры экрана. Напомним, что в параграфе 4.2 мы могли вычислить размеры первой буквы T такими, чтобы окончательный результат полностью вписывался в пределы границ экрана. Если такой аналитический подход оказывается слишком сложным, а иногда и просто невозможным, то всегда годится другой способ, который и применим здесь. Будем просто все точки вычислять дважды. При первом проходе полученные данные будут использованы для нахождения наименьших и наибольших значений всех точек по координатным осям x и y, основанных на начальной точке **u** = (1, 0). Затем сравним полученные диапазоны с доступными пределами в нашей системе координат. Путем деления последних диапазонов на первые получим коэффициенты масштабирования fx и fy. Возьмем наименьший из них и дополнительно уменьшим его немного, чтобы обеспечить свободные поля по краям экрана. Таким образом будет определен коэффициент масштабирования, который и будем использовать. Кроме вычисленных значений диапазона нам интересно также знать центр вычисленных координат, чтобы его можно было сопоставить с центром экрана.

Сказанного о проблеме масштабирования достаточно для наших целей. Можно заметить, что вместо двукратного повторения всех вычислений, можно было бы запомнить все точки либо в оперативной памяти, либо на диске и использовать их на втором этапе. Такой подход привлекателен, если процесс вычислений занимает очень много времени. В программе TRANSFOR все

Рис. 4.3. Образец результата, полученного при работе программы TRANSFOR

точки вычисляются дважды. Для процессора 8088 вычисления занимают не очень длительное время до тех пор, пока глубина рекурсии не превышает 10, но потребуется несколько часов, если эта глубина равна 18, что и произошло при подготовке рис. 4.3. Всегда очень раздражает, если компьютер выполняет какую-либо работу, ничего не отображая на экране. Поэтому программа TRANSFOR отображает значение счетчика в диапазоне от 1 до 32, когда, находясь в текстовом режиме, она выполняет расчет граничных значений. Этот счетчик увеличивается на 1 каждый раз при достижении пятого уровня рекурсии. Затем происходит переключение в графический режим и теперь в верхней части экрана будут отображаться 32 точки по горизонтальной прямой на расстоянии 8 пикселов друг от друга. После отображения последней точки все эти точки будут удалены, что указывает на получение на экране окончательного результата. Такая индикация очень полезна особенно при больших значениях глубины рекурсии, поскольку иначе непонятно, закончилось ли формирование изображения на экране, а если нет, то сколько еще осталось ждать.

Эту программу можно изменять для получения различных видов привлекательных картинок, сформированных из точек,

хотя большинство наборов входных данных дает не очень стоящие результаты. После многих экспериментов автор использовал такие входные данные:

Глубина рекурсии = 18, $a = 0.7$, $b = 0.3$, $e = 0.5$, $d = 0.3$, $x0 = 1.0$

что привело к получению изображения, показанного на рис. 4.3. Фигуры с рекурсивным подобием, аналогичные полученным здесь, называются *фракталами*. При повышении глубины рекурсии увеличенные детали фрактала подобны полному изображению.

```
/* TRANSFOR:                                                    */
/*                    Рекурсия и преобразования                 */

#include <stdio.h>
#include "grasptc.h"

float xreal(float x);
float yreal(float y);

void transf(float x, float y, int n, int prescan);

float fx, fy, f, xC, yC, xmin0=100, xmax0=-100, ymin0=100,
    ymax0=-100, xC0, yC0, a, b, c, d, x0, x, y;
int count=0, recdepth, level_5;

main()

{   int i;
    printf ( "\nГлубина рекурсии (например, 12) : ");
    scanf("%d", &recdepth); level_5 = recdepth - 5;

    printf ( "\nВведите числа a, b, c, d, x0  "
             "(например, .7  .3  .5  .3  1) :\n");
    printf("                                                    ");
    scanf("%f %f %f %f %f", &a, &b, &c, &d, &x0);
    printf("\nПожалуйста ждите, конечное значение счетчика будет 32 :\n");
    transf(1.0, 0.0, recdepth, 1);

    initgr();
    fx = x_max/(xmax0 - xmin0);
    fy = y_max/(ymax0 - ymin0);
    f = (fx < fy ? fx : fy) * 0.8;
    xC0 = (xmin0 + xmax0)/2; yC0 = (ymin0 + ymax0)/2;
    xC = x_max/2; yC = y_max/2;
    count = 0;
    transf(1.0, 0.0, recdepth, 0);

    for (i=1; i<=32; i++) putpixel(8*i, 0, backgrcolor);
    endgr();
}
```

```
void transf(float x, float y, int n, int prescan)

{   float x1, y1;
    if (n > 0)
    {   if (prescan)
        {   if (n == level_5) printf("%d ", ++count);
            if (x < xmin0) xmin0 = x;
            if (x > xmax0) xmax0 = x;
            if (y < ymin0) ymin0 = y;
            if (y > ymax0) ymax0 = y;
        } else
        {   if (n == level_5) putpixel(count++8, 0, foregrcolor);
            putpixel(IX(xreal(x)), IY(yreal(y)), foregrcolor);
        }

        x1 = a * x + b * y;
        y1 = b * x - a * y;
        /* Отражение и сжатие относительно фиксированной точки (0, 0) */
        transf(x1, y1, n-1, prescan);

        x1 = c * (x-x0) - d * y + x0;
        y1 = d * (x-x0) + c * y;
          /* Поворот и сжатие относительно фиксированной точки (x0, 0) */
        transf(x1, y1, n-1, prescan);
    }
}

float xreal(float x)
{   return xC + f * (x - xC0);
}

float yreal(float y)
{   return yC + f * (y - yC0);
}
```

4.4. КРИВЫЕ ГИЛЬБЕРТА

Рекурсия может быть использована для получения линейного рисунка, известного под именем *кривой Гильберта*. Кривая Гильберта основана на изображении буквы П, вычерченной в виде трех сторон квадрата, как показано в левой части рис. 4.4. Существуют кривые Гильберта порядков 1, 2, ..., обозначаемые как H_1, H_2, ... В центре рис. 4.4 изображена кривая H_2, в которой некоторые отрезки прямых линий, так называемые *связки*, вычерчены в виде толстых линий. В действительности эти отрезки прямых линий должны иметь одинаковую толщину с другими отрезками, здесь же они показаны толстыми единственно с целью демонстрации способа получения H_2 из H_1 (показанного

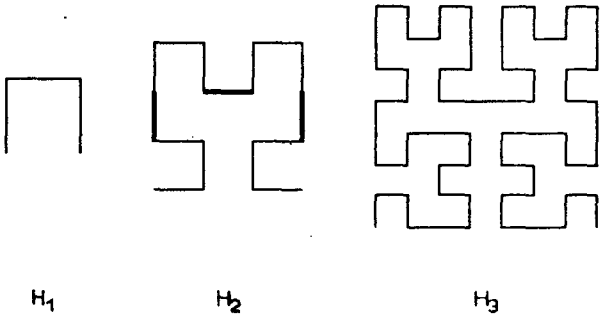


Рис. 4.4. Кривые Гильберта 1, 2 и 3 порядков

слева). Мы видим, что H_2 можно рассматривать как большую букву П, четыре части которой заменены меньшими по размеру буквами П. Эти меньшие буквы П соединены тремя связками. Каждая сторона меньшей буквы П имеет ту же длину, что и связка, они в три раза меньше стороны квадрата, в который вписывается H_2. Применим ту же процедуру к каждой из четырех букв П, составляющих H_2, то есть каждую букву П в H_2 заменим меньшей H_2, одновременно уменьшим длину связок так, чтобы их длины стали равными длине элементарного отрезка прямой линии, которые содержатся в трех малых фигурах H_2. Таким образом мы получим фигуру H_3, показанную на рис. 4.4 справа. Теперь все элементарные отрезки в семь раз меньше, чем длина сторон квадрата, в который вписывается фигура H_3. Отсюда получаем, что коэффициенты уменьшения для этих элементарных отрезков в фигурах H_1, H_2, H_3, ... образуют ряд чисел 1, 3, 7, ..., то есть в общем случае коэффициент уменьшения для фигуры H_n может быть вычислен по формуле $2^n - 1$.

Заметим, что связки в фигуре H_2 вычерчиваются в тех же направлениях, как и три отрезка, образующие букву П в фигуре H_1. При желании эти последние отрезки прямых линий можно рассматривать как связки, соединяющие четыре точки, которые в свою очередь можно принять за кривую Гильберта нулевого порядка.

В нашей программе для кривых Гильберта используем рекурсивную функцию *Hilbert* со следующими аргументами:

– Координаты точек A, B, C (см. рис. 4.5)

- Горизонтальные и вертикальные компоненты двух направленных связок; причем одна лежит на отрезке AB, а другая — на AC. Они задаются в виде векторов, то есть как пара чисел (dx, dy), где переменные dx и dy могут принимать положительные, нулевые или отрицательные значения, в зависимости от относительного положения точек A, B и C. Эти два вектора в программе обозначаются как **dAB** и **dAC**.
- Глубина рекурсии n. Для $n = 0$ функция ничего не будет делать.

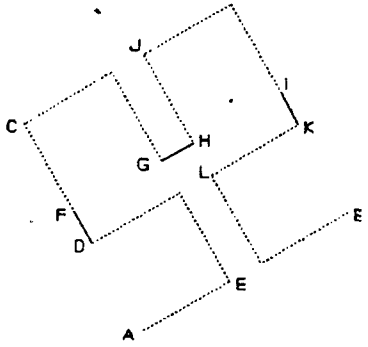


Рис. 4.5. Взаимосвязь точек в кривой Гильберта

Будем считать, что рис. 4.5 является вариацией изображения буквы П (повернутой на угол 30° против часовой стрелки), позиция которой целиком определяется тремя заданными точками A, B и C. Будем считать, что точка A является начальной точкой, а точка B — конечной. Основной причиной задания точки C является необходимость указания, с какой стороны от направленного отрезка AB должна лежать вычерчиваемая кривая. Оба заданных вектора связок **dAB** и **dAC** отмечены на рис. 4.5 в виде связок в трех местах, а именно как отрезки DF, GH и IK. Три заданные точки A, B, C и два заданных вектора **dAB** и **dAC** позволяют определить позиции точек D, E, F, G, H, I, J, K на рис. 4.5. (Мы не будем требовать, чтобы угол CAB был прямым углом или чтобы длина отрезка AB совпадала с длиной отрезка AC, поэтому вместо квадрата каждая буква П может иметь форму любого параллелограмма). В общем случае точечные линии на рис. 4.5 фактически не вычерчиваются. Вместо этого мы будем выполнять рекурсивное обращение к нашей функции *Hilbert* для каждой из

четырех точечных букв П на рис. 4.5. Для вычерчивания трех отрезков связок DF, GH и IK будем обращаться к нашей функции *draw*.

```
/* HILBERT:                                                          */
/*                    Кривые Гильберта любого порядка                */

#include "grasptc.h"

typedef struct {float x, y;} vec;
int recdepth, steps;

void Hilbert(vec A, vec B, vec C, vec dAB, vec dAC, int n)

{   vec D, E, F, G, H, I, J, K, L;
    if (n > 0)
    {   D.x = (A.x + C.x - dAC.x)/2;
        D.y = (A.y + C.y - dAC.y)/2;
        E.x = (A.x + B.x - dAB.x)/2;
        E.y = (A.y + B.y - dAB.y)/2;
        F.x = D.x + dAC.x; F.y = D.y + dAC.y;
        G.x = F.x + E.x - A.x; G.y = F.y + E.y - A.y;
        H.x = G.x + dAB.x; H.y = G.y + dAB.y;
        I.x = F.x + B.x - A.x; I.y = F.y + B.y - A.y;
        J.x = C.x + H.x - F.x; J.y = C.y + H.y - F.y;
        K.x = I.x - dAC.x; K.y = I.y - dAC.y;
        L.x = H.x - dAC.x; L.y = H.y - dAC.y;

        Hilbert(A, D, E, dAC, dAB, n-1);
        draw(F.x, F.y);        /* Связь DF */        /* Link DF */
        Hilbert(F, G, C, dAB, dAC, n-1);
        draw(H.x, H.y);        /* Связь GH */ /* Link GH */
        Hilbert(H, I, J, dAB, dAC, n-1);

        draw(K.x, K.y);        /* Связь IK */ /* Link IK */
        dAB.x = -dAB.x; dAB.y = -dAB.y;
        dAC.x = -dAC.x; dAC.y = -dAC.y;
            /* Эти изменения векторов dAB и dAC являются только      */
            /* локальными                                            */
        Hilbert(K, B, L, dAC, dAB, n-1);
    }


void square(float xA, float yA, float xB, float yB, float xC, float yC)

    vec A, B, C, dAB, dAC;
    A.x = xA; A.y = yA;
    B.x = xB; B.y = yB;
    C.x = xC; C.y = yC;

    dAB.x = (xB - xA)/steps;
    dAB.y = (yB - yA)/steps;
```

```
    dAC.x = (xC - xA)/steps;
    dAC.y = (yC - yA)/steps;

    move(xA, yA);
    Hilbert(A, B, C, dAB, dAC, recdepth);
}

main()

{   float xCenter, yCenter, h, xP, yP, xQ, yQ, xR, yR;

    printf ( "\nВведите глубину рекурсии : ");
    scanf("%d", &recdepth);
    steps = (1 << recdepth) - 1;
                                /* steps = power(2, recdepth) - 1        */

    Initgr();
    xCenter = x_max/2;
    yCenter = y_max/2;
    h = y_max/40;

    xP = xR = xCenter - 3 * h;
    xQ = xCenter + 3 * h;
    yP = yQ = yCenter - 4 * h;
    yR = yCenter + 4 * h;

    square(xQ, yQ, xR, yR, xQ+8*h, yQ+6*h);      /* Правый квадрат   */
    square(xR, yR, xP, yP, xR-8*h, yR)           /* Левый квадрат    */
    square(xP, yP, xQ, yQ, xP, yP-6*h);          /* Нижний квадрат   */
    endgr();
}
```

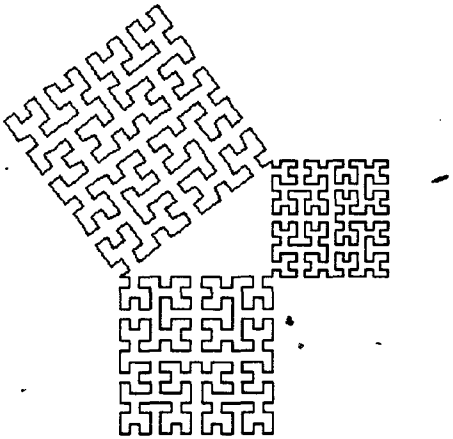


Рис. 4.6. Результат работы программы HILBERT

4.5. КРИВАЯ ДРАКОНА

Кривые с повторяющимся рисунком совсем не обязательно должны генерироваться с помощью рекурсивных функций. Рассмотрим кривую, которую можно получить как бы многократным складыванием длинной полоски бумаги некоторым систематическим образом. Если такую полоску сложить пополам трижды и слегка развернуть, чтобы все углы стали равными 90°, то получим фигуру, показанную на рис. 4.7. Если это окажется слишком трудным для понимания, то следует лишь вообразить, что угол в точке М посредине полоски становится пренебрежимо малым, а когда он станет равным нулю, то снова будем уменьшать угол в середине и так далее.

Будем отслеживать ход кривой, начиная с отрезка прямой линии 1, и на каждом углу поворачивать отрезок на 90° либо вправо, либо влево. Задача заключается в выборе направления поворота (влево или вправо) в конце каждого отрезка прямой линии i ($i = 1, 2, \ldots$). Присвоим код 1 для поворота влево и код 3 для поворота вправо и обозначим код для отрезка прямой линии с номером i через $T(i)$. (Тогда можно сказать, что в конце отрезка прямой линии с номером i мы должны повернуть на $T(i) \times 90°$ влево, поскольку поворот на 90° по часовой стрелке эквивалентен повороту на 270° против часовой стрелки). Из рис. 4.7 можно видеть, что

$$T(1) = 1,\ T(2) = 1,\ T(3) = 3,\ T(4) = 1,\ T(5) = 1,\ T(6) = 3,\ T(7) = 3$$

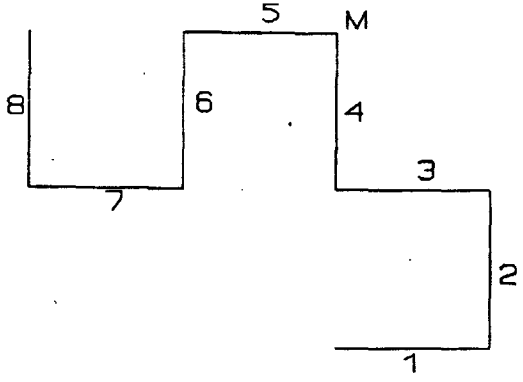


Рис. 4.7. Кривая дракона из восьми элементов

Аналогичным образом можно исследовать подобную кривую из 16 прямолинейных отрезков и так далее. При этом окажется, что функцию T можно определить следующим образом для натурального ряда чисел i, используя операторы / и % для деления нацело и определения остатка при делении целых чисел:

$$T(i) = T(i/2) \quad \text{если } i \text{ четное}$$
$$T(i) = i \,\%\, 4 \quad \text{если } i \text{ нечетное}$$

Хотя предыдущая формула рекурсивна, исследуемая кривая может быть легко получена с помощью нерекурсивной программы. Кривые такого типа иногда называют *кривыми дракона*, чем и объясняется название программы DRAGON. Как и в параграфе 4.3, будем вычислять кривую дважды, чтобы обеспечить размещение кривой в пределах границ экрана. Здесь этот способ может оказаться очень эффектным, поскольку можно использовать целочисленные переменные вместо переменных с плавающей точкой. Это оказывается возможным потому, что все отрезки прямых линий имеют одинаковую длину, а в программе мы можем применять любые единицы длины. Кроме всего прочего, реальные размеры отрезков прямых линий вычисляются только однажды, до их вычерчивания.

Замечательное свойство кривых дракона заключается в том, что в них отсутствуют самопересечения. Это можно очень четко продемонстрировать путем скругления всех углов или, вернее, заменив их небольшими отрезками прямых линий, то есть везде будем иметь углы 135° вместо 90°. Для работы программы DRAGON необходимо ввести только одно число — количество вычерчиваемых отрезков прямых линий (горизонтальных и вертикальных). Хотя программа воспринимает любое число, но при желании получить законченную картинку это число должно представлять собой степень числа 2. Например, рис. 4.8 составлен из 256 отрезков, это число достаточно мало, чтобы ясно различить скругленные углы.

```
/* DRAGON:                Кривая дракона                    */

#include <stdio.h>
#include "grasptc.h"

int x=0, y=0, dx=4, dy=0, n, xmin=10, xmax=-10,
    ymin=10, ymax=-10, ixC, iyC;
float fx, fy, f, xC, yC;
```

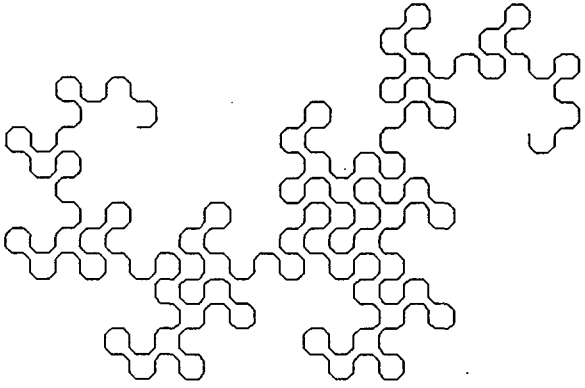


Рис. 4.8. Кривая дракона со скругленными углами

```
float xreal(int x);
float yreal(int y);
void curve(int prescan);
void step(int r, int prescan);

main()

{   printf ( "Сколько отрезков ? (например, 256) : ");
    scanf("%d", &n);
    curve(1);
    initgr( );
    fx = x_max/(xmax - xmin); fy = y_max/(ymax - ymin);
    f = (fx < fy ? fx : fy) * 0.7;
    ixC = (xmin + xmax)/2;
    iyC = (ymin + ymax)/2;
    xC = x_max/2;
    yC = y_max/2;
    x = y = 0; dx=4; dy=0;
    move(xreal(x+dx/4), yreal(y));
    curve(0);
    endgr( );
}

void curve(int prescan)
{   int i, j, r;
    for (i=1; i<=n; i++)
    {   j=i;
        while ((j & 1) == 0) j >>= 1;
        r = j & 3;
        step(r, prescan);
    }
}
```

```
void step(int r, int prescan)

{   int t, dx1, dy1;
    if (prescan)
    {   x += dx; y += dy;
        if (x < xmin) xmin = x;
        if (x > xmax) xmax = x;
        if (y < ymin) ymin = y;
        if (y > ymax) ymax = y;
    } else
    {   dx1 = dx/4; dy1 = dy/4;
        draw(xreal(x + dx1), yreal(y + dy1));
        x += dx; y += dy;
        draw(xreal(x - dx1), yreal(y - dy1));
    }
    if (r == 1) {t = dx; dx = -dy; dy = t;} else
        /* r == 3 */{t = dx; dx = dy; dy = -t;}
}

float xreal(int x)
{   return xC + f * (x - ixC);
}

float yreal(int y)
{   return yC + f * (y - iyC);
}
```

4.6. ОКРУЖНОСТИ И КВАДРАТЫ

Окружности

Можно реализовать множество привлекательных рисунков с помощью функций, которые решают две задачи: во-первых, они вычерчивают некоторую фигуру, скажем, окружность с центром и радиусом, заданными в качестве аргументов, во-вторых, они рекурсивно несколько раз обращаются сами к себе для вычерчивания той же самой фигуры, но меньшей, чем исходная. Таким образом было получено изображение, показанное на рис. 4.9.

Всегда очень раздражает, если пользователь вынужден учитывать размеры экрана. Этого не требуется при использовании программы CIRCLES, с помощью которой было получено изображение на рис. 4.9 и ряд других. Для программы CIRCLES не нужно задавать входные данные об абсолютном размере изображения. Но все-таки необходимо ввести некоторые данные, перечисленные ниже. Соответствующие имена переменных в программе и их конкретные значения указаны в скобках.

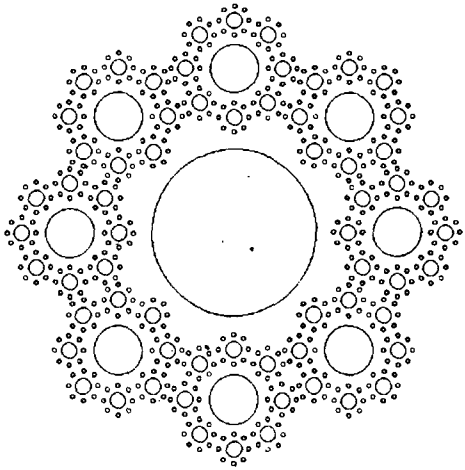


Рис. 4.9. Результат работы программы CIRCLES

1 Глубина рекурсии, показывающая, сколько разных окружностей будет вычерчено ($n = 4$).

2 Коэффициент уменьшения для вычисления радиуса каждой сателлитной окружности относительно радиуса базовой окружности ($f = 0.3$).

3 Коэффициент, на который должен быть умножен радиус окружности, чтобы получить радиус окружности, на которой размещаются центры сателлитных окружностей ($c = 2.0$).

