Министерство образования и науки Российской Федерации
Федеральное агентство по образованию

ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
РОССИЙСКИЙ ГОСУДАРСТВЕННЫЙ ГИДРОМЕТЕОРОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ

А.Д. Шишкин, Е.А. Чернецова

# ПРАКТИКУМ
## по дисциплине
# «КОМПЬЮТЕРНАЯ ГРАФИКА»

*Издание 2-ое*

РГГМУ
Санкт-Петербург
2008

**УДК 681.327:327**

Шишкин А.Д., Чернецова Е.А. Практикум по дисциплине «Компьютерная графика». Изд. 2, переработанное Санкт-Петербург, РГГМУ, 2008. – 72 с.

В лабораторный практикум по дисциплине «Компьютерная графика» включены лабораторные работы по основным разделам этой дисциплины, читаемой на кафедре «Морские информационные технологии». Лабораторные работы ориентированы на приобретение навыков студентами построения геометрических двух – и трехмерных объектов, сложных полигонов, позволяющим создавать реалистические изображения на экране персонального компьютера. Работы ориентированы на использование языка Си, графического пакета Borland C++ и Borland C++ Builder 6.

Содержащиеся в практикуме сведения теории, методические указания и рекомендации по выполнению лабораторных работ позволяют использовать его в качестве дополнительного пособия для закрепления курса лекций.

Практикум предназначен для студентов гидрометеорологического университета и может быть полезным для всех желающих ознакомиться с основами компьютерной графики.



орологический

## ПРЕДИСЛОВИЕ

Практикум содержит семь лабораторных работ и подготовлен в соответствии с программой дисциплины «Основы компьютерной графики». Его цель – закрепить у студентов теоретические знания, полученные в лекционном курсе, путем создания и реализации программ построения двухмерных и более сложных трехмерных объектов. Каждая лабораторная работа включает в себя описание работы, необходимые для ее выполнения теоретические сведения, краткие методические указания и фрагменты программ. Расположение лабораторных работ идет по мере нарастания сложности изображаемых геометрических картинок. На этих примерах студент может найти описание реализаций принципов программирования, которые могут оказаться полезными при решении реальных задач.

По сравнению с обычными распечатками, которые студент делает на печатающем устройстве в других курсах, графический вывод информации очень привлекателен. Особенный интерес к компьютерной графике проявляется тогда, когда графическое изображение получается в результате собственной деятельности. Принцип "сделай сам" применим ко всем видам деятельности, но в машинной графике мы имеем уникальный случай иметь в собственном распоряжении очень точного и исполнительного помощника.

Дело в том, что с персональными компьютерами можно научиться обходиться так, что они окажутся в состоянии выполнять функции, не подвластные иным средствам, решать задачи, к которым не подобраться другими путями. Мы имеем в виду способность персонального компьютера заполнять вполне осязаемое трехмерное пространство зримыми изображениями. Эту его способность принято называть компьютерной или машинной графикой.

Задача лабораторных работ состоит не только в том, чтобы научить студента строить простые и привлекательные картинки. Но и познакомить их со средствами, при помощи которых возможно не только выводить на экран изображение, но и делать так, чтобы это изображение двигалось, и соответствующие формы изменялись плавно, без неприятного глазу дергания на экране.

Компьютер может выполнить любые рисованные картинки, если мы можем проинструктировать компьютер, как их сделать. К сожалению, последнее утверждение является не исключением, а правилом. Большинство компьютерных пользователей не могут заставить программу вычертить ту картинку, которую они хотят иметь, даже, если они затратили массу денег на приобретение сложного программного обеспечения. Необходимо примириться с тем обстоятельством, что пользователь должен иметь навыки программирования на каком – либо языке.

Для сознательного и достаточно эффективного овладения необходимым набором базовых рецептов машинной графики необходимы два условия [1]:

– определенные навыки работы с каким-нибудь языком программирования достаточно высокого уровня и

– знание некоторых элементов аналитической геометрии и линейной алгебры (разумеется, при непременной возможности работать непосредственно за персональным компьютером).

Студенты должны быть знакомы с концепциями, которые не подвергаются постоянным изменениям и остаются существенными в течение длительного времени. Такой фундаментальной концепцией, например, являются алгоритмы машинной графики, знание которых будет значительно глубже, если алгоритмы изучаются не только теоретически, но и практически. Поэтому уметь программировать алгоритмы является обязательным, даже если в будущем работа по программированию не планируется.

Некоторых пояснений требует выбор используемого в лабораторных работах языка программирования. В принципе для написания программ может использоваться любой язык программирования. Лабораторные работы ориентированы на применение языка Си не только по тому, что он достаточно распространен и на нем написано достаточно большое количество программ по машинной графике, но главным образом из-за его высокого качества и компактности. (Одна строка программы на языке Си может соответствовать десяти строкам на языке Бейсик).

Программирование на языке Си требует аккуратности. Логические ошибки на языке Си приводят к синтаксическим ошибкам значительно реже, чем, например, на языке Паскаль. Но логические ошибки в программе на языке Си могут привести к неправильным

результатам или к выводу сообщений о технических ошибках, которые могут оказаться совершенно непонятными. Другими словами, ошибки могут привести к непредсказуемым результатам.

В компиляторе Турбо Си фирмы Borland International имеется широкий набор графических подпрограмм, позволяющих создавать двух и трехмерные графические изображения. Поэтому, прежде, чем приступить к описанию лабораторных работ, далее приводится краткое описание графики Borland C++[2].

Для помощи студентам при выполнении некоторых работ в Приложении приведен ряд отлаженных программ на языке Си, работающих в пакетах Turbo C, в Borland C++ for Windows и C++ Builder 6.

## ВВЕДЕНИЕ В ГРАФИКУ BORLAND C++

На экране IBM-совместимого компьютера может устанавливаться как текстовый режим, так и графический. Управление экраном в графическом режиме производится с помощью набора функций, прототипы которых находятся в заголовочном файле GRAPHICS.H[2]. Там же объявлены константы и макросы. Файл GRAPHICS.H должен быть подключен с помощью директивы #include препроцессора языка Си ко всем модулям, использующим графические подпрограммы.