4 Количество окружностей на каждой орбите (*nsatellite* = 8).

5 Код (1 или 0) для индикации, нужен ли файл типа HP-GL. Если да, то имя этого файла будет CIRCLES.HPG (*hpg* = 1).

На рис. 4.9 изображены окружности четырех разных размеров. Допустим, что наибольшая окружность имеет радиус r. Тогда остальные окружности будут иметь радиусы fr, f^2r, f^3r. Радиус наибольшей орбиты определится как $R = cr$. Центры наибольшей окружности и следующей за ней лежат на расстоянии R друг от друга. Тогда fR и f^2R будут радиусами других орбит. Это означает, что расстояние между центрами самой внутренней (наибольшей) окружности и самой внешней (наименьшей) окружности будет равно

$$R + fR + f^2R$$

Поскольку для вычисления значения радиуса R нам необходимо соотнести это выражение с доступным полем на экране, мы должны учесть размеры и самой маленькой окружности — она хотя и маленькая, но все-таки имеет определенные размеры и ее радиус следует добавить к вышеприведенному ряду. Самым простейшим способом решения этой проблемы будет добавление следующего члена ряда (это означает отведение места для следующей орбиты, как если бы значение n было на единицу больше заданного).

Следовательно, если $n = 4$, то будем использовать ряд

$$R + fR + f^2R + f^3R$$

В общем случае это расстояние равно

$$\begin{aligned} s &= R + fR + f^2R + \ldots + f^{n-1}R \\ &= R(1 + f + f^2 + \ldots + f^{n-1}) \\ &= R(1 - f^n)/(1 - f) \end{aligned}$$

(Если читатель не знаком с теорией вычисления суммы конечной геометрической прогрессии, то проверить последнее равенство можно вычислением произведения $(1 - f)(1 + f + f^2 + \ldots + f^{n-1})$, что в результате дает $1 - f^n$). В программе CIRCLES степень f^n вычисляется в переменной p. Поскольку y_max меньше, чем x_max, то важно, чтобы сумма s была не больше, чем $y_max/2$. Из этого следует, что радиусы R и r для наибольшей орбиты и наибольшей окружности соответственно должны быть вычислены как

```
R = 0.5 * y_max * (1-f)/(1-p);
r = R/c;
```

Остальную часть программы CIRCLES понять нетрудно, поэтому детально обсуждать ее не будем.

```
/* CIRCLES:                                                    */
/*          Программа рекурсивного вычерчивания окружностей.   */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "grasptc.h"

float f;
int nsatellite, hpg;
float ccos[100], csin[100];
```

```
void scircle(float x, float y, float r)          /* Только для hpg = 1        */

{   int n=(int)(30 * r + 8), i;
    float theta=2*PI/n;
    move(x+r, y);
    for (i=1; i<=n; i++) draw(x+r*cos(i*theta), y+r*sin(i*theta));
}

void circles(float x, float y, float r, int n)

{   int i, n1=n-1;
    float fr=f*r;

    if (n-->0)
    {   if (hpg == 1) scircle(x, y, r); else circle_uc(x, y, r);
        for (i=0; i<nsatellite; i++)
        circles(x+r*ccos[i], y+r*csin[i], fr, n1);
    }
}

main()

{   int n, i;
    float p=1, r, R, c, theta;

    printf ( "\nГлубина рекурсии (например, 4) : "); scanf("%d", &n);
    printf ( "\nКоэффициент уменьшения (например, 0.3) : ");
    scanf("%f", &f);

    printf ("\nДля окружности с радиусом r орбитой ее сателлитов "
        "будет \nокружность с радиусом cr.\n\nВведите значение"
        " параметра c (например, 2.0) : "); scanf("%f", &c);

    for (i=0; i<n; i++) p *= f;
    printf ( "\nЧисло сателлитных окружностей (например, 8) : ");
    scanf("%d", &nsatellite);
    printf ( "\nВведите 1, если нужен файл HP-GL, или 0 в"
        " противном случае : "); scanf("%d", &hpg);

    if (hpg == 1) fplot = fopen("circles.hpg", "w");
    if (nsatellite > 100) exit(1);
    theta = 2 * PI/nsatellite;

    for (i=0; i<nsatellite; i++)
    {   ccos[i] = c * cos(i * theta);
        csin[i] = c * sin(i * theta);
    }

    initgr( );

    R = 0.5 * y_max * (1-f)/(1-p); r = R/c;
    circles(x_max/2, y_max/2, r, n);
    endgr( );
}
```

Квадраты

Мы можем использовать квадраты (или иные фигуры) таким же образом, как и окружности. Программа SQUARES, приведенная ниже, аналогична программе CIRCLES, которая была только что описана выше, но кроме того в ней демонстрируются некоторые новые аспекты. Во-первых, можно выбрать пустые или заполненные квадраты. Если квадраты пустые, то весь рисунок состоит только из отрезков прямых линий. В этом случае программа одновременно создает файл SQUARES.HPG, содержащий коды HP-GL. При желании программа может также пропустить квадраты, лежащие между каждым сателлитным квадратом и его базовым квадратом. На рис. 4.10 показаны заполненные квадраты, нормально с четырьмя сателлитами для каждого квадрата, тогда как на рис. 4.11 изображено только три из них. На рис. 4.10 сателлиты для каждого квадрата вычерчиваются до вычерчивания самого квадрата, поэтому сателлиты частично закрываются основным квадратом.

Если желательно получить противоположный результат, то для этого в функции *squares* нужно просто переместить вызов функции *square1* на четыре строчки выше, то есть сразу же после открытия скобки.

Изображение на рис. 4.11 представляет собой специальный случай — при выборе $f = 0.4$ и $c = 1.4$, то есть при

$$c = 1 + f$$

промежутка между каждым сателлитом и базовым квадратом не будет. Они не будут и налагаться друг на друга.

```
/* SQUARES                                                          */
/*      :   Программа с рекурсивной функцией черчения квадратов.    */

#define EAST 1
#define NORTH 2
#define WEST 3
#define SOUTH 4

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "grasptc.h"

float f, c;
int filling, all4;
```

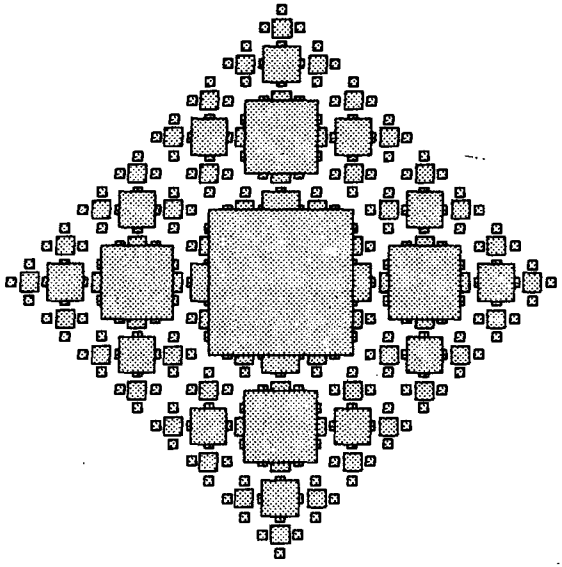


Рис. 4.10. Заполненные квадраты

```
void square1(float x, float y, float r)

{   float xA, yA, xB, yB, xC, yC, xD, yD;
    int points[10];

    xA = x-r; yA = y-r;
    xB = x+r; yB = y-r;
    xC = x+r; yC = y+r;
    xD = x-r; yD = y+r;

    if (filling)
    {   points[0] = IX(xA); points[1] = IY(yA);
        points[2] = IX(xB); points[3] = IY(yB);
        points[4] = IX(xC); points[5] = IY(yC);
        points[6] = IX(xD); points[7] = IY(yD);
        points[8] = points[0]; points[9] = points[1];
        fillpoly(5, points);
    } else

    {   move(xA, yA); draw(xB, yB);
        draw(xC, yC); draw(xD, yD); draw(xA, yA);
        /* Здесь применяются функции 'move' и 'draw' вместо 'drawpoly*/
        /* с целью получения выходного файла SQUARES.HPG.          */
    }
}
```

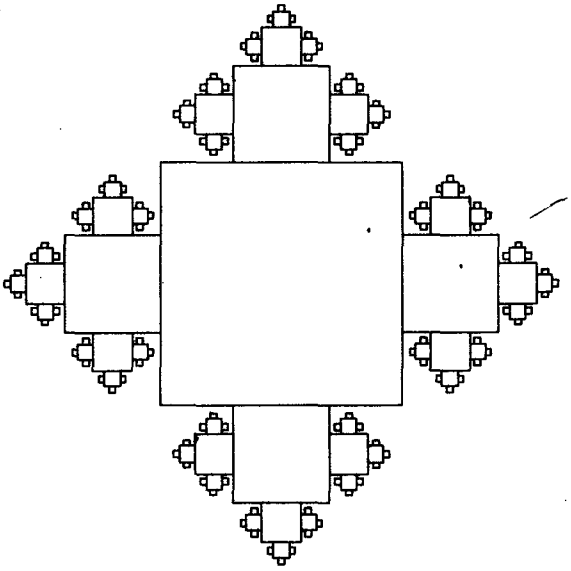


Рис. 4.11. Пустые квадраты

```
void squares(float x, float y, float r, int n, int dir)

{   int n1=n-1;
    float fr=f*r, cr=c*r;
    if (n > 0)
    {   if (dir != WEST || all4) squares(x+cr, y, fr, n1, EAST);
        if (dir != SOUTH || all4) squares(x, y+cr, fr, n1, NORTH);
        if (dir != EAST || all4) squares(x-cr, y, fr, n1, WEST);
        if (dir != NORTH || all4) squares(x, y-cr, fr, n1, SOUTH);
        square1(x, y, r);
    }
}

main()

{   int n, i, m;
    float p=1, r, R;
    printf ( "\nГлубина рекурсии (например, 5) : ");   scanf("%d", &n);
    printf ( "\nКоэффициент уменьшения (например, 0.5) : ");
    scanf("%f", &f);

    printf ("\nДля квадрата с размерами сторон 2r центры сателлитных"
            " квадратов будут \nрасполагаться на расстоянии cr от точки P"
            " — центра исходного квадрата.\n\nВведите значение c "
            "(например, 2.0) : ");      scanf("%f", &c);
```

```
    for (i=0; i<n; i++) p *= f;
    printf ("\nВведите число 1, если квадраты должны быть закрашены,"
            "\nили 0 в противном случае : ");    scanf("%d", &filling);
    if (!filling) fplot = fopen("squares.hpg", "w");

    do
    {   printf ( "\nЖелаете 3 или 4 сателлита для каждого квадрата? ");
        scanf("%d", &m);
    } while (m < 3 || m > 4);
    all4 = (m — 4);

    initgr( );
    setfillstyle(CLOSE_DOT_FILL, foregrcolor);
    R = y_max/2.2 * (1-f)/(1-p); r = R/c;
    squares(x_max/2, y_max/2, r, n, 0);
    endgr( );
}
```

Принцип базовой и сателлитной фигур можно применить и для трехмерного пространства. Тогда будем получать трехмерные объекты (например, сферу) с меньшими сферами в виде сателлитов. Эти сателлиты в свою очередь будут иметь своих сателлитов и так далее. Здесь можно провести аналогию с планетой Земля, которая является сателлитом Солнца, но в то же время имеет своего сателлита — Луну. На рис. 4.12 показана ситуация, в которой сателлиты "с внутренней стороны" опущены, подобно тому, как это делалось с квадратами на рис. 4.11.

Рис. 4.12 был получен с помощью программы SPHERES. В ней выполняются преобразования координат для перехода от исходной "мировой системы координат" (x, y, z) к "системе видовых координат" (x_e, y_e, z_e) с началом системы в точке наблюдателя и осью z, направленной в сторону объекта. Обсуждение такой системы можно найти в книге "*Принципы программирования в машинной графике*" или, в более краткой форме, в книге "*Интерактивная трехмерная машинная графика*".

Но в программе SPHERES алгоритм удаления невидимых линий из этих книг не используется, она основана на принципиально ином способе отображения только видимых сфер или видимых частей.

Во-первых, известно, что сферы всегда видны в виде кругов, независимо от положения наблюдателя. Поэтому можно вычислять только координату положения центра сферы и радиус окружности, отображаемой на экране вместе с координатой положения точки центра по оси z_e. Вместо непосредственного вы-

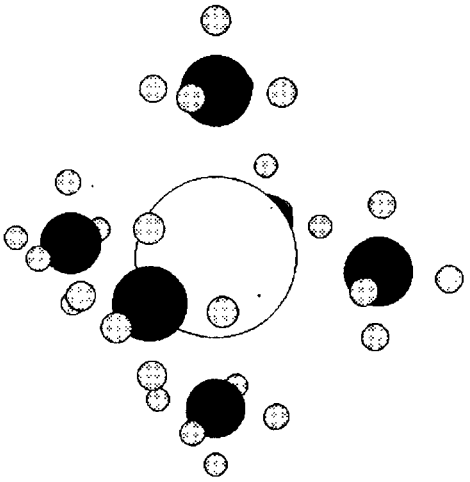


Рис. 4.12. Перспективное изображение сфер

черчивания будем запоминать эти данные в массиве. В качестве ключа для сортировки будем использовать записанное значение z_e. Затем будем вычерчивать изображение всех сфер в порядке уменьшения их значений z_e. Каждая сфера изображается в виде окружности с заданным цветом заполнения. При таком способе более близкие к наблюдателю сферы вычерчиваются самыми последними. Если их место на экране уже было занято другими сферами, то ранее выведенные сферы будут перекрашены. Этот принцип известен как "*алгоритм закраски*", он аналогичен действию художника, рисующего картину.

```
/* SPHERES:                                          */
/*                    Сферы в перспективе            */

#include <stdlib.h>
#include <math.h>
#include "grasptc.h"

#define TABLELENGTH 3000
#define EAST 1
#define WEST 2
#define NORTH 3
#define SOUTH 4
#define UP 5
#define DOWN 6
```

```
int m;
float Xmin=1000, Xmax=-1000, Ymin=1000, Ymax=-1000, C, F;
struct sph {float X, Y, ze, R; int rdepth;} table[TABLELENGTH];

float v11, v12, v13, v21, v22, v23, v32, v33, v43,
        f, XC, YC, xC, yC;

void coeff(float rho, float theta, float phi)
{   float costh, sinth, cosph, sinph;

    costh=cos(theta); sinth=sin(theta);
    cosph=cos(phi); sinph=sin(phi);

    v11=-sinth; v12=-cosph*costh; v13=-sinph*costh;
    v21=costh;  v22=-cosph*sinth; v23=-sinph*sinth;
    v32=sinph;      v33=-cosph;
    v43=rho;
}

void perspective(float x, float y, float z, float *pX, float *pY, float *pze)

{   float xe, ye, ze;
                                    /* Координаты глаза        */
    xe = v11*x + v21*y;
    ye = v12*x + v22*y + v32*z;
    ze = v13*x + v23*y + v33*z + v43;

                                    /* Экранные координаты */
    *pX = xe/ze;
    *pY = ye/ze;
    *pze = ze;
}

void sphere1(float x, float y, float z, float r, int n)

{   float X, Y, Xtop, Ytop, R, ze, zetop;
    perspective(x, y, z, &X, &Y, &ze);

    if (X < Xmin) Xmin = X;
    if (X > Xmax) Xmax = X;
    if (Y < Ymin) Ymin = Y;
    if (Y > Ymax) Ymax = Y;
    perspective(x, y, z+r, &Xtop, &Ytop, &zetop);

    R = Ytop - Y;
    table[m].X = X; table[m].Y = Y; table[m].ze = ze; table[m].R = R;
    table[m].rdepth = n;

    if (++m == TABLELENGTH)
    {   printf   /* "Too many spheres" */
                 ("Слишком много сфер"); exit(1);
    }
    if (m % 20 == 0) printf("m = %d\n", m);
}
```

```
void spheres(float x, float y, float z, float r, float n, float dir)

{   int n1=n-1;
    float fr=F*r, cr=C*r;

    if (n > 0)
    {   sphere1(x, y, z, r, n);
        if (dir != WEST) spheres(x+cr, y, z, fr, n1, EAST);
        if (dir != EAST) spheres(x-cr, y, z, fr, n1, WEST);
        if (dir != SOUTH) spheres(x, y+cr, z, fr, n1, NORTH);
        if (dir != NORTH) spheres(x, y-cr, z, fr, n1, SOUTH);
        if (dir != DOWN) spheres(x, y, z+cr, fr, n1, UP);
        if (dir != UP) spheres(x, y, z-cr, fr, n1, DOWN);
    }
}

float xreal(float X)
{   return xC + f*(X - XC);
}

float yreal(float Y)
{   return yC + f*(Y - YC);
}

void quicksort(int l, int r)
                /* Подобный алгоритм сортировки описан в целом        */
                /* ряде книг, в том числе в книге автора              */
                /* "Программы и структуры данных на языке Си".        */

{   int i=l, j=r;
    float pivot=table[(l+r)/2].ze;
    struct sph w;

    do
    {   while (table[i].ze < pivot) i++;
        while (table[j].ze > pivot) j--;
        if (i > j) break;
        w = table[i]; table[i] = table[j]; table[j] = w;
    } while (++i <= --j);

    if (l < j) quicksort(l, j);
    if (i < r) quicksort(i, r);
}

void main(void)

{   float rho, theta, phi, fx, fy, xx, yy, rr;
    int recdepth, l, X, Y, RX, RY, code, fill;

    printf ( "\nВведите коэффициенты уменьшения c и f (например, 2 0.4) : ");
    scanf("%f %f", &C, &F);
    printf ( "\nРасстояние наблюдения (например, 12) : ");
    scanf("%f", &rho);
```

```
    printf ( "\nЗадайте два угла, в градусах.\n");
    printf("\nTheta — от оси x в горизонтальной плоскости (например, 20) : ");
    scanf("%f", &theta); theta *= PI/180;
    printf ( "\nPhi — измеряемый от оси z в вертикальной плоскости"
                " (например, 75) :  ");
    scanf("%f", &phi); phi *= PI/180;

    printf ( "\nГлубина рекурсии (например, 3) :  "); scanf("%d", &recdepth);
    coeff(rho, theta, phi);
    spheres(0.0, 0.0, 0.0, 1.0, recdepth, 0);
    quicksort(0, m-1);

    initgr( );
    xC = x_max/2;
    yC = y_max/2;
    XC = (Xmin+Xmax)/2;
    YC = (Ymin+Ymax)/2;

    fx = x_max/(Xmax-Xmin);
    fy = y_max/(Ymax-Ymin);
    f = 0.9 * (fx < fy ? fx : fy);

    for (i=m-1; i>=0; i--)
    {   xx = xreal(table[i].X); X = iX(xx);
        yy = yreal(table[i].Y); Y = iY(yy);
        rr = f * table[i].R;
        RX = XPiX(rr); RY = YPiX(rr);
        code = table[i].rdepth % 3;
        fill = (code == 2 ? SOLID_FILL :
                code == 1 ? CLOSE_DOT_FILL : EMPTY_FILL);
        setfillstyle(fill, foregrcolor);
        fillellipse(X, Y, RX, RY);
    }
    endgr( );
}
```

4.7. ФРАКТАЛЫ

На рис. 4.11 все квадраты соединены между собой, поэтому вполне возможно вычертить только внешний контур (огибающую) всей фигуры. Это демонстрируется на рис. 4.13, полученном с помощью программы SQFRACT. При увеличении глубины рекурсии мы получим замкнутую область, граница которой становится очень большой по сравнению с самой областью. Замкнутая кривая, подобная показанной здесь, называется *фрактальной кривой* или просто *фракталом*. Фрактал подобен острову с береговой линией, кажущейся ровной издали, но становящейся совершенно нерегулярной по мере приближения.

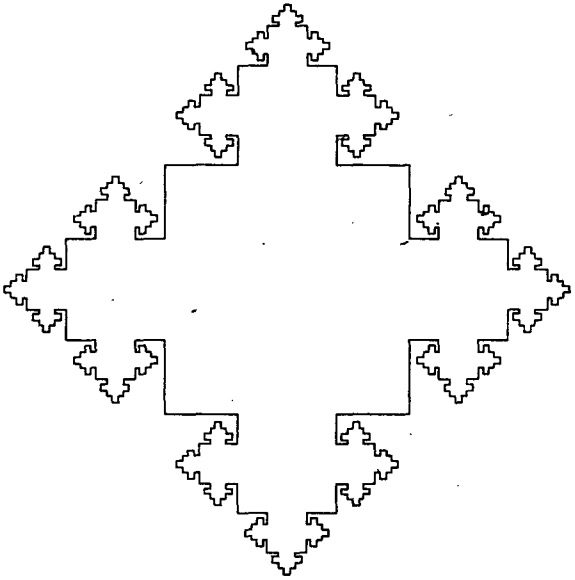


Рис. 4.13. Внешний контур квадратов, вычерченный в виде единой кривой

```
/* SQFRACT:          Вычерчивание фрактала на основе квадратов.      */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "grasptc.h"

float f, fact;

void side(float xA, float yA, float xB, float yB, int n)
{   float xP, yP, xQ, yQ, xR, yR, xS, yS, fdx, fdy;
                    /* Текущая позиция пера в точке (xA, yA)     */
    if (n == 0) draw(xB, yB); else
    {   fdx = fact * (xB-xA);       fdy = fact * (yB-yA);
        xP = xA + fdx;              yP = yA + fdy;
        xS = xB - fdx;              yS = yB - fdy;
        xQ = xP + (yS - yP);        yQ = yP - (xS - xP);
        xR = xQ + (xS - xP);        yR = yQ + (yS - yP);
        draw(xP, yP);
        side(xP, yP, xQ, yQ, n-1);
        side(xQ, yQ, xR, yR, n-1);
        side(xR, yR, xS, yS, n-1);
        draw(xB, yB);
    }
}
```

```
main()
{   Int n, l;
    float p=1, r, R, xC, yC;

    printf  ( "\nГлубина рекурсии (например, 5) : ");
    scanf("%d", &n);

    printf ( "\nКоэффициент уменьшения (например, 0.4) : ");
    scanf("%f", &f);

    for (i=0; i<n; i++) p *= f;
    fact = 0.5*(1-f);

    initgr( );
    R = y_max/2.2 * (1-f)/(1-p); r = R/(1+f);
    xC = x_max/2; yC = y_max/2;
    move(xC-r, yC-r);

    side(xC-r, yC-r, xC+r, yC-r, n);
    side(xC+r, yC-r, xC+r, yC+r, n);
    side(xC+r, yC+r, xC-r, yC+r, n);
    side(xC-r, yC+r, xC-r, yC-r, n);
    endgr( );
}
```

Первые четыре аргумента функции *side* представляют собой координаты обеих концевых точек A и B отрезка прямой линии. Но этот отрезок будет вычерчен только в том случае, если пятый аргумент функции (n) будет равен нулю. В противном случае на этом отрезке будут сформированы две новые точки P и S. Они будут определять концевые точки стороны уменьшенного квадрата (при равенстве $PS = f \bullet AB$), как показано на рис. 4.14. Из рисунка очевидно, что координаты точки Q можно вычислить следующим образом:

$$x_Q = x_P + (y_S - y_P)$$
$$y_Q = y_P + (x_S - x_P)$$

(Напомним, что аналогичный способ формирования новой точки уже обсуждался более подробно при рассмотрении кривой Гильберта в параграфе 4.4). Координаты точки R определяются очень просто, поскольку она находится в таком же отношении к точке Q, как точка S относится к точке P. Теперь можно вычертить отрезок AP, выполнить рекурсивное обращение к функции *side* для сторон PQ, QR, RS меньшего квадрата и закончить вычерчиванием отрезка SB.

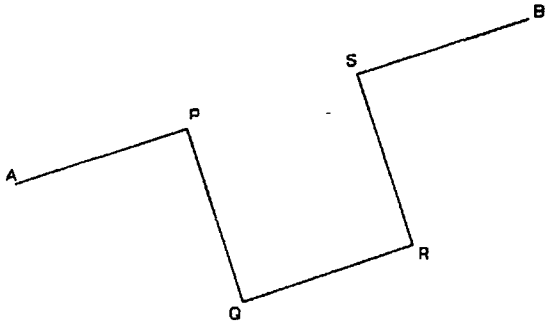


Рис. 4.14. Конструирование точек P, Q, R, S

Вместо непосредственного вычерчивания отрезков AP и SB для них можно рекурсивно обратиться к функции *side*. Хотя в данном конкретном случае это и не дает удовлетворительного результата, но полезна сама идея. Она приводит к целому классу интересных новых кривых, состоящих из отрезков прямых линий, которые в отличие от рис. 4.13, имеют почти одинаковую длину. (Напомним, что с подобной ситуацией мы встречались в случае кривых Гильберта, это обсуждалось в параграфе 3.3).

Рассмотрим общую программу для генерации таких кривых. Во-первых, базовая фигура может быть задана либо в виде горизонтального отрезка прямой линии, либо в виде правильного многоугольника. Во-вторых, вместо вычисления позиций новых точек P, Q, R, S ("*модельных точек*") относительно позиций концевых точек A и B, как на рис. 4.14, пользователь в качестве входных данных может задать любое количество таких точек, не обязательно четыре. Введем локальную систему координат, в которой точка A совпадает с точкой начала (0, 0), а точка B — с точкой (1, 0). Тогда позиции модельных точек могут быть выражены в этих локальных координатах. Рассмотрим для примера рис. 4.15, где определены три новые точки с координатами:

x	*y*
0.45	0
0.50	0.45
0.55	0

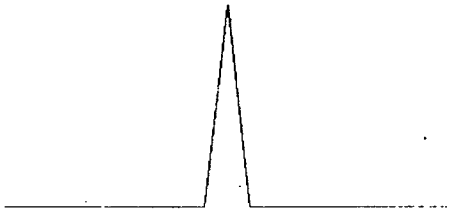


Рис. 4.15. Базовая фигура с тремя новыми точками

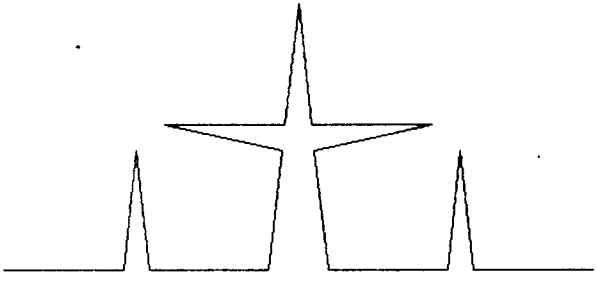


Рис. 4.16. Рекурсивное использование базовой фигуры

Напомним, что обе концевые точки A (0, 0) и B (1, 0) неявно добавляются к модельным точкам, которые мы должны ввести, поэтому вообще число модельных точек всегда на две точки больше, чем заданное число.

Если же всю фигуру целиком применять к каждой из ее четырех частей, то получим рис. 4.16 и так далее. Поскольку мы можем задать правильный многоугольник, например, с четырьмя сторонами, то программа FRCURVE, в которой реализована эта операция, сформирует изображение, показанное на рис. 4.17.

```
/* FRCURVE:                                                    */
/*        Эта программа производит замену либо одного горизон-  */
          тального отрезка, либо каждой стороны правильного много-*/
          угольника фрактальной кривой с любой заданной формой.  */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "grasptc.h"

int nmodel = 0;
float xx[30], yy[30];
```

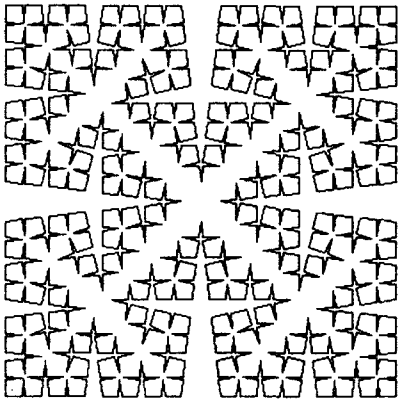


Рис. 4.17. Та же фигура применяется к сторонам квадрата

```
void side ( float xA, float yA, float xB, float yB, int n )

{   int i;
    float x, y, x1, y1, dx=xB-xA, dy=yB-yA;

    if (n == 0)
    {   move(xA, yA); draw(xB, yB);
    } else

    {   x1 = xA; y1 = yA;
        for (i=1; i<=nmodel; i++)
        {   x = x1; y = y1;
            x1 = xA + dx * xx[i] - dy * yy[i];
            y1 = yA + dy * xx[i] + dx * yy[i];
            side(x, y, x1, y1, n-1);
        }
    }
}

main( )

{   int n, k, i;
    float pi, theta, r, x, y, x1, y1, xC, yC,
    sizefactor, xmargin, phi;

    printf ( "\nВведите 1, если в качестве базы берется горизонтальный"
             "отрезок.");
    printf ( "\n\nДля базы в виде правильного k-угольника введите"
             " число k\n");
    printf ( "(например, 4) : ");
    scanf("%d", &k);
```

```
    printf ( "\nГлубина рекурсии (например, 4) : ");
    scanf("%d", &n);
    printf ( "\nСколько опорных точек между (0, 0) и (1, 0) ? ");
    printf   /* "(for example: 3):" */
        ("(например, 3) : ");
    scanf("%d", &nmodel);

    nmodel++;
        /* Параметр nmodel определяет количество отрезков прямой   */
        /* между концевыми точками (0, 0) и (1, 0) базового отрезка.   */
    printf ( "\nВведите пары координат (x, y) опорных точек, исключая"
            " точки (0, 0) и (1, 0). \nНапример:  .45   0\n          "
            " .5   .45\n          .55   0\n\n");

for (i=1; i<nmodel; i++) scanf("%f %f", xx+i, yy+i);
    printf ( "\nКоэффициент размера (1.0 = нормальный размер) : ");
    scanf("%f", &sizefactor);

    xx[nmodel] = 1.0;
        /* Остальные элементы массива инициализируются нулями  */
    initgr( );          initgraph()
    if (k < 3)
    {   xmargin = 0.5*(x_max-sizefactor*(x_max-1));
    side(xmargin, 0.5 * y_max, x_max - xmargin, 0.5 * y_max, n);
    } else

    {   xC = x_max/2; yC = y_max/2;
        r = 0.9 * yC * sizefactor;
        pi = 4 * atan(1.0);
        theta = 2 * pi / k; phi = -0.5 * theta;
        x1 = xC + r * cos(phi); y1 = yC + r * sin(phi);

        for (i=0; i<k; i++)
        {   x = x1; y = y1;
            phi += theta;
            x1 = xC + r * cos(phi);
            y1 = yC + r * sin(phi);
            side(x, y, x1, y1, n);
        }
    }
    endgr( );
}
```

Программу FRCURVE можно использовать для получения большого разнообразия интересных рисунков. Например, если ввести следующие семь пар чисел, определяющие пары координат по оси *x* и *y* для семи модельных точек на единичном интервале, то получим рис. 4.18 (при глубине рекурсий 3), который известен как *фрактал Минковского*.

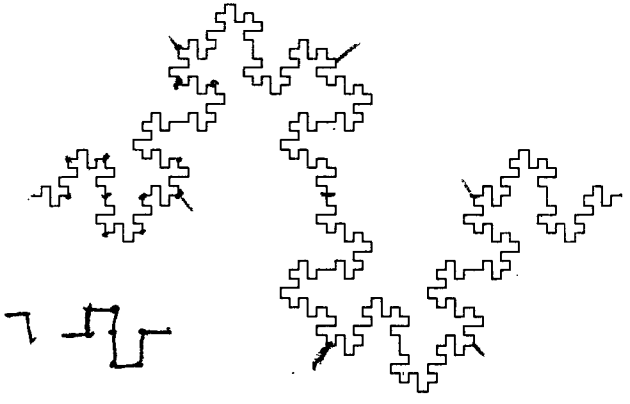


Рис. 4.18. Фрактал Минковского

0.25	0
0.25	0.25
0.5	0.25
0.5	0
0.5	–0.25
0.75	–0.25
0.75	0

Заметим, что наилучшие результаты получаются тогда, когда все отрезки прямой линии, соединяющие заданные модельные точки (включая концевые точки A и B единичного интервала), имеют одинаковую длину, что имело место в нашем предыдущем примере (где их длина равна 0.25).

Это остается верным и для трех вершин равностороннего треугольника (с длиной стороны 1/3); основание которого расположено в середине единичного интервала. Если в качестве базовой фигуры задать большой квадрат, то существуют два способа размещения малого модельного треугольника на сторонах квадрата: они могут быть направлены наружу или внутрь, как показано на рис. 4.19 и рис. 4.20.

При глубине рекурсии $n = 4$ (вместо 1 на рис. 4.19 и рис. 4.20) получим изображения рис. 4.21 и 4.22, известные как кривые фон Коши.

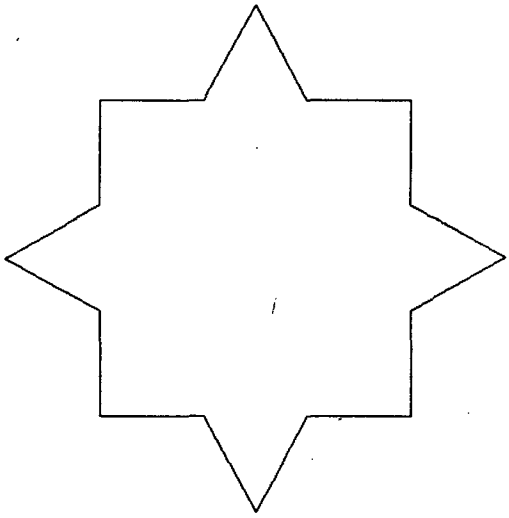


Рис. 4.19. Модельные треугольники направлены наружу

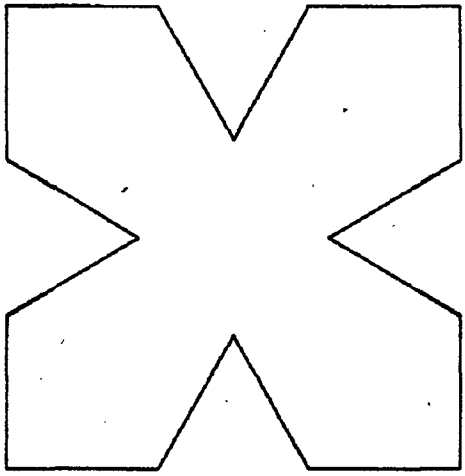


Рис. 4.20. Модельные треугольники направлены внутрь

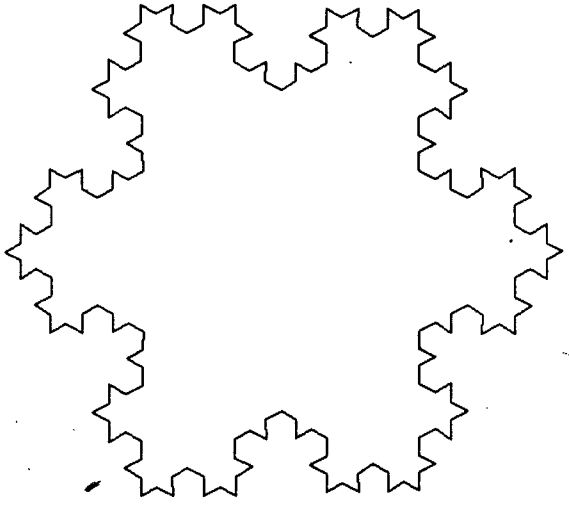


Рис. 4.21. Кривая фон Коши, основанная на треугольниках, направленных наружу

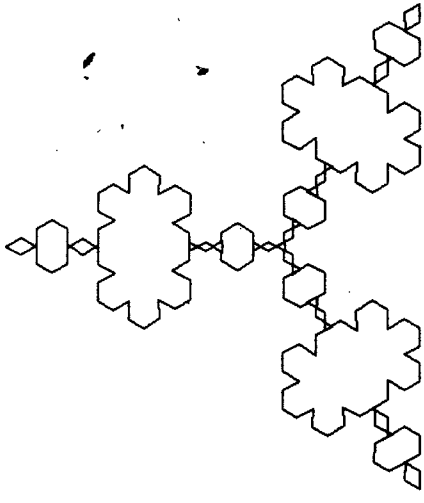


Рис. 4.22. Кривая фон Коши, основанная на треугольниках, направленных внутрь

Глава 5

ИНТЕРАКТИВНАЯ ГРАФИКА

5.1. ГРАФИЧЕСКИЙ ВВОД С ПОМОЩЬЮ "МЫШИ"

Кроме анализа вопросов вывода графической информации необходимо уделить внимание также и вводу графической информации. Среди устройств графического ввода без сомнения самым популярным является устройство типа "мышь". Очень многие знают, как использовать мышь в связи с их любимыми промышленными программными пакетами, но сравнительно мало кто из программистов знает, как ее использовать в качестве устройства ввода в своих собственных программах. Руководства пользователей "мыши" обычно содержат очень много информации об одной сфере применения и очень мало о другой. Обычно можно обнаружить весьма впечатляющий список специальных свойств конкретного варианта реализации "мыши" (часто как на языке Ассемблера, так и на Бейсике), тогда как нас интересуют характеристики, независимые от изделия.

После включения компьютера необходимо установить драйвер мыши, если это не делается автоматически. Например, если драйвер мыши записан в виде файла MOUSE.COM и требуется устанавливать драйвер только при необходимости в нем, то он подключается командной строкой

```
MOUSE
```

Если же драйвер мыши должен устанавливаться при каждом запуске системы (независимо от фактического его использования), то строка вида

```
DEVICE=MOUSE.SYS
```

должна быть включена в файл CONFIG.SYS

При любом способе драйвер "мыши" размещается в памяти, а его пусковой адрес записывается по адресу 51 × 4 = 204, что соответствует вектору прерывания 51 (шестнадцатиричное 0x33). Это число (51) можно обнаружить в следующих двух функциях, через которые мы будем осуществлять связь с драйвером мыши, который в свою очередь реализует связь с самой "мышью".

```
#include <dos.h>

union REGS regs;

int msinit(int Xlo, int Xhi, int Ylo, int Yhi)

{   int retcode;
    regs.x.ax = 0;
    int86(51, &regs, &regs);

    retcode = regs.x.ax;
                    /* -1: мышь установлена;    0: не установлена    */
    if (retcode == 0) return 0;

    regs.x.ax = 7;
    regs.x.cx = Xlo; regs.x.dx = Xhi;
    int86(51, &regs, &regs);

    regs.x.ax = 8;
    regs.x.cx = Ylo; regs.x.dx = Yhi;
    int86(51, &regs, &regs);
    return retcode;
}

void msget(int *pX, int *pY, int *pbuttons)

{   regs.x.ax = 3;
    int86(51, &regs, &regs);

    *pX = regs.x.cx; *pY = regs.x.dx;
    *pbuttons = regs.x.bx;
}
```

Тип *union REGS* определен в заголовочном файле DOS.H. Для нашей переменной *regs* этого типа мы указываем использование машинных регистров AX, BX, CX, DX.

В программах на языке Ассемблера эти регистры можно использовать как до, так и после выполнения так называемого "программного прерывания", для которого в программу должна быть включена команда

```
INT 51
```

Вместо этого мы можем обратиться к функции *int86*, которая определена в DOS.H как

```
int Int86(int Intno, union REGS *inregs, union REGS *outregs);
```

Параметры *intno*, *inregs*, *outregs* обозначают номер прерывания, состояния регистров до обращения и состояния регистров после обращения соответственно. Хотя нам придется выполнять несколько различных операций (называемых "функциями" в руководствах для "мыши"), номер прерывания для мыши всегда должен быть равен 51. Вместо применения различных номеров прерываний нам придется заносить в регистр AX (то есть в переменную *regs.x.ax*) код конкретной операции, которую нужно будет выполнить. В наших функциях *msinit* и *msget* используются четыре таких кода, а именно:

0: Сброс драйвера мыши и вызов состояния,

7: Задание границы по координате *x*,

8: Задание границы по координате *y*,

3: Считывание позиции и состояние кнопок.

Обычно функция *msinit* вызывается только один раз, в самом начале. Затем, как мы вскоре убедимся, будет многократно вызываться функция *msget*. Для переменных *X*, *Y* и *button*, все типа *int*, обращение

```
msget(&X, &Y, &buttons)
```

приведет к занесению текущих значений координат *x* и *y* в переменные *X* и *Y* и ненулевого значения в *buttons*, если была нажата кнопка на корпусе мыши. Очевидно, что это имеет смысл только в том случае, если существует некоторая система координат с минимальными и максимальными значениями для *X* и *Y*. Если бы в функции *msinit* было выполнено только одно первое обращение к функции *int86* (опуская аналогичные обращения с кодами 7 и 8 в аргументе *regs.x.ax*), тогда по умолчанию при перемещении мыши переменным *X* и *Y* будут присвоены следующие значения:

X: 0, 8, 16, ..., 632

Y: 0, 8, 16, ..., 192

То есть по умолчанию диапазон координат основан на цветном графическом адаптере с разрешением 640 × 200 с величиной

шага 8. (Действительные максимальные значения 639 и 199 для осей *X* и *Y* округлены до ближайшего меньшего значения, кратного 8, чем и объясняются числа 632 и 192). Этим же объясняется, почему первые значения для *X* и *Y*, получаемые от функции *msget* до каких-либо перемещений "мыши", равны 320 и 100 соответственно — эти значения соответствуют точке в середине экрана для CGA. При величине шага 8 изменение значений координат по оси *Y* от 0 да 192 означало бы, что существует только 25 позиций по вертикали, что не слишком много. Даже при более высокой разрешающей способности, например, 720 × 348 для HGA, иногда нам требуется более мелкий шаг, чем 8. Отсюда следует, что нам необходимы как большее количество шагов, так и более высокие максимальные значения для *X* и *Y*. Такой результат может быть получен более простым образом, без действительного изменения размера шага.

Предположим, что нужно получить точно столько же значений *X* и *Y*, сколько имеется экранных пикселов на горизонтальной и вертикальной прямых линиях соответственно, что, как обычно, равно числам *X_max* + 1 и *Y_max* + 1. Затем мы можем воспользоваться обращением

```
msinit(0, 8 * X_max, 0, 8 * Y_max);
```

Тогда после каждого обращения к функции *msget* мы можем просто поделить возвращаемые значения *X* и *Y* на 8, чтобы получить желаемые значения пикселных координат. Полученный таким образом результат может быть использован для перемещения *курсора* (который иногда также называется *локатором*) по экрану в соответствии с перемещением "мыши". Однако совсем не обязательно ассоциировать перемещение "мыши" с графическим выводом. Поэтому в нашей первой программе при перемещении "мыши" мы просто будем отображать получаемые координаты в цифровом виде.

При вводе данных от "мыши" имеется одна существенная проблема, которую нельзя игнорировать. Можно конечно, запрограммировать очень простой цикл, в котором производятся обращения к функции *msget*, чтобы узнать значения координат. Но, к сожалению, это привело бы к получению очень длинного списка данных с идентичными значениями, повторенными множество раз. Нам же в действительности нужна функция, которая

возвращает значения *X* и *Y* (или состояние кнопки), если появилось что-то новое. Это означает, что мы должны использовать цикл ожидания, который завершается, как только изменяются координаты *X*, *Y* или произведено нажатие на одну из кнопок.

Однако реализация такого цикла непосредственным образом приведет к новой проблеме: мы можем потребовать выполнения некоторых действий, как только будет нажата клавиша на клавиатуре, но машина не почувствует никакого запроса от клавиатуры, если она ожидает только изменений в состоянии "мыши". Поэтому потребуется включение в цикл обращения к стандартной функции *kbhit*, что и реализовано в функции *msread*, которую можно считать находящейся на более высоком уровне, чем функция *msget*:

```
int msread(int *pX, int *pY, int *pbuttons)
{   static int X0=-1, Y0=-1, but0=-1;
    do
    {   if (kbhit( )) return getch( );
        msget(pX, pY, pbuttons);
    } while (*pX == X0 && *pY == Y0 && *pbuttons == but0);
    X0 = *pX; Y0 = *pY; but0 = *pbuttons;
    return -1;
}
```

Все эти только что описанные функции используются в программе MSDEMO1. Она отображает координаты *X* и *Y*, полученные от "мыши" и поделенные на 8 для получения значений пикселных координат 0, 1, 2, ...

Изображение на экране изменяется лишь в том случае, если "мышь" передвинулась или была нажата одна из ее кнопок. Для "мыши" типа Genius с тремя кнопками ассоциируются значения 1, 4, 2 (слева направо). Если не было нажатия на кнопки, то возвращается значение 0.

```
/* MSDEMO1.C:                                                    */
/*     Демонстрация работы "мыши" в качестве устройства ввода.   */

#include <conio.h>
#include <stdio.h>
#include <dos.h>

union REGS regs;
```

```
int msinit(int Xlo, int Xhi, int Ylo, int Yhi)

{   int retcode;
    regs.x.ax = 0;
    int86(51, &regs, &regs);
    retcode = regs.x.ax;
                    /* -1: мышь установлена; 0: не установлена  */
    if (retcode == 0) return 0;
    regs.x.ax = 7; regs.x.cx = Xlo; regs.x.dx = Xhi;
    int86(51, &regs, &regs);
    regs.x.ax = 8; regs.x.cx = Ylo; regs.x.dx = Yhi;
    int86(51, &regs, &regs);
    return retcode;
}

void msget(int *pX, int *pY, int *pbuttons)

{   regs.x.ax = 3;
    int86(51, &regs, &regs);
    *pX = regs.x.cx; *pY = regs.x.dx;
    *pbuttons = regs.x.bx;
}

int msread(int *pX, int *pY, int *pbuttons)

{   static int X0=-1, Y0=-1, but0=-1;
    do
    {   if (kbhit( )) return getch( );
        msget(pX, pY, pbuttons);
    } while (*pX == X0 && *pY == Y0 && *pbuttons == but0);
    X0 = *pX; Y0 = *pY; but0 = *pbuttons;
    return -1;
}

main( )

{   int buttons, X, Y, X___max=719, Y___max=347;          /* HGA */

    printf ( "\nЭта программа предполагает наличие драйвера ""мыши"".\n"
        "Если это так, то при перемещении ""мыши"" или при нажатии\n"
        "на любую из ее кнопок на экране будут отображаться значения\n"
        " координат X и Y и код нажатой кнопки. Выполнение программы\n"
        " прекращается при нажатии любой клавиши на клавиатуре.\n\n");

    if (msinit(0, 8 * X___max, 0, 8 * Y___max) == 0)
    {   printf ( "n""Мышь"" или драйвер ""мыши"" не установлены."); exit(1);
    }

    while (1)
    {   if (msread(&X, &Y, &buttons) >= 0) break;
        printf ("Кнопка =%2d  X=%5d  Y=%5dn", buttons, X/8, Y/8);
    }
}
```

В этой программе переменным *X_max* и *Y_max* присваиваются значения, соответствующие адаптеру HGA. Если имеется другой графический адаптер, то эти значения необходимо изменить. Конечно, в практической программе специально это делать не нужно, поскольку "мышь" применяется только при работе в графическом режиме, а наш графический модуль GRASPTC и соответствующий заголовочный файл обеспечивают присвоение точных значений переменным *X_max* и *Y_max* после обращения к функции *initgr*.