Так же как и в текстовом режиме, все графические функции оперируют окнами. В терминологии Borland C++ окно называется *viewport*. Отличие графического окна от текстового состоит в том, что левый верхний угол окна имеет координаты (0, 0), а не (1, 1). По умолчанию графическое окно занимает весь экран.

Прежде, чем использовать графические функции, необходимо установить видеоадаптер в графический режим. Для его установки (инициализации) служит функция initgraph(). Ее прототип записывается следующим образом

```
void far initgraph( int far *driver, int far *mode, char far *path);
```

В состав графического пакета входят заголовочный файл GRAPHICS.H, библиотечный файл GRAPHICS.LIB, драйверы графических устройств (*.BGI) и символьные шрифты (*.CHR).

Функция initgraph() считывает в память соответствующий драйвер, устанавливает видеорежим, соответствующий аргументу mode, и определяет маршрут к директории, в которой находится соответствующий драйвер *.BGI.

Если маршрут не указан, то предполагается, что этот файл располагается в текущей директории.

Заголовочный файл определяет макросы, соответствующие драйверам:

| | | |
|---|---|---|
| DETECT | 0 | Автоматическая установка режима наибольшего графического разрешения |
| CGA | 1 | |
| MCGA | 2 | |
| EGA | 3 | |
| EGA64 | 4 | |
| EGAMONO | 5 | |
| IBM8514 | 6 | |
| HERCMONO | 7 | |
| ATT400 | 8 | |
| VGA | 9 | |
| PC3270 | 10 | |
| CURRENT_DRIVER | –1 | |

При использовании initgraph() можно указать или конкретный драйвер, или задать автоматическое определение (детектирование) типа видеоадаптера и выбора соответствующего драйвера уже во время выполнения программы (макрос DETECT). Это позволяет переносить без изменения программы на компьютеры с другими видеоадаптерами. Значение mode должно быть одним из перечисленных в табл. 1.

*Таблица 1*

| Драйвер | Значение | Разрешение | Палитра | Количество страниц |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| CGACO | 0 | 320x200 | 0 | 1 |
| CGAC1 | 1 | 320x200 | 1 | 1 |
| CGAC2 | 2 | 320x200 | 2 | 1 |
| CGAC3 | 3 | 320x200 | 3 | 1 |
| CGAHI | 4 | 640x200 | 1 | |
| MCGAC0 | 0 | 320x200 | 0 | 1 |
| MCGAC1 | 1 | 320x200 | 1 | 1 |
| MCGAC2 | 2 | 320x200 | 2 | 1 |
| MCGAC3 | 3 | 320x200 | 3 | 1 |
| MCGAMED | 4 | 640x320 | 1 | |
| MCGAHI | 5 | 320x350 | 1 | |
| EGALO | 0 | 640x200 | 16 цветов | 4 |
| EGAHI | 1 | 640x350 | 16 цветов | 2 |
| EGA64LO | 0 | 640x200 | 16 цветов | 1 |
| EGA64HI | 1 | 640x350 | 4 цвета | 1 |
| EGAMONOHI | 0 | 640x200 | 1 | 4 |
| HERCMONOHI | 0 | 720x348 | | 2 |
| ATT400C0 | 0 | 320x200 | 0 | 1 |
| ATT400C1 | 1 | 320x200 | 1 | 1 |

*окончание таблицы 1*

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| ATT400C2 | 2 | 320x200 | 2 | 1 |
| ATT400C3 | 3 | 320x200 | 3 | 1 |
| ATT400MED | 4 | 640x400 | | 1 |
| ATT400H1 | 5 | 640x200 | | 1 |
| VGALO | 0 | 640x200 | 16 цветов | 4 |
| VGAMED | 1 | 640x350 | 16 цветов | 2 |
| VGAHI | 2 | 640x480 | 16 цветов | 1 |
| PC3270HI | 0 | 720x350 | | 1 |
| IBM8514LO | 0 | 640x480 | 256 цветов | |
| IBM8514HI | 1 | 1024x768 | 256 цветов | |

Чтобы выйти из графического режима и вернуться в текстовый режим необходимо использовать функции

void far closegraph(void);

и void restorecrtmode(void);

Функция closegraph() используется, если программа дальше будет работать в текстовом режиме. Эта функция освобождает память, используемую графическими функциями, и устанавливает текстовый режим, который был до вызова функции initgraph(). Если программа завершает работу, то можно использовать функцию restorecrtmode(), которая устанавливает видеоадаптер в текстовый режим, который предшествовал первому вызову функции initgraph().

Тип видеоадаптера определяет, какое количество цветов и какие цвета могут быть использованы в графическом режиме. Наибольшая разница существует между адаптерами CGA и EGA. Количество цветов приведено так же в табл.1.

Видеоадаптер CGA имеет четыре цвета в палитре и четыре палитры. Это значит, что на экране одновременно может быть четыре разных цвета. Цвета нумеруются от 0 до 3. Чтобы выбрать палитру, установите режим CGACx, где x-номер палитры. Цвет с номером 0 всегда совпадает с цветом фона. В качестве фона могут использоваться шестнадцать цветов с номерами от 0 до 15. Соответствие цвета и номера показано в табл.2. В режиме CGAHI может быть только два цвета, один из которых черный цвет фона. Для цветов, так же как и в текстовом режиме, определены макросы, приведенные в табл. 2.