5.2. ИНТЕРАКТИВНАЯ ДЕМОНСТРАЦИОННАЯ ПРОГРАММА

Теперь, когда мы знаем, как получить входные данные от "мыши", это входное устройство можно использовать в любой интерактивной графической программе. Сначала обсудим программу MSDEMO2, в которой показано, как запрограммировать "мышь" для перемещения курсора по экрану. Эта программа подготовит нас к пониманию более сложной программы SDRAW, приведенной и описанной в параграфе 5.6.

Программа MSDEMO2 не является практической программой, но она очень информативна. Ее можно использовать не только для перемещения курсора, но также и для вычерчивания отрезков прямых либо под управлением "мыши", либо путем ввода координат их концевых точек с клавиатуры. При использовании "мыши" отрезок вычерчивается за четыре шага:

(1) Переместить курсор в начальную точку вычерчиваемого отрезка,

(2) Нажать кнопку на корпусе "мыши",

(3) Переместить курсор в конечную точку отрезка,

(4) Отпустить кнопку на корпусе "мыши".

Для достаточно быстрого перемещения курсора необходимо, чтобы шаг курсора, определяемый перемещением "мыши", был больше, чем один пиксел. С другой стороны, нам бы хотелось иметь доступ с помощью курсора к любому пикселу. Эти два требования кажутся противоречивыми, но мы решим эту проблему допущением возможности использования очень малых шагов курсора путем нажатия на клавиши со стрелками. Так указан-

ные шаги (1) и (3) расширены опцией использования четырех клавиш со стрелками для более точной регулировки.

Как обычно, мы будем применять пользовательские координаты, имеющие следующие преимущества перед пикселными координатами.

- реальная длина единиц (например, в дюймах) не зависит от направления измерения по вертикали или горизонтали;
- ось y направлена вверх.

Новая система пользовательских координат основана на глобальных переменных с плавающей точкой *x_max* и *y_max*. (Напомним, что *x_max* = 10.0 и *y_max* = 7.0; см. также параграф 1.5). Фактически мы будем использовать целочисленные переменные *x* и *y* с максимальными значениями *x_max200* и *y_max200*, которые вычисляются следующим образом:

```
x_max200 = 200 * x_max;
y_max200 = 200 * y_max;
```

Эти значения использованы в обращении к функции *msinit*, введенной в предыдущем параграфе. Поскольку "мышь" выдает значения координат, отличающиеся на восемь единиц, то получим 200 × 10.0/8 − 1= 249 шагов в горизонтальном направлении. Это признано совершенно достаточным (тогда как при значениях 100 или 1000 вместо 200 размер шага оказался бы менее удобным). Так как мы будем использовать новые координаты в командах HP-GL, то для отличия наших новых координат *x* и *y* от пикселных координат *X* и *Y* будем применять буквы HPG, включаемые в качестве расширения имен файлов. Последний тип координат будем использовать для вывода графических изображений на экран, поэтому было бы удобным иметь набор функций для преобразований HPG-координат в пикселные координаты и наоборот. Они должны быть по возможности быстро работающими, поэтому в них не должно выполняться никаких вычислений с плавающей точкой. При отработке заданного значения HPG-координаты *x* нам нужно было бы умножить ее на частное

$$\frac{X_max}{x_max200}$$

чтобы получить значение пикселной координаты. Но вместо использования этого частного в виде числа с плавающей точкой мы сначала умножим координату *x* на *X_max*, а затем поделим на *x_max200*. Обычный тип *int* не позволяет применять большие числа, которые потребуются для этого умножения, поэтому воспользуемся типом *long int*.

При объявлении

```
long x_max200, y_max200;
```

приведенные ниже функции преобразований *Xpixel* и *Ypixel* воспринимают HPG-координаты *x* и *y* в качестве аргументов и вычисляют пикселные координаты *X* и *Y*. Обратные им функции *xhpg* и *yhpg* по заданным пикселным координатам *X* и *Y* в качестве аргументов вычисляют HPG-координаты *x* и *y*:

```
int Xpixel(int x)
{   return (int)((long)X_max * x / x_max200);
}

int xhpg(int X)
{   return (int)(X * x_max200 / X_max);
}

int Ypixel(int y)
{   return Y_max - (int)((long)Y_max * y / y_max200);
}

int yhpg(int Y)
{   return (int)((Y_max - Y) * y_max200 / Y_max);
}
```

Очень хорошим упражнением является проверка специальных случаев

```
Xpixel(0)   = 0          Xpixel(x_max200)   = X_max
xhpg(0)     = 0          xhpg(X_max)        = x_max200

Ypixel(0)   = Y_max      Ypixel(y_max200)   = 0
yhpg(0)     = y_max200   yhpg(Y_max)        = 0
```

Наименьший размер шага "мыши" составляет восемь единиц в HPG-координатах по осям *x* и *y*. А поскольку эти координаты изменяются в более широком диапазоне, чем пикселные координаты *X* и *Y*, то величина шага в восемь единиц не очень велика. Однако иногда может потребоваться доступ к точкам внутри заданной "сетки точек" или, иначе говоря, нам может понадо-

биться поместить курсор в точку, HPG-координаты которой не кратны 8. В этих случаях мы можем применить клавиши со стрелками. Например, при одиночном нажатии на клавишу "стрелка вправо" теоретически курсор переместится на одну HPG-единицу вправо. Практически же курсор может остаться неподвижным, поскольку экран видео-дисплея имеет несколько меньшую разрешающую способность, поэтому из-за ошибки округления мы будем иметь

```
Xpixel(x+1) = Xpixel(x)          или
Xpixel(x+1) = Xpixel(x)+1
```

Таким образом может потребоваться несколько раз нажать на клавишу "стрелка вправо", чтобы заметить реальное перемещение курсора на один пиксел вправо, то же самое будет происходить и при перемещениях курсора в других направлениях. Реализуем эту идею путем введения *параметров коррекции* для *x* и *y*, которые обычно равны нулю, но будут положительными или отрицательными после нажатия на соответствующие клавиши со стрелками. Эти параметры коррекции будут алгебраически добавляться к координатам *xm* и *ym*, выдаваемым "мышью", так что координаты *x* и *y* курсора всегда будут вычисляться как

$$x = xm + xcorr$$
$$y = ym + ycorr$$

где значения *xm* и *ym* кратны 8, а *xcorr* и *ycorr* равны нулю до тех пор, пока не нажата какая-либо из клавиш со стрелками.

Кроме повышения скорости есть и еще другая причина для применения размера шага величиной в восемь единиц. Невидимая *сетка точек* с координатами, кратными 8, позволяет вычерчивать прямые линии, которые действительно горизонтальны или вертикальны и помещать курсор в ту же точку, которая была определена ранее. Теперь предположим, что в результате нажатия на некоторую клавишу со стрелками переменные *xcorr* и *ycorr* стали не нулевыми (и в то же время не кратными 8). Если затем будем продолжать пользоваться только "мышью" то координаты *x* и *y* уже не будут кратны 8,поэтому курсор уже не будет перемещаться по узлам исходной сетки точек, а будет устанавливаться в некоторых новых точках, которые отстоят от исходных точек на *xcorr* единиц вправо и на *ycorr* единиц выше. Это

может оказаться именно тем, что нам нужно, но рано или поздно нам может потребоваться вернуться к исходному набору сетки точек. Это может быть реализовано путем нажатия на клавишу 'Home'. Нажатие на эту клавишу приведет к восстановлению нулевого значения для переменных *xcorr* и *ycorr*, что устанавливает курсор в позицию с координатами $x = xm$, $y = ym$.

Чтобы посмотреть действие HPG-координат в реальных условиях, предусмотрена также возможность ввода координат x_1, y_1 и x_2, y_2 для двух концевых точек P_1 и P_2 отрезка прямой линии, который должен быть вычерчен. Такой способ вычерчивания линий очень общий и точный, но требует большой подготовительной работы.

Рис. 5.1 получен в результате работы программы MSDEMO2. Автор нарисовал эскиз дерева и свои инициалы в нижнем правом

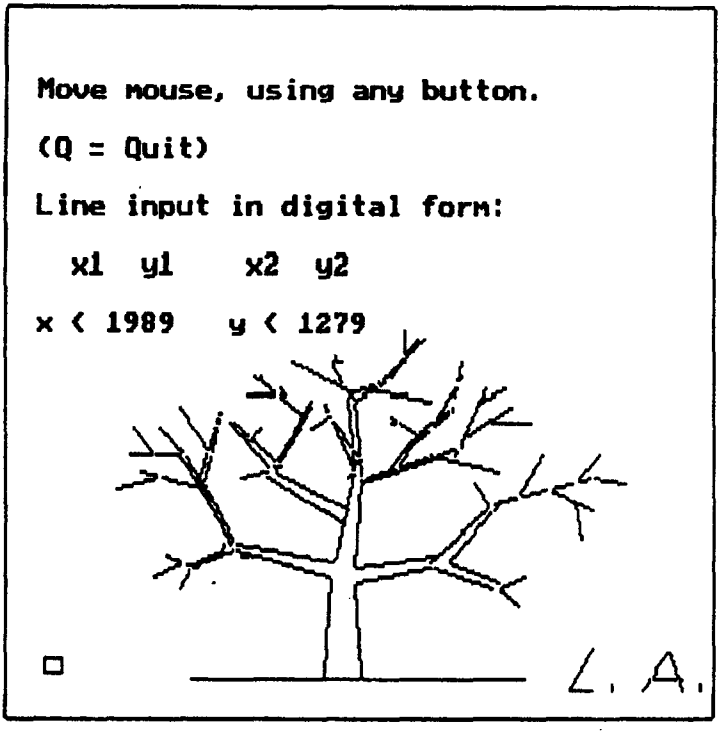


Рис. 5.1. Образец изображения на экране как результат работы программы MSDEMO2

углу путем перемещения "мыши", задавая следующие текстовые строки для вычерчивания ограничивающего квадрата:

```
 100   300   1000   300
1000   300   1000  1200
1000  1200    100  1200
 100  1200    100   300
```

Наиболее существенной частью программы MSDEMO2 является цикл do-while в функции *main*. До тех пор, пока не нажата ни одна из кнопок на корпусе "мыши", переменная *buttons* будет иметь нулевое значение. Функция *newposition*, вызываемая внутри этого цикла, дважды обращается к функции *cursor*: сначала стирается существующий курсор, а затем вычерчивается новый.

Отметим, что функция *newposition* через свои параметры возвращает как HPG-координаты *x* и *y*, так и пикселные координаты *X*, *Y*. Как только кнопка оказывается нажатой, переменная *buttons* получит ненулевое значение, что приведет к вызову функции *getline*. В этот момент становится известной начальная точка отрезка (обозначим ее A), который должен быть вычерчен, и координаты этой точки передаются в функцию *getline* через ее параметры. В функции *getline* любая новая позиция курсора, обнаруженная при нажатой кнопке, интерпретируется как предполагаемая другая концевая точка B отрезка, который подлежит вычерчиванию.

Перемещение "мыши" на этом этапе предполагает перемещение точки B, в то время как точка A остается неподвижной. После этого дважды вычерчивается отрезок AB, сначала до изменения точки B, затем после ее изменения, что приводит к стиранию предыдущего отрезка AB и вычерчиванию нового. Таким образом мы можем "перетаскивать" точку B в желательную позицию, всегда отмечая не только текущую позицию точки B, но и отображая соответствующий отрезок AB. Только отпускание кнопки "мыши" приведет к фиксации текущего отрезка прямой линии AB.

Автор надеется, что читатель найдет и другие интересные программистские детали в приведенном ниже исходном тексте программы.

```
/* MSDEMO2:                                                          */
/*          Демонстрационная программа черчения отрезков прямых      */
           (в качестве подготовки к программе SDRAW, см. 5.6).       */

#include <conio.h>
#include <dos.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

#include "grasptc.h"

union REGS regs;

int Xpixel(int x), Ypixel(int y), xhpg(int X), yhpg(int Y);
int msinit(int Xlo, int Xhi, int Ylo, int Yhi);
int msread(int *pX, int *pY, int *pbuttons);

void getline(int *pxstart, int *pystart);
void drline(int x1, int y1, int x2, int y2);

void digitalline(char ch);
void cursor(int X, int Y);
void newposition(int *px, int *py, int *pX, int *pY, int *pbut);

long x_max200, y_max200;
int y200, xmin, xmax, ymin, ymax, Xmin, Xmax, Ymin, Ymax;

#define txt(linenr, str) gotoxy(1, linenr); cprintf(str);

main()

{   int buttons, xm, ym, X, Y, x, y;
    char str[100];
    clrscr();

    txt(1,"Эта программа требует наличия драйвера ""мыши"".");
    txt(4,"Отрезки прямых можно вычертить следующим образом:");
    txt(5,"Установить курсор в начальную точку отрезка, нажать кнопку");
    txt(6,"на корпусе ""мыши"", переместить курсор в конечную точку");
    txt(7,"отрезка и отпустить кнопку.");
    txt(9,"Нормально курсор может находиться только в точках
                                                        растровой");
    txt(10,"сетки, расположенных на расстоянии восьми единиц друг от");
    txt(11,"друга по горизонтали и по вертикали.");
    txt(12,"При необходимости более точной установки курсора
                                                        можно вос-");
    txt(13,"пользоваться четырьмя клавишами со стрелками. Для возврата");
    txt(14,"курсора к точкам сетки следует нажать клавишу ""Home"".");
    txt(16,"Отрезок прямой можно вычертить также путем ввода четырех");
    txt(17,"чисел x1, y1, x2, y2 — двух пар координат концевых точек.");
    txt(19,"Нажмите любую клавишу на клавиатуре для начала"
            "демонстрации.");
```

```
    if (getch( ) == 3) exit(0);                    /* 3 = Ctrl-C */
    initgr( );
    outtextxy(50, 50, "Перемещайте ""мышь"" при нажатой любой кнопке.");
    outtextxy(50, 70, "(Q = Выход)");
    x_max200 = 200 * x_max; y_max200 = y200 = 200 * y_max;
    Xmin = 4; Xmax = X__max - 4;
    Ymin = 2; Ymax = Y__max - 12;
    xmax = xhpg(Xmax); ymax = yhpg(Ymin);
    xmin = xhpg(Xmin); ymin = yhpg(Ymax);
    outtextxy(50, 90, "Ввод отрезка в цифровой форме: ");
    outtextxy(50, 110, "  x1  y1   x2  y2");
    sprintf(str, "x < %d   y < %d", xmax+1, ymax+1);
    outtextxy(50, 130, str);
    setwritemode(XOR_PUT);

    if (msinit(0, (int)x_max200, 0, (int)y_max200) == 0)
    {   to_text( );
        cprintf ( "Нет ""мыши"" или драйвера ""мыши"" "); exit(1);
    }

    while (msread(&xm, &ym, &buttons) >= 0)  ;
        /* Считывание значений xm и ym и пропуск нажатия клавиш        */
    x = xm; y = ym; cursor(x, y);

    for ( ; ; )                    /* Главный цикл программы          */
    {   newposition(&x, &y, &X, &Y, &buttons);
        if (buttons) getline(&x, &y);
    }
}

        /*          Функции в алфавитном порядке                       */

void cursor(int x, int y)

{   int Xm4, Ym2, Ym1, Xp4, Yp2, Yp1, X, Y;
    X = Xpixel(x); Y = Ypixel(y);
    Xm4=X-4; Xp4=X+4; Ym2=Y-2; Ym1=Y-1; Yp1=Y+1; Yp2=Y+2;
    line(Xm4, Ym2, Xp4, Ym2); line(Xm4, Yp2, Xp4, Yp2);
    line(Xm4, Ym1, Xm4, Yp1); line(Xp4, Ym1, Xp4, Yp1);
}

void digitalline(char ch)

{   int x1, y1, x2, y2, i=0;
    char str[100];
    do
    {   str[i++] = ch; ch = getch( );
    } while (ch != '\n' && ch != '\r');
    str[i] = '\0';
    if (sscanf(str, "%d %d %d %d", &x1, &y1, &x2, &y2) < 4) return;
    drline(x1, y1, x2, y2);
}
```

```
void getline(int *pxstart, int *pystart)

{   int x, y, xstart, ystart, buttons, x0, y0, X, Y;
    x = xstart = *pxstart; X = Xpixel(x);
    y = ystart = *pystart; Y = Ypixel(y);
    do
    {   x0 = x; y0 = y;
        newposition(&x, &y, &X, &Y, &buttons);
        if (x != x0 || y != y0)
        {   drline(xstart, ystart, x0, y0);          /* Стирание    */
            drline(xstart, ystart, x, y);            /* Перенос     */
        }
    } while (buttons);
    *pxstart=x; *pystart=y;
}

void drline(int x1, int y1, int x2, int y2)
{   int X1, Y1, X2, Y2;
    X1 = Xpixel(x1); Y1 = Ypixel(y1);
    X2 = Xpixel(x2); Y2 = Ypixel(y2);
    if (X1 == X2 && Y1 == Y2) return;
    line(X1, Y1, X2, Y2);
}

void endprogram(void)
{   to_text( ); exit(0);
}

int msinit(int Xlo, int Xhi, int Ylo, int Yhi)
{   int retcode;
    regs.x.ax = 0; int86(51, &regs, &regs);
    retcode = regs.x.ax;
            /* -1 = установлена; 0 = не установлена    */
    if (retcode == 0) return 0;
    regs.x.ax = 7; regs.x.cx = Xlo; regs.x.dx = Xhi;
    int86(51, &regs, &regs);
    regs.x.ax = 8; regs.x.cx = Ylo; regs.x.dx = Yhi;
    int86(51, &regs, &regs);
    return retcode;
}

int msread(int *px, int *py, int *pbuttons)
{   static int x0=-10000, y0, but0;
    int xnew, ynew;
    do
    {   if (kbhit( )) return getch( );
        regs.x.ax = 3; int86(51, &regs, &regs);
        xnew = regs.x.cx; ynew = regs.x.dx;
        *pbuttons = regs.x.bx;
    } while (xnew == x0 && ynew == y0 && *pbuttons == but0);
    *px = xnew; *py = y200 - ynew;
```

```
    x0 = xnew; y0 = ynew; but0 = *pbuttons;
    return -1;
}

void newposition(int *px, int *py, int *pX, int *pY, int *pbut)

{   int ch, x0=*px, y0=*py, x, y;
    static int xm, ym, xcorr=0, ycorr=0;
    ch = msread(&xm, &ym, pbut); ch = tolower(ch);
    if (ch >= 0)
    {   if (ch == 0)
        {   ch = getch( );
            switch (ch)
            {   case 72: ycorr++; break;            /* Вверх      */
                case 75: xcorr--; break;            /* Влево      */
                case 77: xcorr++; break;            /* Вправо     */
                case 80: ycorr--; break;            /* Вниз       */
                case 71: xcorr = ycorr = 0; break;   /* Home       */
            }
        }
    }
    if (ch == 'q') endprogram( ); else
    if (isdigit(ch)) digitalline(ch);
    x = xm + xcorr; y = ym + ycorr;
    if (x < xmin) x = xmin;
    if (x > xmax) x = xmax;
    if (y < ymin) y = ymin;
    if (y > ymax) y = ymax;
    if (x != x0 || y != y0)
    {   cursor(x0, y0);                         /* Стирание       */
        cursor(x, y);                           /* Отображение    */
    }
    *px = x; *py = y;
    *pX = Xpixel(x); *pY = Ypixel(y);
}

int xhpg(int X)
{   return (int)(X * x_max200 / X__max);
}

int Xpixel(int x)
{   return (int)((long)X__max * x / x_max200);
}

int yhpg(int Y)
{   return (int)((Y__max - Y) * y_max200 / Y__max);
}

int Ypixel(int y)
{   return Y__max - (int)((long)Y__max * y / y_max200);
}
```

5.3. ОБЛАСТИ ВЫВОДА И ИЗОБРАЖЕНИЯ

Области вывода

В Турбо Си можно определить "*область вывода*" в виде прямоугольника (с горизонтальной стороной), в пределах которого будет выполняться вывод графического изображения. Поскольку мы уже знаем, как выводить графическое изображение в любом месте экрана, то можно высказать сомнение, действительно ли необходима такая концепция. Но предположим, что в графическом режиме на экран выведено несколько строк текста, а затем их нужно заменить другими строками с другим текстом.

Достаточно курьезно, но без специальных мер старый текст не исчезнет с экрана при выводе нового текста на том же месте, что может привести к недоразумениям. Поэтому перед выводом нового текста необходимо очистить область экрана, занимаемую старым текстом. Достаточно эффективный способ реализации такой операции заключается в очистке некоторой области вывода, охватывающей старый текст. В любой момент времени существует только одна текущая область вывода и все пикселные координаты определены только в ней. По умолчанию текущая область вывода охватывает весь экран. Ниже приводятся объявления некоторых функций Турбо Си, которые относятся к областям вывода:

Тип *struct viewporttype*, появившийся в последнем прототипе, определен в заголовочном файле GRAPHICS.H:

```
struct viewporttype
{   int left, top, right, bottom;
    int clipflag;
}
```

Посмотрим, как решить упомянутую выше проблему замены текста. Для очистки на экране любого прямоугольника (с горизонтальной стороной) можно применить следующую функцию:

```
void clearrectangle(int Xtop, int Ytop, int Xbottom, int Ybottom)

{   struct viewporttype vp;
    getviewsettings(&vp);
    setviewport(Xtop, Ytop, Xbottom, Ybottom, 0);
    clearviewport();
    setviewport(vp.left, vp.top, vp.right, vp.bottom, vp.clip);
}
```

В этой функции сначала выполняется обращение к функции *getviewsettings* для вызова информации относительно текущей области вывода (которая может занимать весь экран) и запоминания ее в локальной переменной *vp*. Затем определяется новая область вывода с координатами верхнего левого угла (*Xtop*, *Ytop*) и нижнего правого угла (*Xbottom*, *Ybottom*) только для того, чтобы очистить ее с помощью последующего обращения к функции *clearviewport*.

Наконец восстанавливается установка исходной области вывода, которая была сохранена в локальной переменной *vp*. Функцию *clearrectangle* можно использовать, например, в функции для отображения строки текста в нижней части экрана, со стиранием в этом месте любого предыдущего текста. Здесь применяются глобальные переменные *X_max* и *Y_max*, которые объявлены в файле GRASPTC.H. Нужные значения этим переменным были присвоены в функции *initgr* при обращениях к функциям *getmaxx*( ) и *getmaxy*( ) соответственно.

```
void displaybotlin (char *str)
{   clearrectangle (1, Y_max-10, X_max-1, Y_max-1, 0);
    outtextxy (4, Y_max-10, str);
}
```

Обратите внимание — очистка прямоугольника в этой функции не влияет на выводимые линии на самой границе экрана, которые могли бы быть вычерчены, например, при выполнении обращения к функции

```
rectangle(0, 0, X_max, Y_max);
```

При нашем первом обращении к функции *setviewport* было использовано значение *clipflag* = 0. При значении *clipflag* = 1 вычерчиваемые линии внутри области вывода будут отсекаться по границам области — если в пределах области вывода будет находиться только часть отрезка прямой линии, то только эта часть и будет начерчена. При значении *clipflag* = 0 результат вывода линий, выходящих за пределы области, непредсказуем.

Следует помнить, что пиксельные координаты в функциях вывода графической информации всегда определены относительно области вывода. Это означает, что при отображении текстовой строки в нижней части экрана мы могли бы использовать единственную функцию

```
void displaybotlin1 (char *str)

{   struct viewporttype vp;
    getviewsettings(&vp);
    setviewport(1, Y__max-10, X__max-1, Y__max-1, 0);
    clearviewport( );
    outtextxy (3, 0, str);
    setviewport(vp.left, vp.top, vp.right, vp.bottom, vp.clip);
}
```

вместо двух функций *displaybotlin* и *clearrectangle*. Отметим, что в обращении к функции *outtextxy* заданы пикселные координаты $X = 3$ и $Y = 0$. Они указываются относительно только что определенной области вывода. Сама функция *setviewport* не относится к функциям вывода графической информации, следовательно в обращении к ней должны быть заданы абсолютные координаты, независимые от текущей области вывода.

Функция *clearrecangle* необходима нам в любом случае, поскольку она оказывается полезной и для других целей. Поэтому в действительности мы будем применять функцию *displaybotlin*, а не *displaybotlin*1.

Изображения

В Турбо Си есть три очень полезные функции для считывания изображения с экрана (подобного содержимому области вывода), и запоминания его в памяти с возможностью последующего вызова для отображения на экране. Они объявлены в файле GRAPHICS.H следующим образом:

```
unsigned far imagesize(int left, int top, int right, int bottom);
void    far getimage(int left, int top, int right, int bottom, void far *bitmap);
void    far putimage(int left, int top, void far *bitmap, int op);
```

Перед использованием функции *getimage* можно сделать запрос о количестве байт, требуемых для этой функции. Для этого необходимо просто вызвать функцию *imagesize*. Например, предположим, что на экран выведено некоторое графическое изображение, часть которого временно должна быть заменена некоторым сообщением (хранящимся в строковой переменной *str*), которое занимает одну строку, например, в середине экрана. Одновременно в следующей строке должно быть выведено сообщение

```
нажмите любую клавишу...
```

Как только будет нажата какая-нибудь клавиша, должно быть восстановлено первоначальное изображение. В следующей программе IMAGE, которая, как обычно, должна быть связана в единое целое с модулем GRASPTC, содержится функция *message*, выполняющая эти действия.

```
/* IMAGE:                                                          */
/*        Программа демонстрирует работу функций Турбо Си          */
/*            imagesize, getimage, and putimage.                   */

#include <alloc.h>
#include <conio.h>
#include "grasptc.h"

void clearrectangle(int Xtop, int Ytop, int Xbottom, int Ybottom)

{   struct viewporttype vp;
    getviewsettings(&vp);

    setviewport(Xtop, Ytop, Xbottom, Ybottom, 0);
    clearviewport( );
    setviewport(vp.left, vp.top, vp.right, vp.bottom, vp.clip);
}

void message(char *str)

{   char *buffer;
    int xC=X__max/2, yC=Y__max/2, left, right, top, bottom, h, w, w1;
    h = textheight("A");
    top = yC - 2 * h;
    bottom = yC + 2 * h;

    w = textwidth(str);
    w1 = textwidth("Press any key...");
    if (w1 > w) w = w1;
    w += 16;
    left = xC - w/2; right = xC + w/2;
    buffer = farmalloc(imagesize(left, top, right, bottom));

    if (buffer == NULL) {outtextxy(xC, yC, "Не хватает памяти"); return;}
    getimage(left, top, right, bottom, buffer);
    clearrectangle(left, top, right, bottom);
    settextjustify(CENTER_TEXT, CENTER_TEXT);
    outtextxy(xC, yC-h-h/4, str);
    outtextxy(xC, yC+h, "Press any key...");
    getch( );

    putimage(left, top, buffer, COPY_PUT);
    farfree(buffer);
}

#define N 20
```

```
main()

{   int xC, yC, i, xstep, ystep;
    initgr();
    xC=X_max/2; yC=Y_max/2; xstep = xC/N; ystep = yC/N;
    for (i=1; i<=N; i++) ellipse(xC, yC, 0, 360, i*xstep, i*ystep);
    message   ("Это только пример временной записи"
              " текстовой строки поверх графики.");
    message("Еще раз");
    endgr();
}
```

Основная программа вычерчивает 20 концентрических эллипсов, а после того, как будет завершен их вывод, в центре экрана появятся две строки текста, показанные на рис. 5.2. После нажатия клавиши, запрашиваемого в сообщении на экране, текст в центре экрана заменяется новым сообщением

```
Еще раз
Нажмите любую клавишу...
```

Поскольку первая из этих двух строк занимает гораздо меньше места, чем первая строка на рис. 5.2, то многие из исходных

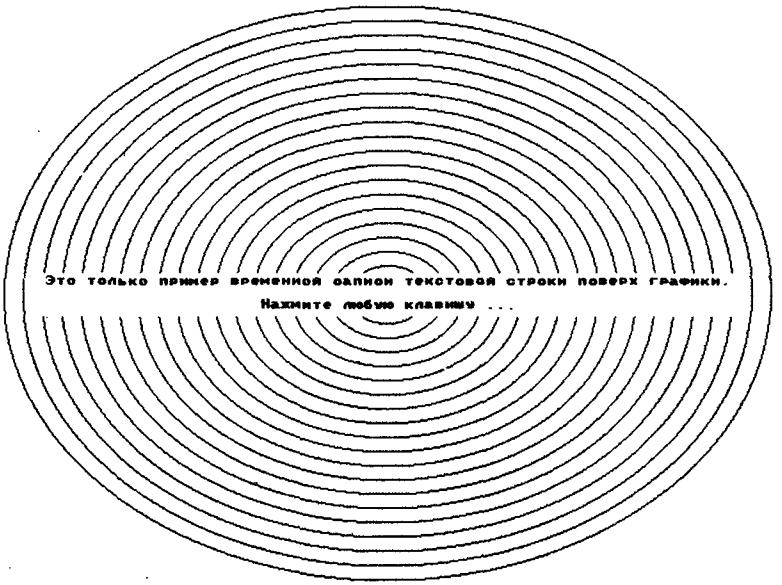


Рис. 5.2. Текст, заменяющий сохраненное изображение

эллипсов будут восстановлены и на этом процесс закончится. После вторичного нажатия на клавишу будут восстановлены все исходные эллипсы.

Как всегда, в завершение нужно опять нажать на какую-нибудь клавишу, чтобы переключиться в текстовый режим и закончить выполнение программы.

В действительности рис. 5.2 был получен с помощью несколько отличающейся версии программы IMAGE, что увидим в следующем параграфе.

Мы еще не обсудили четвертый параметр *op* в обращении к функции *putimage*. Он обозначает код операции, для которого определены следующие возможные значения:

Название	*Значение*
COPY_PUT	0
XOR_PUT	1
OR_PUT	2
AND_PUT	3
NOT_PUT	4

Обычно используется значение *COPY_PUT*, которое и применялось здесь. При этом сохраненное изображение просто налагается на имеющееся на экране. Каждый пиксел, имеющий цвет, выбранный для изображения, считается установленным в 1, а каждый пиксел, имеющий цвет фона, ассоциируется с 0 в экранной памяти компьютера.

При значении *XOR_PUT* над всеми битами сохраненного изображения выполняется операция "*исключительное ИЛИ*" с соответствующими битами изображения на экране. В результате будет инвертировано только такое подмножество пикселов, которое соответствует битам сохраненного изображения, установленным в 1.

При значении *OR_PUT* такие биты не инвертируются, а им присваивается цвет изображения, а при значении *AND_PUT* выбранные пикселы получат цвет изображения только в том случае, если они уже имели этот цвет, в противном случае им будет задан цвет фона. Кроме *COPY_PUT* значения *OR_PUT* и *NOT_PUT* также имеют очень полезные применения, что будет видно из следующего параграфа.

5.4. УТОЛЩЕНИЕ ЛИНИЙ

Часто бывает необходимо включить в отчет или в книгу все графическое изображение, полученное на экране компьютера. Как уже обсуждалось в главе 3, если мы должны использовать графику в битовом отображении, то могут возникнуть проблемы, если изображение на экране значительно больше той картинки, которую нам желательно получить в окончательном результате. Хотя пакеты программ для настольного издательства типа Ventura Publisher или WordPerfect позволяют уменьшить изображение до желательного размера, это не всегда решает нашу проблему, поскольку тонкие линии (и тонкие символы) могут исчезнуть при уменьшении. Поэтому может оказаться желательным сделать линии и символы несколько толще, прежде чем копировать изображение с экрана. Если в нашей программе чертятся только линии, то наилучшим средством обойти эту сложность является использование функции Турбо Си *setlinestyle*, описанной в параграфе 2.8, чтобы увеличить толщину линий до трех пикселов. Однако эта функция не влияет на отображение других пикселов, например, образующих изображения символов. Функция, приведенная ниже, выполняет задачу утолщения линий, символов и других объектов. Она основана на предположении, что (при монохромной графике) был использован цвет изображения для тонких объектов, которые могут быть сделаны толще за счет пикселов, окрашенных в цвет фона.

```
/* FATTER:                                                    */
/*          Эта функция утолщает линии и символы на экране    */

#include <alloc.h>
#include "grasptc.h"

void fatter(void)
{   char *buf;
    buf = farmalloc(imagesize(0, 0, X__max-1, Y__max-1));
    if (buf == NULL)
    {   outtextxy(X__max/2, Y__max/2, "Не хватает памяти");
        return;
    }
    getimage(0, 0, X__max-1, Y__max-1, buf);
    putimage(1, 0, buf, OR_PUT);
    putimage(0, 1, buf, OR_PUT);
    putimage(1, 1, buf, OR_PUT);
    farfree(buf);
}
```

Эта функция просматривает содержимое всего экрана (за исключением столбца $X = X_max$ и строки $Y = Y_max$) и копирует его трижды следующим образом. Если в бите, соответствующем любой позиции (X, Y) изображения, записана единица, то она копируется в позиции $(X+1, Y)$, $(X, Y+1)$ и $(X+1, Y+1)$. С другой стороны, если в бите для этой позиции записан 0, то три упомянутых позиции остаются без изменения. В результате на экране больше не останется ни одной линии толщиной только в один пиксел. (Возможно за исключением только самой правой и самой нижней границ экрана).

Этот способ имеет и недостаток: если между двумя линиями имеется промежуток шириной в один пиксел, то он исчезнет. Практически, как правило, линии располагаются друг от друга дальше, чем один пиксел, так что возможно это не слишком серьезный недостаток. Однако следует быть очень осторожным с символами. Например, при использовании шрифта Турбо Си по умолчанию (с битовым отображением) пустой промежуток в верхней части буквы 'е' по высоте занимает всего один пиксел, поэтому он исчезнет. С другой стороны, вертикальные линии в шрифте по умолчанию уже занимают по ширине два пиксела, а поскольку мы желаем сами увеличить толщину этих линий, то лучше бы подошла ширина в один пиксел. Для шрифта по умолчанию увеличение размера символов (путем использования функции *settextstyle*) приводит к увеличению символов в два раза, что в большинстве случаев значительно больше, чем нам нужно.

Но всегда можно применить штриховые типы шрифтов, что обеспечивает гораздо лучшие результаты в этом отношении. Из различных штриховых шрифтов выберем "малый шрифт", поскольку он обеспечивает формирование символов из элементарных кривых толщиной всего в один пиксел. Очевидно, что это оказывается полезным, если мы будем использовать как увеличенные символы, так и нашу функцию *fatter* (иначе для наших целей этот тип шрифта окажется самым плохим из всех). Для монохромного графического адаптера типа Геркулес (с разрешением 720 × 348) автор считает, что хорошие результаты дают обращения к функциям

```
settextstyle ( SMALL_FONT, HORIZ_DIR, 0 );
setusercharsize ( 3, 2, 3, 2 );
```

Как уже говорилось в параграфе 1.6, последняя функция обеспечивает управление размером текстовой строки для штриховых шрифтов. Мы можем использовать ее только после обращения к функции *settextstyle*, в котором для последнего аргумента (*charsize*) задано значение 0. В вышеприведенном обращении к функции *setusercharsize* и ширина, и высота символов масштабируется с коэффициентом 3.0/2.0. Автор фактически включает обращение к этим двум функциям сразу же после вызова функции *initgr* в программе IMAGE параграфа 5.3, а оператор

```
fatter();
```

включен в функцию *message* этой программы сразу же после

```
outtextxy(xC, yC+h, "Press any key... ");
```

После того как была вызвана функция *fatter*, обращение к функции *getch* приводит к временному останову выполнения программы и, при наличии в памяти утилиты GRAB из пакета WordPerfect, мы можем нажать комбинацию клавиш Shift-Alt-F9 и тем самым запомнить содержимое экрана. Таким образом был сформирован рис. 5.2. Если после этого нажмем другую клавишу, как требует сообщение на экране, то выполнение программы возобновится, но теперь результаты будут не такими хорошими, как были с нашей версией без обращения к функции *fatter*. Напомним, что перед отображением двух текстовых строк в середине эллипсов, мы сохранили изображение в соответствующей части экрана, используя функцию *getimage*, но в тот момент эллипсы еще не были утолщены. Следовательно, теперь, когда мы восстанавливаем части эллипсов с помощью функции *putimage*, восстановленные дуги эллипса остались тонкими, тогда как остальные эллипсы стали толстыми. Это произошло потому, что обращение к функции *fatter* появляется только в специальной версии программы IMAGE, а не в тексте программы, приведенной в предыдущем параграфе.

5.5. МЕНЮ

Рассмотрим теперь как можно перемещать "мышь" для выбора некоторого элемента из меню. Для простоты введем меню, состоящее только из трех элементов *Item A*, *Item B* и *Quit* ("Имя А", "Имя В", "Выход"). Имена этих трех элементов ото-

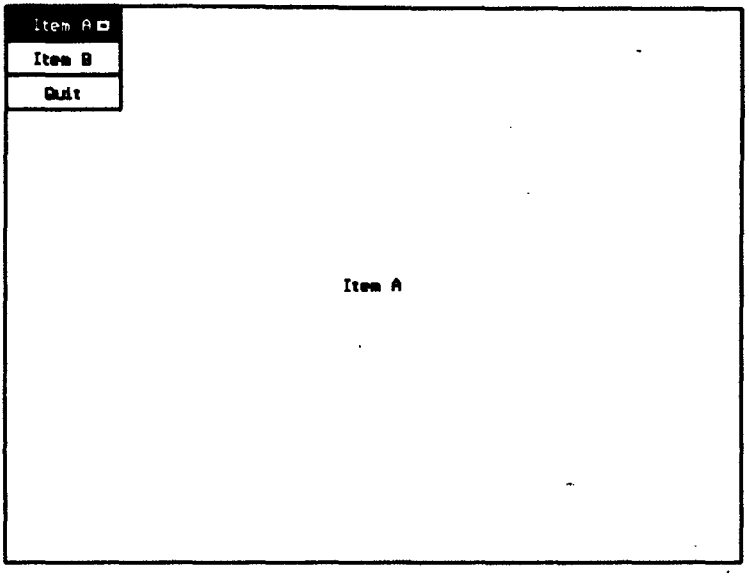


Рис. 5.3. Меню и выбранный элемент

бразим в боксах, представляющих собой прямоугольники, расположенные друг над другом в левой верхней части экрана. Если курсор поместить в один из этих трех прямоугольников и затем нажать кнопку на "мыши", то указанное имя элемента появится в центре экрана. Если именем окажется *Quit* — "Выход", все пикселы этого бокса будут инвертированы примерно на 2 с. На рис. 5.3 показано изображение на экране после выбора элемента "*Item A*".

Способ формирования толстых линий с помощью функции *putimage* как говорилось в предыдущем параграфе, не работает, если часть изображения на экране инвертирована, что мы имеем в данном случае для бокса "*Item A*". Напомним, что при работе функции *putimage* и при режиме *XOR_PUT* увеличивается количество пикселов, имеющих цвет изображения. Обычно это приводит к увеличению толщины линий и символов, поскольку пикселы в них имеют цвет изображения. Однако, если пикселы в части экрана инвертированы, то пикселы, образующие линии и символы в этой части экрана, имеют цвет фона, а фон отображается с цветом изображения. Поскольку при нашем способе

увеличивается количество пикселов с цветом изображения за счет пикселов с цветом фона, то тонкие линии и символы в инвертированной области могут совершенно исчезнуть, вместо того, чтобы стать толще! Поэтому в нашей программе для меню мы не будем использовать функцию *fatter*, а лучше применим функцию Турбо Си *setlinestyle* (описанную в параграфе 2.8), позволяющую вычерчивать линии толщиной в три пиксела. Кроме того, если эта функция совмещается с вызовом

```
setwritemode(XOR_PUT);
```

то толстые линии действительно инвертируются, что мы и будем применять для вычерчивания и стирания курсора. Что же касается символов, то опять будем применять "малый шрифт" с коэффициентом масштабирования 1.5, как это делалось в предыдущем параграфе. Но вместо однократного вычерчивания текста, скажем, в точке (*X*, *Y*), выведем его четыре раза в точках (*X*, *Y*), (*X*+1, *Y*), (*X*, *Y*+1), (*X*+1, *Y*+1). Хотя в руководстве по Турбо Си об этом и не говорится прямо, но режим вывода *XOR_PUT* применим также и для штриховых шрифтов, а это означает, что при четырехкратном выводе текста в очень близких позициях необходимо будет временно возвратиться к режиму записи *COPY_PUT*. Остальные детали программы можно выяснить непосредственно из текста программы MENU:

```
/* MENU:                    Очень простое меню.                    */

#include <dos.h>
#include <stdlib.h>
#include <alloc.h>
#include "grasptc.h"

union REGS regs;

int msinit(int Xlo, int Xhi, int Ylo, int Yhi);
void msget(int *pX, int *pY, int *pbuttons);
void msread(int *pX, int *pY, int *pbuttons);
void mouse_cursor(int *pX, int *pY, int *pbut);
void cursor(int X, int Y);
void clearrectangle(int Xtop, int Ytop, int Xbottom, int Ybottom);
void invertbox(int i, int X, int Y);
void outtxtxy1(int X, int Y, char *str);

char *ptr;

#define WIDTH 100
#define HEIGHT 30
```

```
main( )

{   int buttons, XC, YC, X, Y, i;
    initgr( );
    XC = X__max/2; YC = Y__max/2;
    setlinestyle(SOLID_LINE, 0, THICK_WIDTH);
    settextstyle(SMALL_FONT, HORIZ_DIR, 0);
    setusercharsize(3, 2, 3, 2);
    settextjustify (CENTER_TEXT, CENTER_TEXT);
    ptr = farmalloc(imagesize(1, 1, WIDTH-1, HEIGHT-1));

    if (ptr == NULL)
    {   outtextxy(XC, YC, "Не хватает памяти" );
        delay(1000); exit(1);
    }
    if (msinit(0, X__max, 0, Y__max) == 0)
    {   outtextxy(XC, YC, "Нет ""мыши"" или ее драйвера");
        delay(1000); exit(1);
    }
    rectangle(1, 1, X__max-1, Y__max-1);
    line(WIDTH, 0, WIDTH, 3*HEIGHT);

    for (i=1; i<=3; i++) line(0, i*HEIGHT+1, WIDTH, i*HEIGHT+1);
    outtxtxy1(WIDTH/3, HEIGHT/2,        "Item A");        /* "Буква A"  */
    outtxtxy1(WIDTH/2, 3*HEIGHT/2, "Item B");             /* "Буква B"  */
    outtxtxy1(WIDTH/2, 5*HEIGHT/2, "Quit");               /* "Выход"    */
    msread(&X, &Y, &buttons);
    setwritemode(XOR_PUT);
    cursor(X, Y);        /* Отображение самого первого курсора          */

    do
    {   mouse_cursor(&X, &Y, &buttons);
        if (buttons && X < WIDTH && Y < 3*HEIGHT)
        {   do mouse_cursor(&X, &Y, &buttons); while (buttons);
                /* Теперь кнопка больше не нажимается !                  */
            i = Y/HEIGHT;
            invertbox(i, X, Y);    /* Выделение выбранного элемента     */
            clearrectangle(XC-100, YC-20, XC+100, YC+20);
            switch (i)
            {   case 0: outtxtxy1(XC, YC, "Item A" );     /* "Буква A"  */
                    break;
                case 1: outtxtxy1(XC, YC, "Item B" );     /* "Буква B"  */
                    break;
                case 2: outtxtxy1(XC, YC, "Quit" );       /* "Выход"    */
                    delay(3000); to_text( ); exit(0);
            }
            delay(2000);
            invertbox(i, X, Y);
        }
    } while (1);
}
```

```
void clearrectangle(int Xtop, int Ytop, int Xbottom, int Ybottom)

{   struct viewporttype vp;
    getviewsettings(&vp);
    setviewport(Xtop, Ytop, Xbottom, Ybottom, 0);
    clearviewport( );
    setviewport(vp.left, vp.top, vp.right, vp.bottom, vp.clip);
}

void cursor(int X, int Y)

{   line(X-5, Y-3, X+5, Y-3);
    line(X-5, Y+3, X+5, Y+3);
    line(X-4, Y-1, X-4, Y+1);
    line(X+4, Y-1, X+4, Y+1);
}

void invertbox(int i, int x, int y)

{   int ih=i*HEIGHT;
    cursor(x, y);                          /*   Стирание          */
    getimage(3, ih+3, WIDTH-2, ih+HEIGHT-1, ptr);
    putimage(3, ih+3, ptr, NOT_PUT);
    cursor(x, y);                          /*   Восстановление    */
}

void mouse_cursor(int *pX, int *pY, int *pbut)

{   int Xold=*pX, Yold=*pY;
    msread(pX, pY, pbut);
    cursor(Xold, Yold);        /*  Стирание старого курсора        */
    cursor(*pX, *pY);          /*  Вычерчивание нового курсора     */
}

int msinit(int Xlo, int Xhi, int Ylo, int Yhi)

{   int retcode;
    regs.x.ax = 0;
    int86(51, &regs, &regs);
    retcode = regs.x.ax;
                  /* -1 = мышь установлена; 0 = не установлена      */

    if (retcode == 0) return 0;
    regs.x.ax = 7; regs.x.cx = Xlo; regs.x.dx = Xhi;
    int86(51, &regs, &regs);
    regs.x.ax = 8; regs.x.cx = Ylo; regs.x.dx = Yhi;
    int86(51, &regs, &regs);
    return retcode;
}

void msread(int *px, int *py, int *pbuttons)
{   static int x0=-10000, y0, but0;
    int xnew, ynew;
```

```
    do
    {   regs.x.ax = 3;
        int86(51, &regs, &regs);
        xnew = regs.x.cx; ynew = regs.x.dx;
        *pbuttons = regs.x.bx;
    } while (xnew == x0 && ynew == y0 && *pbuttons == but0);
    *px = x0 = xnew; *py = y0 = ynew;
    but0 = *pbuttons;
}

void outtxtxy1(int X, int Y, char *str)


{   setwritemode(COPY_PUT);
    outtextxy(X, Y, str);
    outtextxy(X+1, Y, str);
    outtextxy(X, Y+1, str);
    outtextxy(X+1, Y+1, str);
    setwritemode(XOR_PUT);
}
```

5.6. ПРОГРАММА ЧЕРЧЕНИЯ

В этом параграфе обсуждается достаточно простая программа черчения SDRAW. Ее можно применить для вычерчивания прямых линий, окружностей, дуг и стрелок и включения текстовых строк при условии доступности "мыши". Полученный чертеж сохраняется в файле в формате HP-GL, что дает возможность передавать его, например, в издательские пакеты WordPerfect или Ventura Publisher. Кроме того "HPG-файл" можно будет прочитать и в программе SDRAW, что очень полезно, если мы захотим отредактировать чертеж.