*Таблица 2*

| *CGA* | *Номер* | *EGA/VGA* | *Номер* |
|---|---|---|---|
| BLACK | 0 | EGA_BLACK | 0 |
| BLUE | 1 | EGA_BLUE | 1 |
| GREEN | 2 | EGA_GREEN | 2 |
| CYAN | 3 | EGA_CYAN | 3 |
| RED | 4 | EGA_RED | 4 |
| MAGENTA | 5 | EGA_MAGENTA | 5 |
| BROWN | 6 | EGA_BROWN | 7 |
| LIGHTGRAY | 7 | EGA_LIGHTGRAY | 20 |
| DARK BLACK | 8 | EGA_DARK BLACK | 56 |
| LIGHTBLUE | 9 | EGA_LIGHTBLUE | 57 |
| LIGHTGREEN | 10 | EGA_LIGHTGREEN | 58 |
| LIGHTCYAN | 11 | EGA_LIGHTCYAN | 59 |
| LIGHTRED | 12 | EGA_LIGHTRED | 60 |
| LIGHTMAGENTA | 13 | EGA_LIGHTMAGENTA | 61 |
| YELLOW | 14 | EGA_YELLOW | 62 |
| WHITE | 15 | EGA_WHITE | 63 |

В режиме EGA одновременно могут использоваться 16 цветов из 64 цветов, причем каждый из элементов палитры может быть задан пользователем. Палитра CGA по умолчанию соответствует цветам CGA, однако в файле GRAPHICS.H определены константы, которые содержат соответствующие цветам аппаратные значения.

Что касается цветов, то драйвер VGA работает так же, как и драйвер EGA, но только имеет более высокое разрешение.

Установка цвета фона производится функцией

```
void far setbkcolor(int color);
```

а изменение палитры – функцией

```
void far setpalette(int index, int color);
```

причем эта функция неприменима для видеоадаптера CGA за исключением цвета фона, который всегда имеет index , равный нулю.

Напомним, что графический экран представляет собой массив пикселей.

Каждый пиксель соответствует одной точке на экране и может иметь свой цвет. Установить цвет пикселя в точке экрана с координатами (x, y)можно с помощью функции

```
void far putpixel(int x, int y, int color);
```

③ Основными «рисующими» функциями являются line() и circle(). Их прототипы –

void far line(int x, int y, int x1, int y1);

void far circle(int x, int y, int radius);

Функция line() чертит на экране прямую линию от точки с координатами (x, y) до точки с координатами (x1, y1) текущим цветом.

Функция circle() рисует на экране окружность с центром в точке с координатами (x, y) и радиусом radius (единица измерения – пиксель) так же текущим цветом. По умолчанию текущий цвет устанавливается WHITE. (Обратите внимание, что все макросы пишутся прописными буквами). Изменить текущий цвет, т.е. цвет, которым рисуются линии, можно обратившись к функции setcolor() с прототипом

void far setcolor(int color);

К другим "рисующим "функциям относятся:

arc() – рисует дугу окружности;
drawpoly() – рисует контур многоугольника;
ellipse() – рисует эллипс;
linerel() – рисует линию из текущей точки в точку, задаваемую относительным расстоянием;
lineto() – рисует линию из текущей точки в точку с координатами (x, y);
moveto() – перемещает текущую точку в точку с координатами (x, y);
rectangle() – рисует прямоугольник;
setaspectratio() – изменяет коэффициент сжатия, установленный по умолчанию;
setlinestyle() – устанавливает ширину и стиль линии.

В качестве примера описания прототипа приведем прототип функции rectangle()

void far rectangle( int left, int top, int right, int bottom);

Для закрашивания (заполнения) замкнутого контура служит функция floodfill(), которая закрашивает область заданным цветом по заданному шаблону. Ее прототип –

void far floodfill(int x, int y, int bordecolor);

где x и y –координаты точки внутри контура, bordecolor –цвет контура. Цвет и шаблон заполнения устанавливается функцией

voi far setfillstyle (int pattern, int color);

Вид шаблона закрашивания и соответствующие ему макрос и значение (pattern) приведены в табл.3

*Таблица 3*

| Макрос | Значение | Вид шаблона |
|---|---|---|
| EMPTY_FILL | 0 | Заполнение цветом фона |
| SOLD_FILL | 1 | Сплошное заданным цветом |
| LINE_FILL | 2 | Линиями |
| LSTLASH_FILL | 3 | Косыми линиями /// |
| SLASH_FILL | 4 | Яркими косыми линиями // |
| BKSLASH_FILL | 5 | Обратными косыми линиями \\ |
| LTBKSLASH_FILL | 6 | Яркими обратными косыми линиями \\ |
| HATCH_FILL | 7 | Светлая штриховка сеткой |
| XHATCH_FILL | 8 | Крестообразная штриховка |
| INTERLEAVE_FILL | 9 | Перекрестная штриховка |
| WIDE_DOT_FILL | 10 | Заполнение редкими точками |
| CLOSE_DOT_FILL | 11 | Заполнение частыми точками |
| USER_FILL | 12 | Шаблон заполнения, определяемый пользователем |

Есть так же набор функций, которые чертят контур и закрашивают область внутри контура:

bar() – заполненный прямоугольник;
bar3d() – заполненный столбик;
fillelipse() – заполненный эллипс;
fillpoly() – заполненный многоугольник;
pieslice() – заполненный сектор круга;
sector() – заполненный эллиптический сектор.

Прототипы этих функций их применение и особенности использования можно посмотреть с помощью HELP-системы оболочки Borland C++.

Перечислим еще некоторые функции графической библиотеки системы Borland C++.

setfillpattern() – задает шаблон , определяемый пользователем;
getfillpattern() – возвращает тип шаблона заполнения;

getfillsetting() – возвращает информацию о шаблоне и цвете заполнения;
getlinesetting() – возвращает информацию о текущем стиле, толщине и цвете линии;
getpixel() – сообщает о цвете пикселя в точке (x, y).

В начале мы упомянули о графических окнах, но предположили, что окно совпадает со всем экраном. Создать меньшее графическое окно можно используя функцию setviewport(). Ее прототип –

void far setviewport( int left, int top, int right, int bottom, int flag);

параметры left, top, right, bottom задают местоположение и размеры окна в абсолютных координатах, т.е. координатах экрана. Параметр flag устанавливает режим выхода за границу окна. Если flag не нулевой, то происходит автоматическое прерывание выдачи при выходе за границу окна. В противном случае может происходить выход за границу окна. И в том и другом случае ошибка не фиксируется.