Будем использовать меню, выбор из которого осуществляется совмещением курсора (управляемого перемещением "мыши") с нужным элементом и нажатием в этот момент кнопки на корпусе "мыши". Хотя программа и не может конкурировать с коммерческими пакетами черчения, но она пригодна для получения разнообразных простых чертежей, например, таких, как рис. 2.8 в параграфе 2.4. В противоположность профессиональным пакетам здесь приводится полный исходный текст программы SDRAW, поэтому читатель может модифицировать и расширять программу в любом направлении. Перед обсуждением некоторых аспектов программирования в SDRAW, посмотрим, что доступно пользователю.

Пользовательские аспекты

Программой SDRAW можно воспользоваться только в том случае, если имеется устройство ввода типа "мышь" и в компьютере установлен драйвер "мыши". Тогда при вводе команды

```
SDRAW
```

на экране появится следующий текст:

```
SDRAW: Программа для вычерчивания отрезков прямых, дуг,
текстовых строк и формирования файлов типа HP-GL; эти файлы
затем могут быть считаны этой же программой или, например,
пакетами WordPerfect 5.0 или Ventura Publisher.
Отрезки прямых линий вычерчиваются следующим образом:
Поместить курсор в начальную точку отрезка и нажать кнопку
на корпусе 'мыши', совместить курсор с конечной точкой
отрезка и отпустить кнопку на корпусе 'мыши'.
Нормально курсор можно поместить только в точки сетки,
находящиеся на расстоянии восьми единиц друг от друга. Если
курсор нужно переместить на более близкое расстояние, то
используйте четыре клавиши со стрелками. При нажатии на
клавишу 'Home' курсор возвратится к ближайшей точке сетки.
Для других операций нужно использовать 'мышь' совместно с
меню и следовать указаниям, отображаемым на нижней строке.
Нажмите любую клавишу на клавиатуре...
```

После нажатия клавиши на экране появится изображение, содержащее три области:

- Рабочая область в виде большого пустого прямоугольника.
- Область сообщений в нижней части экрана, также вначале пустая.
- Главное меню слева.

При работе с программой SDRAW чертеж строится в рабочей области, а сообщения могут отражаться в области сообщений, как показано на рис. 5.4.

В каком-то месте экрана можно увидеть небольшой квадратик — *курсор*. Перемещение "мыши" вызывает соответствующее перемещение курсора в том же направлении. Черчение отрезка прямой линии реализуется очень просто. Необходимо поместить курсор в одну из концевых точек, нажать кнопку и переместить курсор в другую концевую точку, удерживая кнопку в нажатом состоянии. На экране появится отрезок прямой линии, подобный "резиновой нити", соединяющий первую точку (где была нажата

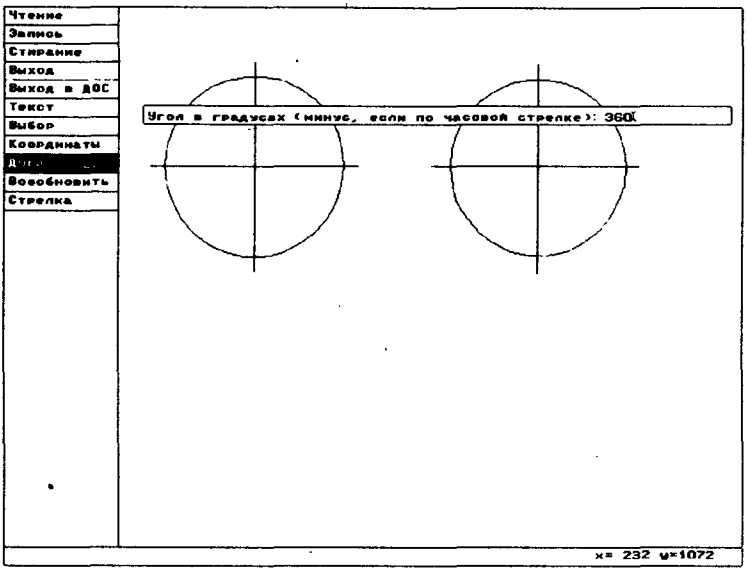


Рис. 5.4. Главное меню и диалоговый бокс

кнопка на корпусе "мыши") с текущей позицией курсора. Когда курсор совпадет с желаемой концевой точкой, нужно отпустить нажатую кнопку и резиновая нить станет фиксированным отрезком прямой линии. Пока работа заключается только в вычерчивании отрезков прямых линий, меню использовать не нужно.

Перемещение "мыши" фактически приводит к ступенчатому движению курсора шагами фиксированного размера. Эти шаги больше одного пиксела и таким образом мы имеем доступ только к *точкам сетки*: расстояние между двумя соседними точками равно размеру шага. При этом легко вернуться к той или иной концевой точке отрезка прямой линии, как это бывает необходимо, например, для вычерчивания последней стороны треугольника (или любого другого многоугольника). Для многих простых эскизов система точек сетки достаточно хороша, чтобы использовать только эти точки в качестве концевых точек отрезков. Но иногда требуется более точное управление перемещением курсора и координаты текущей позиции курсора желательно видеть на экране в цифровой форме. Для этого необходимо совместить курсор с боксом в области меню, обозначенным именем *Coordinates* ("*Координаты*"), и нажать кнопку. После этого

координаты курсора будут отображаться в области сообщения в нижней части экрана, как показано на рис. 5.4. Если перемещать курсор из нижнего левого в верхний правый угол экрана, то будут увеличиваться обе координаты *x* и *y* и так далее. Координаты выражаются в *плоттерных единицах*, которые в 200 раз больше наших обычных пользовательских единиц. Именно эти значения фактически и будут записываться в файл, если в меню будет выбрана опция *Write* ("*Запись*"). Хотя эти значения записываются в виде целых чисел (как это требуется в HP-GL, их диапазон обеспечивает значительно более высокую разрешающую способность, чем диапазон пикселных координат. При использовании "мыши" видно, что размер шага равен восьми плоттерным единицам, так что при этом способе все плоттерные координаты кратны 8. Однако можно задать точки и с другими значениями координат путем использования *клавиш со стрелками*. Например, при однократном нажатии на клавишу "стрелка вправо" (→) значение координаты *x* увеличивается на единицу, то есть, если ее значение было 800, то новое значение будет 801. Теперь при медленном перемещении "мыши" вправо координаты *x* будут принимать значения 809, 817 и так далее, что означает введение новой системы точек сетки. Можно вернуться к старой системе (с координатами, кратными 8) нажатием на клавишу '*Home*'.

В общем случае команда из меню выбирается путем перемещения курсора внутрь бокса меню для этой команды и нажатием кнопки на корпусе "мыши". Обсудим сначала все команды меню, показанные на рис. 5.4.

```
"Чтение"        /* "Read"        */
"Запись"        /* "Write"       */
"Стирание"      /* "Clear"       */
"Выход"         /* "Quit"        */
"Выход в ДОС"   /* "OS shell"    */
"Текст"         /* "Text"        */
"Выбор"         /* "Select"      */
"Координаты"    /* "Coordinates" */
"Дуга"          /* "Arc"         */
"Возобновить"   /* "Refresh"     */
"Стрелка"       /* "Arrow head"  */
```

Команды *Write* ("*Запись*") и *Read* ("*Чтение*") могут быть использованы для сохранения и последующего вызова чертежа,

который был сделан. Файл записывается в формате HP-GL, где для каждого отрезка запоминаются только координаты двух концевых точек. В описываемой версии программа SDRAW не может считывать произвольный файл типа HP-GL, а только такой, какой был сформирован этой же программой. Здесь используется только небольшое подмножество команд HP-GL; это привело не только к упрощению программы, но и помогло избежать ряда других проблем из-за ограничений, налагаемых другими программами. Например, пакет WordPerfect воспринимает только подмножество HP-GL, включая используемое здесь подмножество. Обе команды *Read* ("*Чтение*") и *Write* ("*Запись*") запрашивают от пользователя ввода имени файла; это делается в диалоговом боксе, подобном показанному на рис. 5.4 (что мы вскоре обсудим более подробно), но со следующим текстом:

Имя файла:

Кажется, что такой диалоговый бокс портит уже созданный чертеж, но после удовлетворительного ответа на вопрос в боксе и нажатия на клавишу "Ввод" (Enter) бокс и его содержимое исчезнет и восстанавливается первоначальный чертеж. Если выбрана команда "*Чтение*" ("*Read*"), но файла с заданным именем нет, то в диалоговом боксе появится сообщение:

Нельзя открыть файл; нажмите Esc.

Нажатие клавиши Esc вызовет удаление диалогового бокса. С другой стороны, при задании команды "*Запись*" ("*Write*") и при вводе имени существующего файла в диалоговом боксе появится сообщение

Стереть существующий файл? (Y/N):

Это предотвращает случайное стирание файла.

Команда меню "*Выход*" ("*Quit*") предназначена для завершения работы программы SDRAW и возврата в операционную систему ДОС. Если на экране есть чертеж и он еще не сохранен, то появится запрос:

Сохранить чертеж? (Y/N/Esc):

Кроме ответа "*Да*" или "*Нет*" (*Y/N*) можно также нажать на клавишу Esc, если выход в данный момент нежелателен. Запрос на сохранение чертежа появляется также и в том случае, если

будет задана команда "*Стирание*" ("*Clear*"), которую можно использовать для стирания изображения на экране, чтобы можно было начать формирование нового чертежа.

Возможно также временно выйти из программы SDRAW с помощью команды "*Выход в ДОС*" ("*OS shell*"). Эта команда приводит к появлению нормальной подсказки операционной системы, за которой следует текст

Введите команду EXIT для возврата в SDRAW

Это свойство может быть использовано, например, для просмотра содержимого каталога путем ввода команды *DIR*. После ввода команды *EXIT* на экране будет восстановлен последний чертеж, сформированный программой SDRAW. Такой возврат нужно сделать обязательно и не следует неправильно применять команду "*Выход в ДОС*" вместо "*Выход*" ("*Quit*"), ибо только нормальный выход из программы SDRAW (или нажатием комбинаций клавиш Ctrl-C или Ctrl-Break!) обеспечит освобождение занимаемой ею области памяти и сделает эту память доступной для других целей.

Если в состав чертежа нужно включить текстовые подписи, то, без сомнения, следует выбрать команду "*Текст*" ("*Text*"). Это приведет к отображению следующей текстовой строки в области сообщений в нижней части экрана:

Поместите курсор в точку начала текста и нажмите кнопку

После выполнения этих операций форма курсора изменится (будет изображена фигура, несколько напоминающая букву "I") и ожидается, что текст будет вводиться с отмеченной позиции. Можно ввести только одну строку текста за один раз, при этом нажатие клавиши "*Ввод*" (*Enter*) означает окончание текстовой строки и приводит к появлению нормального курсора в виде квадратика.

Текст, изображенный на рис. 5.5, был введен именно таким образом. (Способ изображения на этом рисунке окружностей и многоугольников опишем несколько позднее).

После задания команды "*Запись*" ("*Write*"), уже упомянутой выше, мы получим HPG файл. Введение его в пакет WordPerfect 5.0 позволило получить рис. 5.6. Интересно сравнить оба эти рисунка.

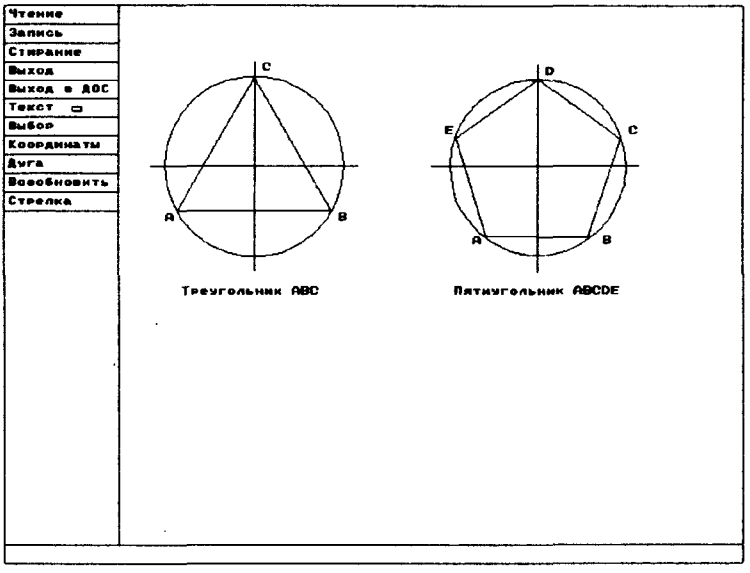


Рис. 5.5. Треугольник и пятиугольник

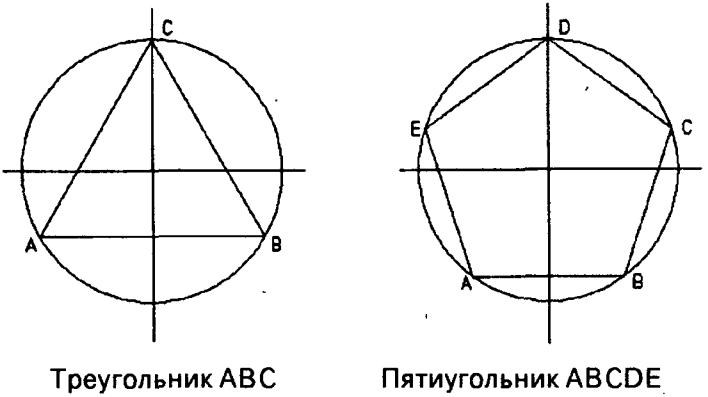


Рис. 5.6. Окончательный чертеж на основе HPG файла

Чтобы увязать между собой начальные позиции нескольких кусков текста, часто бывает полезным иметь на экране отображение значений координат, что, как уже говорилось ранее, делается путем выбора команды "*Координаты*" ("*Coordinates*"). Повторное указание этой команды приводит к прекращению

отображения координат в области сообщений. В этом режиме курсор может быстрее отслеживать перемещение "мыши" (особенно при работе с медленным компьютером PC XT).

При выполнении работы по черчению часто оказывается желательным *удалять* (или *стирать*) некоторые отрезки прямых линий или части текста (все они называются *объектами*). Точно также может потребоваться либо *перенести*, либо *скопировать* объект. Для каждой из этих трех операций нужно указать команду "*Выбор*" ("*Select*"). Объект можно удалить, переместить или скопировать только после выбора его. При совмещении курсора с боксом "*Выбор*" после нажатия кнопки в области сообщений появится следующий текст:

Совместите курсор с выбираемым объектом и нажмите кнопку.

Если выбираемым объектом является отрезок прямой линии, то его можно выбрать совмещением курсора с любым участком этого отрезка; если это текстовая строка, то курсор нужно совместить с начальной позицией, то есть с нижним левым углом первого символа. Нажатие кнопки в этой позиции приведет к появлению нового меню, что показано на рис. 5.7. Как читатель может заметить, это "вторичное меню" накрывает часть главного меню. Теперь в области сообщений появится текст

Укажите нужную операцию.

То есть пользователь должен указать на одну из четырех команд во вторичном меню. Кроме команд "*Удаление*" ("*Delete*"), "*Перенос*" ("*Move*") и "*Копия*" ("*Copy*") в меню имеется команда "*Выход*" ("*Escape*"), которую можно использовать, если, после некоторых раздумий, пользователь ничего не захочет ни удалять, ни переносить, ни копировать. Тогда вторичное меню сразу же исчезнет, а главное меню восстановится полностью.

Если выбранным объектом будет *отрезок прямой линии*, то он отмечается тремя небольшими квадратиками, идентичными курсору. Причем один располагается в средней точке отрезка, а два других — в концевых точках выбранного отрезка. Если теперь задать команду "*Удаление*", то отрезок исчезнет, а главное меню будет восстановлено. Если же будут заданы команды "*Перенос*" или "*Копия*", то в области сообщений появится текст

Совместите со средней или конечной точкой и нажмите кнопку.

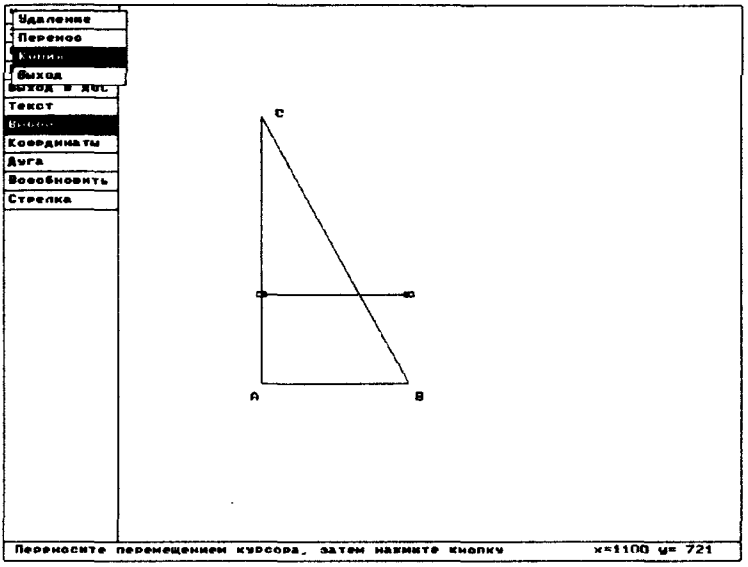


Рис. 5.7. Копирование отрезка AB

После совмещения курсора с некоторой точкой на отрезке или с одной из концевых точек и нажатия на кнопку "мыши" текст в области сообщений изменится на:

Переносите перемещением курсора, затем нажмите кнопку.

Что произойдет после этого, зависит от последней выбранной точки — если это была средняя точка, то перемещение курсора приведет к параллельному переносу отрезка без изменения его длины, как показано на рис. 5.7. Разница между командами "*Перенос*" и "*Копия*" заключается в том, что при выполнении команды "*Перенос*" исходный отрезок исчезает, а при команде "*Копия*" он остается на месте. Если вместо средней точки будет указана концевая точка, то другая концевая точка остается на месте, поэтому перемещаемый или копируемый отрезок изменит свое положение и, возможно, длину.

Если выбранным объектом будет *текст*, то в области сообщений появится такая строка:

Перенесите курсор в нужную точку и нажмите кнопку.

Затем можно осуществить перенос или копирование текстовой строки, как и при перемещении отрезка прямой линии, но без изменения размера строки и ее ориентации.

Команда "*Дуга*" ("*Arc*") позволяет вычерчивать не только дуги окружности, но и полную окружность или правильный многоугольник. Хотя в перечне HP-GL содержатся команды для окружностей и их дуг, в программе SDRAW они не используются, а применяется способ их аппроксимации набором отрезков прямых линий. Основной причиной для этого послужило то, что пакет WordPerfect 5.0 не воспринимает команды HP-GL для окружностей и дуг. Положительной стороной выбора такого способа является возможность применения команды "*Дуга*" для вычерчивания правильных многоугольников. Дуга будет вычерчена на основе заданного центра, начальной точки, величины угла и заданного числа *n*, который указывает, сколько отрезков прямой линии должно быть использовано для аппроксимации дуги. Теперь станут ясными следующие запросы (для ввода числовых данных в диалоговом боксе) и сообщения (в области сообщений), которые последовательно появляются в процессе диалога.

```
Угол в градусах (минус, если по часовой стрелке)
Сколько аппроксимирующих отрезков? (например, 20):
Нажмите на кнопку 'мыши' в позиции центра.
Нажмите на кнопку 'мыши' в позиции начальной точки.
```

На рис. 5.4 можно было видеть первый из запросов в диалоговом боксе. После ответа, например, 360, появится запрос о количестве отрезков. Ответив вводом числа 3, за которым должны быть введены позиции центра и начальной точки, легко получить треугольник, показанный на рис. 5.5.

Для полных окружностей и правильных многоугольников, то есть для угла 360°, ориентация (против или по часовой стрелке) не оказывает влияния на конечный результат. Однако при вводе значения угла меньше, чем 360°, положительное число приведет к вычерчиванию дуги против часовой стрелки. Например, если заданная начальная точка лежит справа от центра, то вычерчивание дуги начнется вверху. Дуга будет вычерчена по часовой стрелке, если вводимому значению угла будет предшествовать знак минус.

По команде "*Возобновить*" ("*Refresh*") сначала стирается все изображение с экрана, а затем все снова перечерчивается на

основе внутренней структуры данных. Эта команда требуется очень редко, но если на экране изображены какие-то непонятные линии и точки, то с помощью этой команды можно уточнить, действительно ли они присущи внутренней структуре. Обычно возобновленное изображение на экране полностью идентично исходному.

Если потребуется изобразить стрелки, то их головки вычерчиваются по команде "*Стрелка*" ("*Arrow head*"). Головка стрелки может быть вычерчена в любой позиции. При задании этой команды появляется сообщение:

Поместите курсор в начало стрелки и нажмите кнопку.

Как только эта операция будет выполнена, появится следующее сообщение

Поместите курсор в конец стрелки и нажмите кнопку.

Так, если нужна стрелка, направленная из точки А в точку В, то сначала нужно начертить отрезок АВ обычным образом, затем использовать точку А (или любую другую точку на этом отрезке) в качестве запрашиваемого "начала стрелки", а потом точку В в качестве "конца стрелки". После этого появится следующий текст в диалоговом боксе:

Длина головки стрелки (например, 40):

Напоминаем, что здесь применяются плоттерные единицы, следовательно, предлагаемое значение соответствует сравнительно большой головке стрелки. Примеры головок стрелок, сформированных программой SDRAW (при длине стрелки 30) можно видеть на рис. 2.8 в параграфе 2.4.

Аспекты программирования

В программе SDRAW применяются только очень простые структуры данных. Имеются две таблицы (с именами переменных *table* и *strtable*) для запоминания отрезков прямых линий и текстовых строк. Хотя они используются как массивы, эти переменные являются указателями, определенными как

```
struct ln_tab_elt (int x1, y1, x2, y2;) *table;
struct str_tab_elt (int x1, y1; char *s;) *strtable;
```

Фактически пространство в памяти для "элементов массива" отводится динамически с помощью функции *farmalloc*.

Поскольку строки имеют переменную длину, то в каждом элементе структуры *strtable*[*i*] запоминаются не сами коды символов, а указатель на них (*s*). Символы, входящие в состав текстовых строк, запоминаются в динамически отводимых областях памяти. Каждый раз, когда вводятся отрезки прямой линии или текстовой строки, они размещаются в конце массивов или *table*, или *strtable*, что можно видеть в функциях *addline* и *addstring*. При удалении отрезка или строки все элементы, следующие за ними в таблице, сдвигаются на одну позицию, поэтому для такого сдвига в программе в функциях *selectlinoptions* и *selecttxtoptions* можно обнаружить соответственно операторы:

```
table[l] = table[l+1];
strtable[l] = strtable[l+1];
```

Как было сказано выше при описании пользовательских аспектов, можно временно перейти в ДОС, аналогично тому, как это делается в интегрированной системе Турбо Си (вводом команд Alt-F, O). Здесь такую же операцию можно выполнить чрезвычайно просто: для фактического перехода к ДОС и возврата в эту же программу все, что нужно сделать, это включить в программу оператор

```
system("");
```

Обычно функция *system* используется для выполнения определенных команд ДОС, которая будет передана этой функции как строковый аргумент. Если это пустая строка, то происходит переход к ДОС, а наша программа остается в памяти. Если затем будет введена команда *EXIT*, то выполнение программы возобновится со следующего оператора после обращения к функции *system*. Для нашей программы ситуация не так проста, поскольку работа происходит в графическом режиме. Поэтому нужно не забывать переключиться в текстовый режим перед обращением к функции *system*. Аналогично, после возврата мы должны перейти в графический режим. Затем нужно восстановить изображение на экране на основе содержимого *table* и *txttable*. Об этом позаботится функция *refresh*. Кстати, эта же функция вызывается и тогда, когда пользователь выбирает команду главного меню "*Возобновить*". Наконец на экране должен быть восстановлен

курсор, он вычерчивается в позиции, определяемой текущими координатами, задаваемыми в качестве аргументов. Ниже приводится функция перехода к ДОС *os_shell* в том виде, как она фактически применяется в программе SDRAW:

```
void os_shell(int x, int y)
{   to_text( ); printf ( "Введите команду EXIT для возврата в SDRAW");
    system("");                          /* Временный выход в ДОС      */
    initgr( ); refresh(x, y);
}
```

Большинство функций, входящих в программу SDRAW подобны функциям, применявшимся в программах MSDEMO1, MSDEMO2 и MSMENU, которые обсуждались в предыдущих параграфах. Поэтому рекомендуется возвратиться к этим программам, если читатель хочет подробно разобраться в работе программы SDRAW, но некоторые детали программы оказались трудными для понимания.

```
/* SDRAW:                                                          */
/*            Черчение линий с применением "мыши" и меню           */

#include <conio.h>
#include <dos.h>
#include <stdio.h>
#include <alloc.h>
#include <process.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
#include <math.h>
#include "grasptc.h"

#define LINTABLEN 8000
#define STRTABLEN 1000
#define MENUWIDTH 100
#define BOXH 16
#define ESC 27

union REGS regs;

void circ_arc(int *px, int *py);
int Xpixel(int x), Ypixel(int y), xhpg(int X), yhpg(int Y);
int msinit(int Xlo, int Xhi, int Ylo, int Yhi);
void msget(int *pX, int *pY, int *pbuttons);
int msread(int *pX, int *pY, int *pbuttons);

void cursor(int X, int Y);
void addline(int Xstart, int Ystart, int X, int Y);
```

```
void addstring(int X, int Y, char *str);
void wrlines(void);
int confirmed(void);
void rdlines(void);
void initmenu(void);
int getstring(int X, int Y, char *mes, char *str, int boxcode);
void clearrectangle(int Xtop, int Ytop, int Xbottom, int Ybottom);
char ermes(char *str);
void displaybotlin(char *str);
void defaultbotlin(void);
void drline(int x1, int y1, int x2, int y2);
void drawnewline(int x1, int y1, int x2, int y2);
void endprogram(void);
void newposition(int *px, int *py, int *pX, int *pY, int *pbut);

void refresh(int x, int y);
void textinput(int *px, int *py, int *pX, int *pY);
void selectobject(int *pX, int *pY);
void getline(int *pXstart, int *pYstart);
void txtcursor(int X, int Y);
void selectlinoptions (int i, int *pX, int *pY, int XX1, Int YY1, int XX2, int YY2);
void selecttxtoptions(int i, int *px, int *py);
void secondmenu(void);
void boxes(int n, int primary);
void invertbox(int i, int x, int y, int primary);
void os_shell(int x, int y);
void arrowhead(int *px, int *py);
int round(float x);

struct lin_tab_elt {int x1, y1, x2, y2;} *table;
struct str_tab_elt {int x1, y1; char *s;} *strtable;

int ntable=0, ntxttable=0, Xmin, Ymin, Xmax, Ymax, xmin, ymin,
xmax, ymax, charheight, charwidth, leftboundary, cursordraw;

char *menuptr, *linbufptr, *boxbuf;
long x_max200, y_max200;
int y200, coordinates=0, modified=0;

#define txt(linenr, str) gotoxy(1, linenr); cprintf(str);

main( )

{   int buttons, xm, ym, i, X, Y, x, y;
    clrscr( );
    txt(1,"SDRAW: Программа для вычерчивания отрезков прямых, дуг,");
    txt(2,"текстовых строк и формирования файлов типа HP-GL; эти файлы");
    txt(3, "затем могут быть считаны этой же программой или, например,");
    txt(4, "пакетами WordPerfect 5.0 или Ventura Publisher.          ");
    txt(6, "Отрезки прямых могут быть вычерчены следующим образом:    ");
    txt(7, "Поместить курсор в начальную точку отрезка и нажать кнопку");
    txt(8, "на корпусе 'мыши', совместить курсор с конечной точкой");
```

```
txt(9, "отрезка и отпустить кнопку на корпусе 'мыши'.          ");
txt(11, "Нормально курсор можно поместить только в точки сетки.");
txt(12,"находящиеся на расстоянии восьми единиц друг от друга. Если");
txt(13, "курсор нужно переместить на более близкое расстояние, то");
txt(14, "используйте четыре клавиши со стрелками. При нажатии на");
txt(15, "клавишу 'Home' курсор возвратится к ближайшей точке сетки.");
txt(17, "Для других операций нужно использовать 'мышь' совместно с");
txt(18, "меню и следовать указаниям, отображаемым на нижней строке.");
txt(20, "Нажмите любую клавишу на клавиатуре..."); getch( );

initgr( );
charwidth = textwidth("A"); charheight = textheight("A");
x_max200 = 200 * x_max; y_max200 = y200 = 200 * y_max;
Xmin = 4; Xmax = X__max - 4;
Ymin = 2; Ymax = Y__max - 16;

table      = farmalloc((long) LINTABLEN * sizeof(struct lin_tab_elt));
strtable   = farmalloc((long) STRTABLEN * sizeof(struct str_tab_elt));
menuptr    = farmalloc((long)imagesize(0, 0, MENUWIDTH+1, Ymax-1));
linbufptr  = farmalloc((long)imagesize(MENUWIDTH, 0, X__max, 16));
boxbuf     = farmalloc((long)imagesize(0, 0, MENUWIDTH-2, BOXH));

if (table == NULL || strtable == NULL || menuptr == NULL ||
    linbufptr == NULL || boxbuf == NULL)
{   to_text( ); cprintf ( "Ошибка памяти"); exit(1);
}

xmax = xhpg(Xmax); ymax = yhpg(Ymin);
xmin = xhpg(Xmin); ymin = yhpg(Ymax);
leftboundary = xhpg(MENUWIDTH+14);

if (msinit(0, (int)x_max200, 0, (int)y_max200) == 0)
{   to_text( ); cprintf ( "Нет 'мыши' или драйвера для нее");
    exit(1);
}

while (msread(&xm, &ym, &buttons) >= 0) ;

x = xm; y = ym;
refresh(x, y);

do
{   newposition(&x, &y, &X, &Y, &buttons);

    If (buttons && X < MENUWIDTH)
    {   do newposition(&x, &y, &X, &Y, &buttons); while (buttons);
            /* Теперь кнопка на корпусе 'мыши' отпущена !          */
        i = Y/BOXH;
        Invertbox(i, x, y, 1);  /* Выделение выбранного элемента   */

        switch (I)
        {   case 0: rdlines( ); break;
            case 1: wrlines( ); break;
```

```
                case 2:                                  /*   Очистка   */
                    if (confirmed( ))
                    {    ntable = ntxttable = 0; modified = 0; refresh(x, y);
                    } else invertbox(i, x, y, 1);
                    break;

                case 3: if (confirmed( )) endprogram( ); break;
                case 4: os_shell(x, y); break;
                case 5:
                    textinput(&x, &y, &X, &Y); break;
                case 6:
                    selectobject(&x, &y); break;
                case 7:
                    coordinates = 1 – coordinates;
                    clearrectangle(512, Ymax+2, Xmax–1, Y__max–1);
                    break;

                case 8:
                    circ_arc(&x, &y); break;
                case 9:
                    refresh(x, y); break;
                case 10: arrowhead(&x, &y);
            }

            if (i != 2 && i != 4 && i != 9) invertbox(i, x, y, 1);
        } else
        if (buttons) getline(&x, &y);
    } while (1);
}


                    /* Функции в алфавитном порядке :        */

void addline(int xstart, int ystart, int x, int y)

{   int aux;
    if (xstart == x && ystart == y) return;

    if (ntable == LINTABLEN)
    {   displaybotlin ( "Таблица отрезкоа заполнена."); return;
    }

    if (xstart > x || xstart == x && ystart > y)
    {   aux = xstart; xstart = x; x = aux;
        aux = ystart; ystart = y; y = aux;
    }

    table[ntable].x1 = xstart; table[ntable].y1 = ystart;
    table[ntable].x2 = x; table[ntable++].y2 = y;
    modified = 1;
}
```

```
void addstring(int x, int y, char *str)

{   char *p;

    if (ntxttable == STRTABLEN)
    {   displaybotlin("Таблица строк заполнена."); return;
    }

    strtable[ntxttable].x1 = x;
    strtable[ntxttable].y1 = y;
    p = farmalloc((long) (strlen(str)+2));

    if (p == NULL) {to_text( );
                    cprintf ( "Проблема с памятью");  exit(1);}

    strcpy(p, str);
    strtable[ntxttable++].s = p;
    modified = 1;
}

void arrowhead(int *px, int *py)

{   int xhead, yhead, xtail, ytail, X, Y, buttons, xar[3], yar[3], i;
    float phi, len, xx[3], yy[3], c, s;
    char str[50];

    displaybotlin (" Поместите курсор в начало стрелки и нажмите кнопку.");

    do newposition(px, py, &X, &Y, &buttons); while (buttons == 0);
    xtail = *px; ytail = *py;
    displaybotlin (" Поместите курсор в конец стрелки и нажмите кнопку.");

    do newposition(px, py, &X, &Y, &buttons); while (buttons == 0);
    xhead = *px; yhead = *py;

    phi = angle(xhead-xtail, yhead-ytail);
    c = cos(phi); s = sin(phi);

    if (getstring(MENUWIDTH+20, 100,
                  "Длина головки стрелки (например, 40): ",
                  str, 1) == 0) return;

    if (sscanf(str, "%f", &len) == 0 || len <= 0) return;
    xx[0] = yy[0] = 0.0; xx[1] = xx[2] = -len;
    yy[1] = 0.3 * len; yy[2] = -yy[1];
    xar[0] = xhead; yar[0] = yhead;

    for (i=1; i<3; i++)
    {   xar[i] = xhead + round(c * xx[i] - s * yy[i]);
        yar[i] = yhead + round(s * xx[i] + c * yy[i]);
        drawnewline(xar[i-1], yar[i-1], xar[i], yar[i]);
    }
    drawnewline(xar[2], yar[2], xhead, yhead);
}
```

```
void boxes(int n, int primary)

{   int i, left, right, offset, level, bottom;
    if (primary)
    {   left = 0; offset = 0; bottom = Ymax;
    } else
    {   left=7; offset=5; bottom = offset + n * BOXH;
    }

    right = left+MENUWIDTH-1;
    clearrectangle(left+1, offset+1, right-1, bottom-1);
    setwritemode(COPY_PUT);

    for (i=0; i<=n; i++)
    {   level = offset+i*BOXH;
        line(left, level, right, level);          /* Горизонтальные линии      */
    }

    line(left, offset, left, bottom);             /* Вертикальные линии        */
    line(right, offset, right, bottom);
    setwritemode(XOR_PUT);
}

void circ_arc(int *px, int *py)

{   char str[100];
    float phi=0, delta, x1, y1, x2, y2, alpha, r, rx, ry;
    int i, n=1, X, Y, buttons, xC, yC, xA, yA;
    if (getstring(MENUWIDTH+20, 100,
                  "Угол в градусах (минус, если по часовой стрелке): ",
                  str, 1) == 0)  return;

    if (sscanf(str, "%f", &phi) == 0)   return;
    phi *= PI/180;                  /* теперь угол phi в радианах          */

    if (getstring(MENUWIDTH+20, 100,
          "Сколько аппроксимирующих отрезков? (например, 20): ",
                  str, 1) == 0)  return;

    if (sscanf(str, "%d", &n) == 0 || n <= 0) return;
    displaybotlin ( " Нажмите на кнопку 'мыши' в позиции центра.");
    do newposition(px, py, &X, &Y, &buttons); while (buttons == 0);
    xC = *px; yC = *py; cursor(xC, yC);
                            /* Маркер остается в позиции центра            */
    cursordraw = 1 - cursordraw;
    displaybotlin ("Нажмите на кнопку 'мыши' в позиции начальной точки.");

    do newposition(px, py, &X, &Y, &buttons); while (buttons == 0);
    defaultbotlin( );
    cursor(xC, yC);                 /* Стирание курсора в центре           */
    cursordraw = 1 - cursordraw;
    xA = *px; yA = *py; delta = phi/n;
    x1 = xA; y1 = yA; rx = x1-xC; ry = y1-yC;
```

```
    alpha = angle(rx, ry);      /* Функция 'angle' определена в LINDRAW  */
    r = sqrt(rx * rx + ry * ry);
    cursor(xA, yA);
            /* Нужно стирание курсора из-за режима COPY_PUT          */
    setwritemode(COPY_PUT);

    for (i=0; i<n; i++)
    {   alpha += delta;
        x2 = xC + r * cos(alpha); y2 = yC + r * sin(alpha);
        drawnewline(x1, y1, x2, y2);
        x1 = x2; y1 = y2;
    }

    setwritemode(XOR_PUT);
    cursor(xA, yA);                /* Вычерчивание курсора                  */
}

void clearrectangle(int Xtop, int Ytop, int Xbottom,int Ybottom)

{   struct viewporttype vp;
    getviewsettings(&vp);
    setviewport(Xtop, Ytop, Xbottom, Ybottom, 0);
    clearviewport( );
    setviewport(vp.left, vp.top, vp.right, vp.bottom, vp.clip);
}

int confirmed(void)

{   char ch;
    if (ntable + ntxttable > 0 && modified)
    {   ch = ermes ( "Сохранить чертеж? (Y/N/Esc): ");
        ch = tolower(ch);
        if (ch == ESC) return 0;
        if (ch == 'y') wrlines( );
    }
    return 1;
}

void cursor(int x, int y)

{   int Xm4, Xp4, Ym2, Yp2, Ym1, Yp1, X, Y;
    static char str[50];

    X = Xpixel(x); Y = Ypixel(y);
    Xm4=X-4; Xp4=X+4; Ym2=Y-2; Yp2=Y+2; Ym1=Y-1; Yp1=Y+1;
    line(Xm4, Ym2, Xp4, Ym2); line(Xm4, Yp2, Xp4, Yp2);
    line(Xm4, Ym1, Xm4, Yp1); line(Xp4, Ym1, Xp4, Yp1);

    if (coordinates && cursordraw)          /* См. пояснения ниже     */
    {   sprintf(str, "x=%4d y=%4d", x, y);
        clearrectangle(512, Ymax+2, Xmax-1, Y__max-1);
        outtextxy(512, Y__max-5, str);
    }
```

```
    cursordraw = 1 -cursordraw;
            /* Значения переменной 'cursordraw'     1, 0, 1, 0, 1, ...   */
            /* См. также функцию  refresh( ).                            */
}

void defaultbotlin(void)
{   displaybotlin("");
}

void displaybotlin(char *str)

{   clearrectangle(1, Ymax+2, X__max-1, Y__max-1);
    outtextxy(2, Y__max-5, str);
}

void drawnewline(int x1, int y1, int x2, int y2)

{   drline(x1, y1, x2, y2);
    addline(x1, y1, x2, y2);
}

void drline(int x1, int y1, int x2, int y2)

{   int aux, X1, Y1, X2, Y2;
    X1 = Xpixel(x1); Y1 = Ypixel(y1);
    X2 = Xpixel(x2); Y2 = Ypixel(y2);

    if (X1 == X2 && Y1 == Y2) return;

    if (X1 > X2 || X1 == X2 && Y1 > Y2)
    {   aux = X1; X1 = X2; X2 = aux;
        aux = Y1; Y1 = Y2; Y2 = aux;
    }

    if (X1 < MENUWIDTH+8) X1 = MENUWIDTH+8;
    line(X1, Y1, X2, Y2);
}

void endprogram(void)
{   to_text( ); exit(0);
}

char ermes(char *str)
{   int left=MENUWIDTH+20;
    char ch;

    getimage(left, 100, Xmax, 116, linbufptr);
    clearrectangle(left, 100, Xmax, 114);
    outtextxy(left+8, 112, str);

    ch = getch( );
    putimage(left, 100, linbufptr, COPY_PUT);
    return ch;
}
```

```
void getline(int *pxstart, int *pystart)

{   int x, y, xstart, ystart, buttons, x0, y0, X, Y;
    x = xstart = *pxstart; X = Xpixel(x);
    y = ystart = *pystart; Y = Ypixel(y);

    do
    {   x0 = x; y0 = y;
        newposition(&x, &y, &X, &Y, &buttons);

        if (x != x0 || y != y0)
        {   drline(xstart, ystart, x0, y0);          /*    Стирание      */
            if (x < leftboundary) {*pxstart=x; *pystart=y; return;}
            drline(xstart, ystart, x, y);            /*    Перенос       */
        }
    } while (buttons);

    addline(xstart, ystart, x, y);
    *pxstart=x; *pystart=y;
}

int getstring(int X, int Y, char *mes, char *str, int boxcode)
{   int len=strlen(mes), i=0, j, maxlen, k;
    char ch, s2[2];
    maxlen=X__max-10-X; s2[1] = '\0';

    if (boxcode)
    {   getimage(X, Y-14, X__max-10, Y+2, linbufptr);
        clearrectangle(X, Y-14, X__max-10, Y+2);
        rectangle(X, Y-14, X__max-10, Y+2);
        outtextxy(X+4, Y, mes);
    }

    while(1)
    {   j = (len+i)*charwidth;
        if (j >= maxlen) break;
        txtcursor(X+j, Y-6);      /* Вычерчивание текстового курсора     */
        ch = getch();
        txtcursor(X+j, Y-6);      /* Стирание текстового курсора         */
        if (ch == ESC) {i=0; break;}
        if (ch == '\n' || ch == '\r') break;

        if (ch == 8)              /* Стирание символа                    */
        {   if (--i < 0) i = 0;
            k = (len+i)*charwidth;
            clearrectangle(X+k, Y-charheight, X+k+charwidth, Y);
        } else
        {   str[i] = s2[0] = ch;
            outtextxy(X+j, Y, s2);
            i++;
        }
```

```
    str[i] = '\0';
    if (boxcode) putimage(X, Y-14, linbufptr, COPY_PUT);
    return *str;
}

void initmenu(void)

{   int Y=BOXH/2+4;
    boxes(11, 1);

    outtextxy(4, Y,      "Чтение");              /* "Read"          */
    outtextxy(4, Y+=BOXH, "Запись");             /* "Write"         */
    outtextxy(4, Y+=BOXH, "Стирание");           /* "Clear"         */
    outtextxy(4, Y+=BOXH, "Выход");              /* "Quit"          */
    outtextxy(4, Y+=BOXH, "Выход в ДОС");        /* "OS shell"      */
    outtextxy(4, Y+=BOXH, "Текст");              /* "Text"          */
    outtextxy(4, Y+=BOXH, "Выбор");              /* "Select"        */
    outtextxy(4, Y+=BOXH, "Координаты");         /* "Coordinates"   */
    outtextxy(4, Y+=BOXH, "Дуга");               /* "Arc"           */
    outtextxy(4, Y+=BOXH, "Возобновить");        /* "Refresh"       */
    outtextxy(4, Y+=BOXH, "Стрелка");            /* "Arrow head"    */
}

void invertbox(int i, int x, int y, int primary)

{   int left, right, offset, level;
                                /*  См. также функцию 'boxes'      */

    if (primary) {left=0; offset=0;} else {left=7; offset=5;}
    right = left+MENUWIDTH-1;
    level = offset + i * BOXH;
    cursor(x, y);                       /*    Стирание             */

    getimage(left+1, level+1,  right-1, level+BOXH-1, boxbuf);
    putimage(left+1, level+1, boxbuf, NOT_PUT);
    cursor(x, y);                       /*    Восстановление       */
}

int msinit(int Xlo, int Xhi, int Ylo, int Yhi)

{   int retcode;
    regs.x.ax = 0;
    int86(51, &regs, &regs);
    retcode = regs.x.ax;        /* -1: установлен; 0: не установлен  */

    if (retcode == 0) return 0;
    regs.x.ax = 7; regs.x.cx = Xlo; regs.x.dx = Xhi;
    int86(51, &regs, &regs);

    regs.x.ax = 8; regs.x.cx = Ylo; regs.x.dx = Yhi;
    int86(51, &regs, &regs);
    return retcode;
}
```

```
int msread(int *px, int *py, int *pbuttons)

{   static int x0=-10000, y0, but0;
    int xnew, ynew;

    do
    {   if (kbhit( )) return getch( );          /* Ввод с клавиатуры          */
        regs.x.ax = 3;
        int86(51, &regs, &regs);
        xnew = regs.x.cx; ynew = regs.x.dx;
        *pbuttons = regs.x.bx;
    } while (xnew == x0 && ynew == y0 && *pbuttons == but0);
    *px = xnew; *py = y200 - ynew;
    x0 = xnew; y0 = ynew; but0 = *pbuttons;
    return -1;                      /* -1 : Не было нажатия на клавишу        */
}

void newposition(int *px, int *py, int *pX, int *pY, int *pbut)

{   int ch, x0=*px, y0=*py, x, y;
    static int xm, ym, xcorr=0, ycorr=0;
    ch = msread(&xm, &ym, pbut); ch = tolower(ch);

    if (ch >= 0)
    {   if (ch == 0)            /* Arrow keys are read as two characters,     */
        {   ch = getch( );      /* the first of which is '\0'.                */
                                /* Коды клавиш со стрелками считываются       */
                                /* как два символа, первый из них '\0'        */

            switch (ch)
            {   case 72: ycorr++; break;                /* Вверх      */
                case 75: xcorr--; break;                /* Влево      */
                case 77: xcorr++; break;                /* Вправо     */
                case 80: ycorr--; break;                /* Вниз       */
                case 71: xcorr = ycorr = 0; break;      /* Home       */
            }
        }
    }
    x = xm + xcorr; y = ym + ycorr;

    if (x < xmin) x = xmin;
    if (x > xmax) x = xmax;
    if (y < ymin) y = ymin;
    if (y > ymax) y = ymax;

    if (x != x0 || y != y0)
    {   cursor(x0, y0);                         /* Стирание          */
        cursor(x, y);                           /* Отображение       */
    }
    *px = x; *py = y;
    *pX = Xpixel(x); *pY = Ypixel(y);
}
```

```
void os_shell(int x, int y)

{   to_text( ); printf ( "Введите команду EXIT для возврата в SDRAW");
    system("");                /* Временный выход в ДОС                */

    initgr( ); refresh(x, y);
}

void rdlines(void)

{   FILE *fp;
    int x1, y1, x2, y2, i, freshstart;
    char str[100];
    signed char ch;

    if (getstring(MENUWIDTH+4, 18, "Имя файла: ", str, 1) == 0)
                                                return;
    fp = fopen(str, "r");

    if (fp == NULL)
    {   while (ermes ( " Нельзя открыть файл; нажмите Esc.") != ESC) ;
                                                return;
    }
    freshstart = (ntable + ntxttable == 0);

    do ch = getc(fp); while (ch != '\n' && ch != EOF);
    setwritemode(COPY_PUT);

    while (fscanf(fp, "LT;PU;PA%d,%d;PD;PA%d,%d;",
                                    &x1, &y1, &x2, &y2) == 4)
    {   drawnewline(x1, y1, x2, y2);
        while (getc(fp) != '\n');
                /* Остальная часть вводимой строки не воспринимается */
    }
    setwritemode(XOR_PUT);

    while (fscanf(fp, "PU;PA%d,%d;LB", &x1, &y1) == 2)
    {   i = 0;
        while (ch = getc(fp), ch != '\003' && ch != EOF) str[i++] = ch;
        str[i] = '\0';

        do ch = getc(fp); while (ch != '\n');
        addstring(x1, y1, str);
        outtextxy(Xpixel(x1), Ypixel(y1), str);
    }

    fclose(fp);
    if (freshstart) modified = 0;
}

void refresh(int x, int y)

{   int i;
    static first=1;
```

```
    settextjustify(LEFT_TEXT, BOTTOM_TEXT);
                /* Соглашение по HP-GL для сокращения времени        */

    if (first) first=0;
    else clearviewport( );
    rectangle(0, 0, X__max, Y__max);
    line(0, Ymax, X__max, Ymax);

    defaultbotlin( );
    initmenu( );
    setwritemode(COPY_PUT);

    for (i=0; i<ntable; i++)
        drline(table[i].x1, table[i].y1, table[i].x2, table[i].y2);

    for (i=0; i<ntxttable; i++)
        outtextxy(Xpixel(strtable[i].x1), Ypixel(strtable[i].y1),
                strtable[i].s);
    setwritemode(XOR_PUT);

    cursordraw = 1;
    cursor(x, y);
            /* При cursordraw = 1, вычерчивается новый курсор;        */
            /* При cursordraw = 0, старый курсор стирается.           */
}

int round(float x)
{   return (int)(x < 0 ? x - 0.5 : x + 0.5);
}

void secondmenu(void)

{   int Y=BOXH/2+9;
    getimage(0, 0, MENUWIDTH+8, Ymax-1, menuptr);
    boxes(4, 0);

    outtextxy(12, Y, "Удаление");
    outtextxy(12, Y+=BOXH, "Перенос");
    outtextxy(12, Y+=BOXH, "Копия");
    outtextxy(12, Y+=BOXH, "Выход");
    displaybotlin ( " Укажите нужную операцию.");
}

void selectlinoptions

    (int i, int *px, int *py, int xx1, int yy1, int xx2, int yy2)
{   int j, l, x=*px, y=*py, x0, y0, dragmode, xxm, yym, buttons,
        *pivotx, *pivoty, dx, dy, first=1, X, Y, XXm, YYm,

    XX1=Xpixel(xx1), YY1=Ypixel(yy1),
    XX2=Xpixel(xx2), YY2=Ypixel(yy2);

    cursor(xx1, yy1); cursor(xx2, yy2);
                    /*  Отображение курсора в концевых точках      */
```

```
xxm = (xx1+xx2)/2; yym = (yy1 + yy2)/2;
XXm = Xpixel(xxm); YYm = Ypixel(yym);

cursor(xxm, yym);          /* Вывод курсора в средней точке        */
cursor(x, y);              /* Стирание курсора                     */

secondmenu( );             /* Отображение вторичного меню          */
cursor(x, y);              /* Восстановление курсора               */

do
{   newposition(&x, &y, &X, &Y, &buttons);
} while (buttons == 0 || X >= MENUWIDTH || Y >= 5+4*BOXH);

j = (Y-5)/BOXH;
invertbox(j, x, y, 0);

switch(j)
{   case 0:                                  /* Удаление */
    {   ntable--;
        for (I=i; I<ntable; I++) table[I] = table[I+1];
        drline(xx1, yy1, xx2, yy2);          /* Стирание */
        break;
    }

    case 1: case 2:        /* Перенос или копия соответственно        */
    {   displaybotlin
(         "Совместите со средней или конечной точкой и нажмите кнопку");
        dragmode = -1;

        do
        {   newposition(&x, &y, &X, &Y, &buttons);
            if (abs(X-XX1) < 3 && abs(Y-YY1) < 3) dragmode = 1; else
            if (abs(X-XX2) < 3 && abs(Y-YY2) < 3) dragmode = 2; else
            if (abs(X-XXm) < 9 && abs(Y-YYm) < 9) dragmode = 0;
        } while (buttons == 0 || dragmode < 0);

        while (newposition(&x, &y, &X, &Y, &buttons), buttons) ;
                                        /* Теперь buttons == 0 */

        x0 = x; y0 = y;
        displaybotlin
         ("Переносите перемещением курсора, затем нажмите кнопку");

        while (newposition(&x, &y, &X, &Y, &buttons), buttons==0)
        {   if (dragmode == 0)    {pivotx = &xxm; pivoty = &yym;} else
            if (dragmode == 1)    {pivotx = &xx1; pivoty = &yy1;} else

            {   pivotx = &xx2; pivoty = &yy2;}

                if (x != *pivotx || y != *pivoty)
                {   if (j == 1 || !first) drline(xx1, yy1, xx2, yy2);
                                                    /* Стирание    */
                else          /*  1 = Перенос,      2 = Копия      */
                addline(xx1, yy1, xx2, yy2);
```

```
                                    /*    j = 2: Копия, первый раз          */
                first = 0;
                cursor(xx1, yy1); cursor(xx2, yy2); cursor(xxm, yym);
                dx = x - x0; dy = y - y0;

                if (dragmode != 2) {xx1 += dx; yy1 += dy;} /*    0 или 1   */
                if (dragmode != 1) {xx2 += dx; yy2 += dy;}
                x0 = x; y0 = y;

                drline(xx1, yy1, xx2, yy2);
                cursor(xx1, yy1); cursor(xx2, yy2); cursor(xxm, yym);

                table[i].x1 = xx1; table[i].y1 = yy1;
                table[i].x2 = xx2; table[i].y2 = yy2;
            }
        }
    }
    invertbox(j, x, y, 0);
  }

  cursor(xx1, yy1); cursor(xx2, yy2); cursor(xxm, yym);
                /*  Стирание курсоров в средней и концевых точках */

  *px=x; *py=y; defaultbotlin( );
  modified = 1;
}

void selectobject(int *px, int *py)

{   int x=*px, y=*py, buttons, i, xx1, yy1, xx2, yy2,
      dx, dy, X, Y, found=0;
    long deviation;

    displaybotlin
      (" Совместите курсор с выбираемым обьектом и нажмите кнопку.");

    do newposition(&x, &y, &X, &Y, &buttons); while (buttons == 0);
    defaultbotlin( );

    for (i=0; i<ntable && !found; i++)
    {   xx1 = table[i].x1; yy1 = table[i].y1;
        xx2 = table[i].x2; yy2 = table[i].y2;

        if (x < xx1-4 || x > xx2+4  /* xx1 <= xx2 */
            || y < yy1-4 && y < yy2 || y > yy1+4 && y > yy2+4) continue;

        dx = xx2 - xx1; dy = yy2 - yy1;
        deviation = (long)(x-xx1)*dy-(long)(y-yy1)*dx;

        if (labs(deviation) < 10*(dx + abs(dy)))
        {   selectlinoptions(i, &x, &y, xx1, yy1, xx2, yy2);
            found = 1;
        }
    }
```

```
    if (!found)
    {   for (i=0; i<ntxttable && !found; i++)
        {   xx1 = strtable[i].x1; yy1 = strtable[i].y1;
            if (abs(x-xx1)+abs(y-yy1) < 50)
            {   selecttxtoptions(i, &x, &y);
                defaultbotlin( );
                found = 1;
            }
        }
    }

    if (!found)
    {   displaybotlin ( " Объект не найден. Нажмите Esc.");
        while (getch( ) != ESC) ;
        defaultbotlin( );
    }

    *px=x; *py=y;
    cursor(x; y);                   /* Стирание курсора              */
    putimage(0, 0, menuptr, COPY_PUT);
    cursor(x, y);                   /* Восстановление курсора        */
}

void selecttxtoptions(int i, int *px, int *py)

{   int x=*px, y=*py, j, l;
    char *p=strtable[i].s;
    int xx=strtable[i].x1, yy=strtable[i].y1, X, Y, buttons;

    cursor(x, y);                   /*  Стирание курсора            */
    cursor(xx, yy);                 /*  Метка строки                */
    secondmenu( );                  /*  Вторичное меню              */
    cursor(xx, yy);                 /*  Снятие метки                */
    cursor(x, y);                   /*  Восстановление              */

    do
    {   newposition(&x, &y, &X, &Y, &buttons);
    }   while (buttons == 0 || X>MENUWIDTH+8 || Y>=4*BOXH);
    defaultbotlin( );
    j = Y/BOXH; invertbox(j, x, y, 0);

    switch(j)
    {   case 0:                     /*  Удаление                    */
            ntxttable--;
            for (i=i; i<ntxttable; i++) strtable[i] = strtable[i+1];
            refresh(x, y); break;

        case 1: case 2:             /* Перенос или копия соответственно  */
            displaybotlin
                (" Перенесите курсор в нужную точку и нажмите кнопку");
            while (newposition(&x, &y, &X, &Y, &buttons), buttons==0);
                /* Кнопка нажимается когда курсор в точке назначения */
```

```
            while (newposition(&x, &y, &X, &Y, &buttons), buttons);
                                        /* Кнопка отпущена*/

            if (j == 1)
            {   strtable[i].x1 = x; strtable[i].y1 = y;
            } else addstring(x, y, p);
            refresh(x, y);
    }

    *px = x; *py = y;
    modified = 1;
}

void textinput(int *px, int *py, int *pX, int *pY)

{   int x=*px, y=*py, buttons, X, Y, nonempty;
    char str[100];

    displaybotlin
            ("Поместите курсор в точку начала текста и нажмите кнопку");

    do newposition(&x, &y, &X, &Y, &buttons); while (buttons == 0);
    clearrectangle(1, Ymax+2, X__max-1, Y__max-1);

    displaybotlin
            (" Вводите текст, заканчивая нажатием на клавишу 'Enter'");

    cursor(x, y);                       /* Стирание курсора          */
    nonempty = getstring(Xpixel(x), Ypixel(y), "", str, 0);

    cursor(x, y);                       /* Восстановление курсора    */

    if (nonempty) addstring(x, y, str);
    defaultbotlin( );

    *px=x; *py=y;
    *pX=X; *pY=Y;
}

void txtcursor(int X, int Y)

{   line(X, Y-2, X, Y+2);
    line(X-2, Y-4, X-1, Y-3); line(X+2, Y-4, X+1, Y-3);
    line(X-2, Y+4, X-1, Y+3); line(X+2, Y+4, X+1, Y+3);
}

void wrlines(void)

{   FILE *fp;
    char ch;
    int i, len;
    char str[100];

    if (getstring(104, 30, "Имя файла: ", str, 1) == 0) return;
    fp = fopen(str, "r");
```

```
    If (fp != NULL)
    {   fclose(fp);
        do
        {   ch = ermes ( "Стереть существующий файл? (Y/N): ");
            ch = tolower(ch);
        }   while (ch != 'y' && ch != 'n');
        If (ch == 'n') return;
    }
    fp = fopen(str, "w");

    If (fp == NULL)
    {   while (ermes ( " Нельзя открыть файл; нажмите Esc.") != ESC) ;
        return;
    }

    fprintf ( fp, "IN;SC%d,%d,%d,%d;SR1.2.3.0;\n",
                        leftboundary, xmax, ymin, ymax);

        /* Формирование фала HP-GL:                                    */
        /* Инициализация; определение предельных значений координат.*/
    for (i=0; i<ntable; i++) fprintf ( fp, "LT;PU;PA%d,%d;PD;PA%d,%d;\n",
                                table[i].x1, table[i].y1, table[i].x2, table[i].y2);

    for (i=0; i<ntxttable; i++)
    {   len = strlen(strtable[i].s);
        strtable[i].s[len] = '\003'; strtable[i].s[len+1] = '\0';
        fprintf(fp, "PU;PA%d,%d;LB%s;\n",
                strtable[i].x1, strtable[i].y1, strtable[i].s);
        strtable[i].s[len] = '\0';
    }

    fclose(fp); modified = 0;
}

int xhpg(int X)
{   return (int)(X * x_max200 / X__max);
}

Int Xpixel(int x)
{   return (Int)((long)X__max * x / x_max200);
}

int yhpg(int Y)
{   return (int)((Y__max - Y) * y_max200 / Y__max);
}

int Ypixel(int y)
{   return Y__max - (int)((long)Y__max * y / y_max200);
}
```