Для работы с экраном, окнами и образами (image) служат следующие функции:

cleardevice() – очищает активную страницу;
setactivpage() – устанавливает номер активной страницы;
setvisualpage() – устанавливает номер видимой страницы;
clearviewport() – очищает активное окно;
getviewsetting() – возвращает информацию об активном окне;
getimage() – записывает образ в заданный участок памяти;
imagesize() – определяет в байтах размер памяти, требуемый для хранения информации о прямоугольной области экрана;
putimage() – помещает на экране ранее записанный в памяти образ.

Функция cleardevice() очищает весь экран, устанавливает текущей точкой левый верхний угол экрана, но оставляет неизменными все установки графического экрана, стиль линии, текущий цвет и т.д.

Функция clearviewport() очищает текущее окно и устанавливает текущую точку в левый верхний угол окна.

В зависимости от типа видеоадаптера и установленного видеорежима система может иметь от одной до четырех буферных страниц. Количество страниц было указано ранее в табл. 1. Каждая из

страниц может быть указана как активная, в которую происходит вывод, и визуальная (видимая), которая отображается на экране. Если они совпадают, то каждый вывод тут же будет отображаться на экране. По умолчанию активная и видимая страницы совпадают.

Для создания движения образа по заданному шаблону на экране служат функции getimage() imagesize() и putimage(): с помощью функции getimage() взять часть экранного образа, вызвать imagesize() для определения размера памяти, необходимой для хранения этого образа, а затем вернуть его на экран в любую желаемую позицию с помощью функции putimage().

Наконец, для осуществления вывода текста в графическом режиме на экран используются функции:

| | |
|---|---|
| outtext() | – выводит строку на экран с текущей позиции; |
| outtextxy() | – выводит строку на экран с заданной позиции; |
| settextjustify() | – устанавливает режим выравнивания текста; |
| setusercharsize() | – устанавливает шрифт, стиль и коэффициент увеличения текста; |
| textheight() | – возвращает высоту строки в пикселах; |
| textwith() | – возвращает ширину строки в пикселах. |

Прототипы этих функций выглядят следующим образом:

```
void far outtext (char far*textstring);

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

Обычно (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 можем изменить их очертание. В графическом режиме символы формируются просто из набора точек. В этом случае любой желаемый шрифт может быть реализован в принципе с помощью программных средств. В графическом пакете языка Турбо Си существует несколько разных шрифтов. Если размер шрифта не задан пользователем, то по умолчанию каждый символ располагается в прямоугольнике 8x8 пикселов. При изображении текста в графическом режиме нет таких средств, как автоматический переход к началу следующей строки при заполнении предыдущей стро-

ки, поэтому нам всегда потребуется проверять, достаточно ли места для отображения строки в пределах границ окна. При выполнении этой операции необходимо быть предельно внимательным, чтобы не перепутать количество символов и количество пикселов. Здесь нам могут оказать помощь две функции Турбо Си, 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. 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 для второго аргументов в этом обращении будут вычислены значения:

719 + 1 – 3 x 8 = 696 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. Первый аргумент *font* может принимать значения из целочисленных значений 0, 1, 2, 3, 4 со следующим смыслом:

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

Если в программу не включается обращение к функции settextstyle, то выбирается шрифт по умолчанию, (т.е. font =

*DEFAULT_FONT)*. Символы при этом шрифте формируются из отдельных точек. При других типах шрифтов символы строятся из *штрихов*, т.е. из 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*. Если задано значение *VERT_DIR*, то текстовая строка вычерчивается в вертикальном положении, как бы повернутой против часовой стрелки на 90 градусов.

Третий аргумент *charsize* определяет размер символов. Задавая значения i = 1, 2, .... 10, можно управлять размером символов. При *charsize = i* символы, состоящие из точек, будут занимать на экране прямоугольник размером 8ix8i пикселов. Если используются символы, изображаемые штрихами, 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 определяется символическая константа *USER_CHAR_SIZE*=0.

Теперь можно применить функцию *setusercharsize* , объявленную как

void far setusercharsize(int multx, int divx, intmulty, int divy);

Параметры *multx, divx, multy, divy* используются для масштабирования ширины и высоты символов: 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 раза с половиной 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 верхний левый 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 |

Приведем пример использования шрифтов в Турбо Си. Строка Turbo C отображается вертикально с типом шрифта "триплекс".

```
    /* FONTDEMO */
/*Демонстрация использования шрифта в графическом режиме*/
#include <graphics.h>
main()
{ int dY, XC;
 initgr();
 settextjustify(CENTER_TEXT, CENTER_TEXT);
 settextstyle(TRIPLEX_FONT, VERT_DIR, 3);
 outtextxy(XC, dY, "Turbo C");
endgr();
}
```

Во время работы программы могут возникать ошибки при выполнении графических операций. Для обработки ошибок, чтобы избежать аварийного прекращения работы программы, служат функции:

graphresult() – возвращает код ошибки выполнения последней графической операции;

grapherrormsg() – возвращает строку с сообщением об ошибке по заданному коду ошибки.

Если произошла ошибка при вызове графической библиотечной функции, устанавливается внутренний код ошибки. Функция graphresult() возвратит код ошибки, а вызов функции grapherrormsg(graphresult()) выдаст сообщение об ошибке. В следующем фрагменте программы приводится рекомендуемый способ инициализации графического режима (с комментариями)

```
/*Запрос автоопределения максимально возможного режима работы видеоадаптера*/
    int driver=DETECT, gmode, errorcode;
    /*инициализация графики */
    initgraph(& graphdriver, &gmode, "");
    /*получение результата инициализации*/
    errorcode= graphresult();
    if(errorcode !=grOk)  /*если произошла ошибка*/
    {
    printf("Ошибка:% s\n", grapherrormsg(errorcode));
    printf( "Для останова нажмите любую клавишу \n");
    getch();
    exit(1);  /* Завершение работы программы*/
     }}
```

В табл. 4 приведены три сообщения об ошибках, их коды и макросы

*Таблица 4*

| *Код* | *Макрос* | *Сообщение об ошибке* |
|---|---|---|
| 0 | grOK | No error |
| 1 | grNoInitGraph | BGI graphics not installed (use initgraph use ) |
| 5 | grNoLoadMem | Not enoph memory to load driver |

## Особенности работы в пакете C++ Builder 6

Графический редактор C++ Builder позволяет создавать картинки из графических примитивов или иллюстрации. В отличие от среды Borland C++ for Windows, которая требует хорошего знания языков программирования, C++ Builder 6 ориентирован на, так называемую, «быструю разработку». В ее основе лежит технология визуального проектирования и событийного программирования, суть которой заключается в том, что среда разработки берет на себя большую часть работы по генерации кода программы, оставляя программисту работу по конструированию диалоговых окон и написанию функций обработки событий.

Среда C++ Builder может вывести графический объект на поверхность формы (Form 1) или компонента (Image). Работа над новым проектом начинается с создания стартовой формы – главного окна программы, которой соответствует свойство Canvas-холст для рисования. Для того, чтобы на поверхности формы или компонента

появился графический примитив необходимо к свойству Canvas применить соответствующий метод.

Например, вывод на поверхность формы прямоугольника с координатами (10, 10, 50, 50) организуется записью

Form1 –> Canvas – > Rectangle (10, 10, 50, 50);

Холст для рисования Canvas состоит из отдельных точек – пикселов. Координаты пикселов нумеруются сверху – вниз и слева – направо. Левый верхний пиксел формы (клиентской области) имеет координаты (0, 0), правый нижний – (ClientWidth, ClientHeight). Доступ к отдельному пикселу осуществляется через свойство Pixels [x][y], где x, y –координаты пиксела. Используя свойство Pixels, можно задать цвет любой точки графической поверхности. Например, инструкция

Canvas –> Pixels[10][10] = clRed;

окрашивает точку поверхности в красный цвет.

Размер графической формы Canvas определяется точками: Pixels [0][0] – левый верхний угол и Pixels [ClientWidth –1][ ClientHeight –1 – нижний правый угол. Вид поверхности формы 1 представлен на рис. В.1.

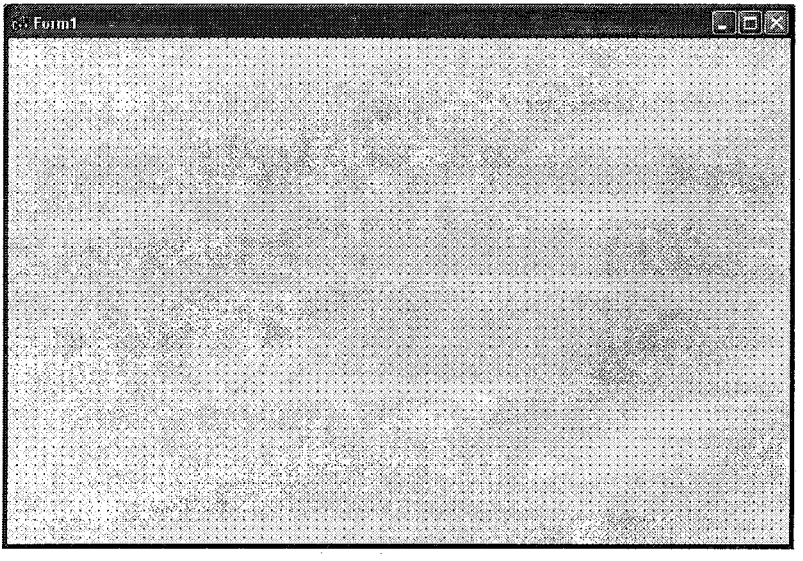


Рис. В.1. Основная форма

Вид графического объекта определяются двумя методами рисования:

- Pen – карандаш;
- Brush – кисть.

Карандаш и кисть являются свойствами объекта Canvas. Свойства объекта Pen задают цвет, толщину и тип линии или границы геометрической фигуры. Свойства объекта Brush задают цвет и способ закраски области внутри прямоугольника, круга, сектора, замкнутого контура.

Любая картинка или схема могут рассматриваться как совокупность графических примитивов: точек, линий, окружностей, дуг и т.д., следовательно, программа должна обеспечивать вычерчивание этих графических примитивов. Вычерчивание примитивов осуществляется *методами*. Например, для вычерчивания линии от точки курсора до координат (x, y) применяется метод LineTo(x, y). А для перемещения карандаша без вычерчивания линии метод MoveTo(x, y). При этом запись будет выглядеть следующим образом

Canvas –> MoveTo(10, 10); // установить карандаш в точку (10, 10)

Canvas –> LineTo(50, 60); // рисовать линию из точки (10, 10) в точку (50, 60).

Используя свойство текущей точки, можно нарисовать любой объект. Например, операторы

```
Canvas –> MoveTo(10, 10);
Canvas –> LineTo(50, 50);
Canvas –> LineTo(10, 50);
Canvas –> LineTo(10, 10);
```

рисуют треугольник.

Для вычерчивания ломаной линии используется метод Poliline. Координаты точек перегиба задаются массивом точек типа TPoint.

***Пример 1.*** **Фрагмент кода рисует ломаную линию, состоящую из трех звеньев.**

```
TPoint p[4]; // Координаты начала, конца и точек перегиба
p[0].x =100; p[0].y = 100; // начало
p[1].x =100; p[1].y = 150; // точка перегиба
p[2].x =150; p[2].y = 150; // точка перегиба
```

```
p[3].x =150; p[3].y = 100; // конец
Canvas –> Poliline (p, 3); // ломаная из трех звеньев.
```

Очевидно, что, если координаты начальной и конечной точек совпадают, то получим замкнутый контур.

Метод Rectangle вычерчивает прямоугольник. Параметрами метода являются координаты двух углов прямоугольника. Например, оператор

```
Canvas –> Rectangle (10, 10, 50, 50);
```

рисует квадрат, левый угол которого находится в точке (10, 10), а правый нижний в точке (50, 50). Цвет, вид и ширину линии контура прямоугольника определяют свойства Pen, а цвет и стиль заливки области внутри прямоугольника-значения свойства Brush той поверхности, на которой метод рисует прямоугольник. Например, следующие операторы рисуют флаг Российской Федерации.