Литература

Ammeraal, L. (1986). *Programming Principles in Computer Graphics*, Chichester: John Wiley.
[Имеется перевод: Аммерал Л. *Принципы программирования в машинной графике.* М.: Сол Систем, 1992]

Ammeraal, L. (1986). *C for Programmers*, Chichester: John Wiley.

Ammeraal,L. (1987). *Computer Craphics for the IBM PC*, Chichester: John Wiley.
[Имеется перевод: Аммерал Л. *Принципы программирования в машинной графике.* М.: Сол Систем, 1992]

Ammeraal, L. (1987). *Programs and Data Structures in C*, Chichester: John Wiley.

Ammeraal, L. (1988). *Interactive 3D Computer Craphics*,Chichester: John Wiley.
[Имеется перевод: Аммерал Л. *Интерактивная трехмерная машинная графика.* М.: Сол Систем, 1992]

Borland International (1988). *Turbo C User's Guide, Turbo C Reference Guide Version 2.0.*

Dettmann, T.R. (1988). *DOS Programmers Reference*, Carmel, Indiana: Que.

Kreyszing, E. (1962). *Advanced Engineering Matheematics*, New York: John Wiley.

Lauwerier, H.A. (1987). *Fractals — Meetkundige Figuren in Eindeloze Herhaling*, Amsterdam: Aramith.

Mosich, D., Shammas, N., and Flamig, B. (1988). *Advanced Turbo C Programmer's Guide*, New York: John Wiley.

Norton, P. (1985). *Programmer's Guide to the IBM PC*, Washington: Microsoft Press.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Адаптер графический 39
анимация 78

Битовое отображение 92

Вектор 50
вектор прерываний 60
векторная графика 92
выход в ДОС 192

Галтель 54
генератор символов 26
Гильберта кривая 129
графика векторная 92
графический
    адаптер 39
    драйвер 13, 29
    режим 12

Деловая графика 87
детерминант 51
диаграмма столбчатая 87
диалоговый бокс 189
дракона, кривая 134
драйвер 29
    графический 13, 29
дуга окружности 41
    — по трем точкам 49

Записи режим 16
заполнение эллипса 72
заливка 73

Издательство настольное 90
инвертирование пикселов 17

Классический стиль 7
ключевые слова языка Си
    far 12
    static 60, 111
    void 7, 8
коды ASCII 62
консольное прерывание 59
коррекция 167
Коши кривая 155
курсор 17, 189

Локатор 17

Маска 16
меню 182
Минковского фрактал 154
многоугольник правильный 74
модель памяти 12
"мышь" 158

Область вывода 174
обработка прерывания 60
огромная модель памяти 12
окружность 41, 196
отношение сторон 22, 34, 37

Перекрестие 18
пиксел 14, 37
пикселные координаты 40
плоттер 96
плоттерные единицы измерения 190
полигон 75
пользовательские координаты 21, 38
прерывания
    вектор 60
    программа 59
программа
    GRAB 93
    SDRAW 187, 199
программное прерывание 159
проектный файл 11
прототип 8

Размер символов 26
рекурсия 120

Сетка точек 166
скалярное произведение 50
сканирования код 62
случайные числа 123
современный стиль 7
стрелка 197
стрелка на клавишах 20

Текст, вывод 25
текстовый режим 12
текущая позиция 13, 23
толстые линии 80
точка (*X*, *Y*) 14
точки сетки 189

Файл
BGI 11
BGIOBJ 31
GRAPHICS.LIB. 11
GRASPTC 106
фона цвет 15
формат HP-GL 96, 187
фрактал 128, 148
функции
объявление 8
описание 8
angle( ) 53, 109, 115
arc( ) 41, 43
arc_uc( ) 47, 108, 115
bar( ) 87
bar3d( ) 87
boundaries_uc( ) 38, 107, 112
circle( ) 41
circle_uc( ) 41, 108, 115
circle80_pc( ) 77, 109, 117
circle80_uc( ) 78, 109, 118
clearviewport( ) 175
closegraph( ) 12, 13
draw( ) 23, 107, 113
drawarc3( ) 52, 109, 116
drawpoly( ) 66
ellipse( ) 41, 45
endgr( ) 23, 107, 114
farfree( ) 111
farmalloc( ) 111
fatline( ) 82, 110, 119
fatline0( ) 82, 110, 118
fillcircle_uc( ) 109, 117
fillellipse( ) 72
fillet( ) 57, 109, 116
fillpoly( ) 66
floodfill( ) 73
getarc_uc( ) 108, 115
getarccoords( ) 41
getaspecratio( ) 35
getbkcolor( ) 14
getcolor( ) 14
getimage( ) 176
getmaxcolor( ) 14
getmaxx( ) 13
getmaxy( ) 13
getpixel( ) 14
getviewsettings( ) 175
graphfreemem( ) 114
graphgetmem( ) 114
graphresult( ) 33
grtext( ) 101, 108, 114
imagesize( ) 176
initgr( ) 23, 107, 112
initgraph( ) 12, 13
installBreak( ) 61, 112
invertpixel( ) 23, 107, 113
IX( ) 40,107, 113
IY( ) 40,107, 113
line( ) 12
line_uc( ) 108, 115
lineto( ) 12
move( ) 23, 107, 113
moveto( ) 12
newInt9( ) 61, 112
outtext( ) 25
outtextxy( ) 25
PI 47, 108
pieslice( ) 87, 89
plotcoor( ) 100, 112
putimage( ) 176
putpixel( ) 14
registerfarbgidriver( ) 32
registerfarbgifont( ) 32
restore_old_break( ) 63, 112
sector( ) 87, 89
setaspectratio( ) 40
setbkcolor( ) 14
setcolor( ) 14
setfillstyle( ) 67
setlinestyle( ) 79
settextjustify( ) 28
settextstyle( ) 27
setusercharsize( ) 28
setviewport( ) 175
setwritemode( ) 17, 42, 78
sharpjoint( ) 82, 110, 119
system( ) 198
textheight( ) 26
textwight( ) 26
to_text( ) 107, 114
XPIX( ) 41, 108, 115
YPIX( ) 41, 108, 115

Цвет 14

Хидерный файл 8, 24

Шаблон заполнения 67
шрифт 25
штриховой шрифт 37

Экрана фиксация 79, 93
эллипс 45

Язык HP-GL 96

Оглавление

ПРЕДИСЛОВИЕ 5

Глава 1. ПИКСЕЛЫ, ЛИНИИ И ТЕКСТ 7

1.1. Введение 7
1.2. Наша первая графическая программа 9
1.3. Пикселы и цвета 14
1.4. Режим записи с инвертированием 16
1.5. Координаты пользователя 21
1.6. Текст и шрифты в графическом режиме 25
1.7. Компиляция и компоновка графических драйверов 29

Глава 2. ОКРУЖНОСТИ, ДУГИ И ПОЛИГОНЫ 34

2.1. Отношение сторон 34
2.2. Окружности и дуги 41
2.3. Дополнительные сведения о дугах 47
Дуга по трем заданным точкам 49
2.4. Галтель (закругление) 54
2.5. Установка прерывания программы 59
2.6. Заполнение области 66
Заполнение окружностей и эллипсов 72
2.7. Окружности и правильные многоугольники 74
Окружности, аппроксимируемые многоугольниками 74
2.8. Линии любой толщины 79
Очень толстые линии с закругленными концами 80
Очень толстые линии с острыми углами 82
2.9. Деловая графика 87

Глава 3. ФОРМИРОВАНИЕ ГРАФИЧЕСКОГО ПРОДУКТА 90

3.1. Введение 90
3.2. Использование графики в WordPerfect 5.0 92
Графика с битовым отображением
(файлы с расширением .WPG) 93
3.3. Утилита GRAB 93
3.4. Векторная графика на основе HP-GL 96
3.5. Генерация команд языка HP-GL 99
Прямые линии 99
Текст 101
3.6. Модуль GRASPTC 106

Глава 4. РЕКУРСИЯ И ФРАКТАЛЫ 120

4.1. Рекурсия 120
4.2. Графика и случайные числа 123
4.3. Рекурсия и преобразования 125
4.4. Кривые Гильберта 129
4.5. Кривая дракона 134
4.6. Окружности и квадраты 137
Окружности 137
Квадраты 141
4.7. Фракталы 148

Глава 5. ИНТЕРАКТИВНАЯ ГРАФИКА 158

5.1. Графический ввод с помощью "мыши" 158
5.2. Интерактивная демонстрационная программа 164
5.3. Области вывода и изображения 174
Области вывода 174
Изображения 176
5.4. Утолщение линий 180
5.5. Меню 182
5.6. Программа черчения 187
Пользовательские аспекты 188
Аспекты программирования 197

Литература 217

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ 218

Серия

МАШИННАЯ ГРАФИКА НА ЯЗЫКЕ СИ

Л.Аммерал (Нидерланды)

Принципы программирования в машинной графике
Машинная графика на персональных компьютерах
Интерактивная трехмерная машинная графика
Программирование графики на Турбо Си
(Переводы с английского языка)

В четырех книгах в доступной форме изложены основные вопросы машинной графики. Книги могут быть использованы как пособие при обучении программированию на языке Си с наглядными примерами представления результатов в виде графических изображений.

Исходные модули программ из всех четырех книг поставляются на дискетах.
В комплекте — 5 дискет 360 Кбайт.

На отдельной дискете модифицированная программа трехмерной графики для формирования анаглифических стерео изображений.

Форма оплаты любая — по безналичному расчету, почтовым переводом или за наличные.

Заказы на книги и дискеты направляйте по адресу:
103104 Москва К-104, а/я 38-Львов.

Программирование графики на Турбо Си
ISBN 5-85316-004-X (рус.)
ISBN 0-471-92439-3 (англ.)