```
Canvas –> Brush->Color=clWhite;
Canvas -> Rectangle(10, 10, 50, 50);
Canvas -> Brush->Color=clBlue;
Canvas -> Rectangle(10, 10, 50, 50);
Canvas -> Brush->Color=clRed;
Canvas -> Rectangle(10, 10, 50, 50);
```

Вместо четырех параметров в метод Rectangle можно передавать один параметр- структуру типа TRect, поля которой определяют положение диагональных углов прямоугольной плоскости.

***Пример 2.*** **Фрагмент программы вычерчивания прямоугольника со структурой**.

```
TRect rct; //прямоугольная область
rct.Top=10;
rct. Left =10;
rct.Bottom =50;
ret. Right =50;
Canvas -> Rectangle(rct);
```

Закрашенные прямоугольники рисуются методом FillRect, где в качестве инструмента используется только кисть. В качестве параметра метода используется структура типа TRect. Поля структуры можно так же задать при помощи функции Rect.

***Пример 3.*** **Фрагмент программы вычерчивания прямоугольника со структурой**

```
TRect ret; //прямоугольная область, которую надо закрасить
rct = Rect (10, 10, 50, 50);
Canvas -> Brush -> Color = clBlue;
Canvas -> FillRect(rct);
```

Для вычерчивания прямоугольника со скругленными углами используется метод RoundRec. Вызов метода выглядит следующим образом:

```
Canvas -> RoundRec (x1, y1, x2, y2, x3, y3);
```

Параметры x1, y1, x2, y2 определяют положение углов прямоугольника, а параметры x3, y3 – размер эллипса для вычерчивания скругленного угла.

Для вычерчивания многоугольника используется метод Poligon. Инструкция вызова метода выглядит так:

```
Canvas -> Poligon(p, n);
```

где p-массив записей типа TPoint[n], который содержит координаты вершин многоугольника, n- количество вершин.

Вид границы многоугольника определяют значений свойства Pen, а вид заливки внутренней области – значения свойства Brush той поверхности, на которой метод рисует.

***Пример 4.*** **Фрагмент программы вычерчивания ромба**

```
TPoint p[4];
// координаты вершин ромба
p[0].x = 50; p[0].y = 100;
p[1].x = 150; p[1].y = 75;
p[2].x = 250; p[2].y = 100;
p[3].x = 150; p[3].y = 125;
Canvas -> Brush->Color=clRed;
Canvas -> Polygon(p, 3);
```

Окружности и эллипсы вычерчиваются одним методом Ellipse. Инструкция вызова метода в общем виде записывается следующим образом:

```
Canvas -> Ellipse(x1, y1, x2, y2);
```

Параметры x1, y1, x2, y2 определяют координаты прямоугольника, внутри которого вычерчивается эллипс или окружность (если прямоугольник является квадратом).

Параметры в метод Ellipse можно передать так же по структуре Trect, аналогичной структуре, приведенной в примере 3.

Метод Arc рисует дугу, являющуюся частью эллипса или окружности. Запись метода следующая:

Canvas-> Arc(x1, y1, x2, y2, x3, y3, x4, y4);

Параметры x1, y1, x2, y2 определяют эллипс (окружность), частью которого является дуга. Параметры x3 и y3 задают начальную, а x4 и y4 – конечную точку дуги. Вычерчивание дуги производится против часовой стрелки. Для создания сектора эллипса или круга используется метод Pie. Инструкция вызова метода аналогична Arc

Canvas-> Pie(x1, y1, x2, y2, x3, y3, x4, y4);

за исключением того, что прорисовывается часть эллипса или окружности, кроме дуги, заключенной между точками x3, y3, x4, y4.

На поверхность графического объекта может быть выведен текст в виде строки типа AnsiString. Это обеспечивается методом TextOutA, инструкция записи которого записывается следующим образом:

Canvas -> TextOutA(x, y, текст);

Параметры x, y определяют координаты точки графической поверхности, от которой выполняется вывод текста. Вид шрифта, его размер и цвет определяются значением свойства Tfont, а размеры области вывода текста – методом TextWidth (ширина) и методом TextHeight (высота). Типы шрифтов определяются свойством Name. Их типы аналогичны, приведенным в MSWord.

***Пример* 5. Вывод текста**

```
AnsiString ms = "Borland C++ Builder"; //строка вывода
Canvas-> Font -> Name= "Times New Roman"; //тип шрифта
Canvas-> Font ->Size = 24; //размер (высота) шрифта
Canvas-> Font -> Color = clBlack; //цвет текста
Canvas -> TextOutA(20, 30, ms); //вывод текста в точке с координатами 20, 30.
```

Следующий фрагмент кода демонстрирует возможность вывода строки текста при помощи двух инструкций

```
Canvas -> TextOutA(20, 50, "Borland C++ Builder");
Canvas -> TextOutA(Canvas-> PenPos.x, Canvas-> PenPos.y, " C++ Builder");
```

В последнем случае текст будет выведен с точки остановки курсора (карандаша).

Более подробно с возможностями графического редактора C++ Builder 6 можно ознакомиться в [3].

При выполнении работ пользователь может на выбор использовать любой пакет. Все программы, приведенные в Приложении, написаны на языке Си. Однако, для использования пакета C++ Builder 6 они так же могут использоваться с необходимыми изменениями.

Для вывода графического объекта на печать необходимо кнопкой PrtScr перекинуть его в буфер компьютера и, открыв программу Paint, вставить рисунок. Отредактированный рисунок дожжен быть помещен в отчет вместе с листингом программы.

## *ЛАБОРАТОРНАЯ РАБОТА № 1*

## *Растровая графика линейных объектов*

**Цель работы**: приобретение навыков работы с графическим редактором в среде Турбо СИ.

### Содержание работы

Для начала обсудим и разберем простейшую программу, в которой используется несколько графических функций, имеющихся в Турбо СИ.

В графическом режиме экран дисплея представляется в виде чертежного поля с прямоугольной системой координат X, Y, начало которой расположено в верхнем левом углу. Координаты выражаются целыми числами в диапазоне от 0 до некоторого максимального значения, зависящего от применяемого графического адаптера. Это максимальное значение будем обозначать $X_{max}$ и $Y_{max}$, где X и Y -прописные буквы, принятые для того, чтобы предотвратить путаницу с $x_{max}$ и $y_{max}$, используемые в программах для различных целей. Экранные координаты показаны на рис. 1.

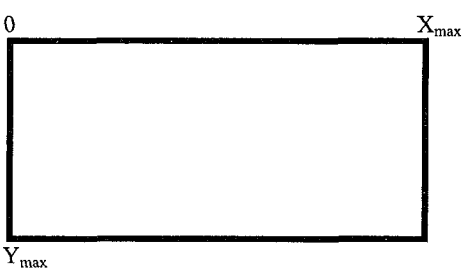


Рис. 1.

Рассмотрим элементы простейшей программы для рисования прямоугольных треугольников.

```
/*TRIA*/
# include <graphics.h>
# include < conio.h>
main ()
{ int gdriver = DETECT, gmode, Xmax , Ymax ;
initgraph(&gdriver, &gmode, "c:\\tc\\bgi");
Xmax = getmaxx(); Ymax = getmaxy();
moveto(0, Ymax); /*Перемещение курсора в нижний левый
угол*/
lineto (Xmax, Ymax); /*Вычерчивание нижнего катета*/
lineto (0, 0); /*Вычерчивание гипотенузы*/
lineto (0, Ymax); /*Вычерчивание катета*/
getch();
closegraph();
}
```

Программа содержит обращение к некоторым графическим функциям Турбо СИ. Заголовочный файл GRAPHICS.H содержит объявление большого числа таких файлов, но мы рассмотрим только некоторые, перечисленные ниже.

```
void far initgraph ( int far * graphdriver,
int far * graphmode,
char far * pathdriver );
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, то при малой модели памяти могли бы появиться конфликтующие форматы указателей.

Первая из перечисленных функций initgraph() переключает компьютер из текстового в графический режим. Она имеет три па-

раметра, которые перечислены ниже вместе с соответствующими аргументами.

| *Параметр* | *Аргумент* |
|---|---|
| graphdriver | &gdriver |
| graphmode | &gmode |
| pathdriver | "\\tc" |

Если для параметра graphdriver будет указана переменная DETECT (значение которой в файле GRAPHICS.H по умолчанию установлено равным 0), то вид адаптера будет устанавливаться автоматически. Кроме того, переменной gmode, указанной в качестве второго аргумента, будет присвоен код наивысшей разрешающей способности. Наконец, в качестве третьего аргумента задается каталог, в котором записан графический адаптер, например, HERC.RGI, в формате текстовой строки.

Функция closegraph() закрывает графический режим работы.

Функции getmaxx() и getmaxy() возвращают наибольшее значение 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, так что их можно будет использовать многократно без повторного обращения к функциям getmaxx () и getmaxy().

Фактическое вычерчивание треугольника выполняется путем обращения к функциям moveto() и lineto(). Координаты здесь определяются целыми числами, обозначающими номера пикселов. Обращение к функции moveto(x, y) означает перемещение курсора из исходной точки в точку с координатами x и y без вычерчивания линии, а функция lineto(x, y) выполняет аналогичное перемещение, но с вычерчиванием линии.

Вместо последовательности обращений moveto(x1, y1); lineto(x2, y2); lineto (x3, y3);
можно записать

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

К функциям, рисующим линии, относится также функция linerel(dx, dy), которая рисует линию из текущей точки в точку, задаваемую относительным расстоянием.

Для рисования линий может использоваться также функция setlinestyle(). Она устанавливает ширину и стиль линии, например:

setlinestyle(SOLID_LINE, 0, NORM_WIDTH) устанавливает стиль сплошной линии, цвета фона, нормальной толщины.

### *Задание на выполнение работы*

1. Внимательно изучить тестовый пример.
2. Войти в интегрированную среду Турбо СИ, активизировать окно File и выбрать подменю New. В появившемся поле редактирования набрать текст приведенной программы.
3. Откомпилировать программу с помощью окна Compile и устранить допущенные ошибки.
4. Запустить программу на выполнение (окно Run или клавиши Ctrl + F9).
5. Модернизировать программу с помощью функций line(), linerel() , setlinestyle(), а также получить три вложенных друг в друга треугольника, размеры которых находятся в пропорции 1/2 и 1/8.
6. Нарисовать стрелку, изображенную на рис. 2

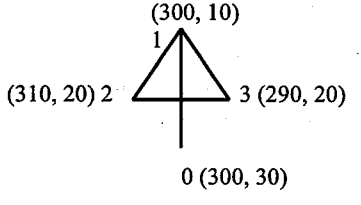


Рис. 2

Используя матрицу поворота координатных осей [3], постройте круг из летящих друг за другом стрелок с началом координат в центре экрана.

### *Содержание отчета*

1. Отпечатанные программы.
2. Вычерченные геометрические объекты.

### *Контрольные вопросы*

1. Назовите основные различия текстового и графического режимов.
2. В каком файле находятся прототипы графических функций ?
3. Что такое графический видеоадаптер? В чем состоит различие видеоадаптеров CGA и EGA?
4. Назовите основные "рисующие" функции.
5. В чем состоит различие между функциями line() и lineto()?

## *ЛАБОРАТОРНАЯ РАБОТА № 2*

## *Растровая графика объектов в среде Borland C++ Builder 6*

**Цель работы**: приобретение навыков работы с графическим редактором в среде C++ Builder 6.

### Содержание работы

Для приобретения основных навыков визуального и событийного проектирования в работе предлагается выполнить ряд примеров, которые позволят на основе базовых элементов в дальнейшем создавать более сложные графические объекты. Выполняя ряд примеров и задач, студент должен освоить технологию и методику создания программ в визуальной среде разработки.

### Инструкция по программированию

Запускается C++Builder обычным образом, т.е. выбором из меню Borland C++ Builder 6 команды **C++ Builder 6**. Вид экрана после запуска несколько необычен, поскольку вместо одного окна появляется сразу пять. При этом окна могут перекрываться и даже закрывать одно другое, как показано на рис. 3.

Запустив C++ Builder, вы увидите заготовку формы будущей программы по центру экрана, за ней Unit1.cpp – окно с листингом (кодом) будущей программы. Сверху экрана расположен список компонентов (объектов), которые можно добавить на форму (Standard, Additional, Win32 и т. п. – это вкладки с различными компонентами). Слева – Object Tree view – дерево (список) компонентов, добавленных в нашу программу. Пока там только Form1 – заготовка формы, под Object Tree View окно Object Inspector (в дальнейшем инспектор) – самое интересное окно, в котором перечислены все свойства выделенного объекта. Щелкните мышкой по Form1 и найдите в инспекторе свойство Caption, щелкните по ней и впишете вместо “Form1” “Калькулятор”, заголовок формы изменится.

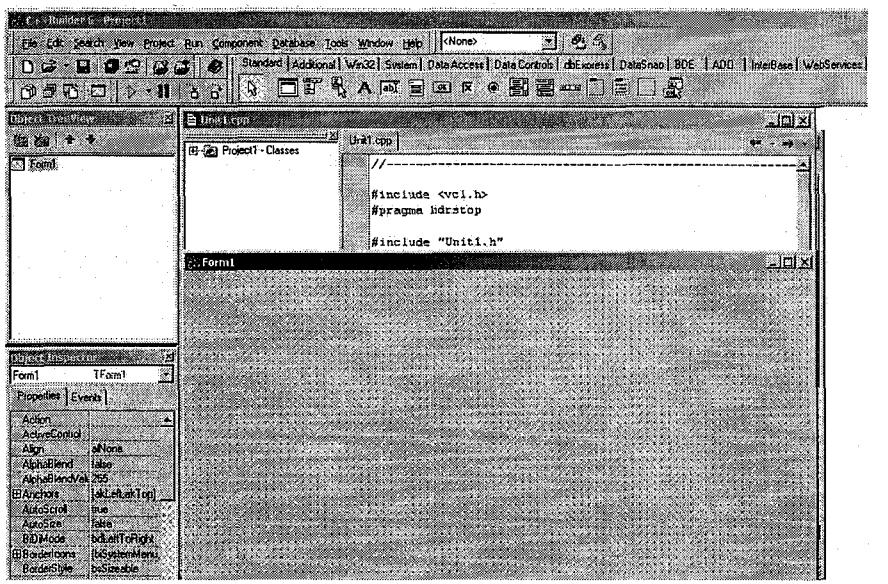


Рис. 3. Основная форма.

Для начала нужно сохранить проект. Щелкните сверху по команде File -> Save project as... (Обязательно перед запуском создайте свою рабочую папку), и сохраните в ней два файла Unit.cpp и Project1.bpr.

Чтобы открыть проект в меню File -> Open project выбрать файл Project1.bpr.

Щелкните мышкой по компоненту Edit (отмечен стрелкой на рис. 4),

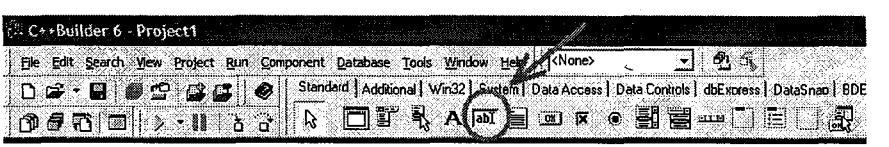


Рис. 4. Рабочее меню.

а затем в любом свободном месте формы. Ее вид изменится и представлен рис.5:

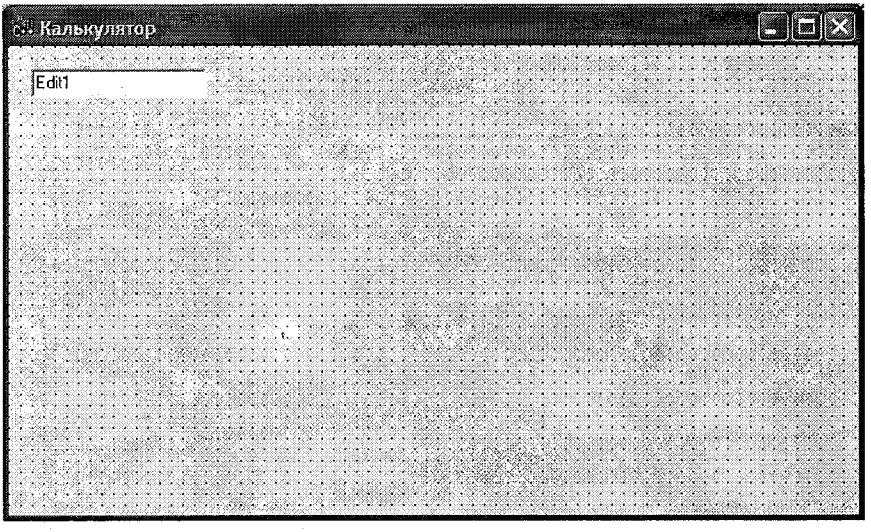


Рис. 5. Форма.

Выделите компонент (объект) Edit1 (рис.6 а) и в Инспекторе измените свойство Text на ноль. В эти окна будут вводиться данные для работы программы.

*а)* *б)*

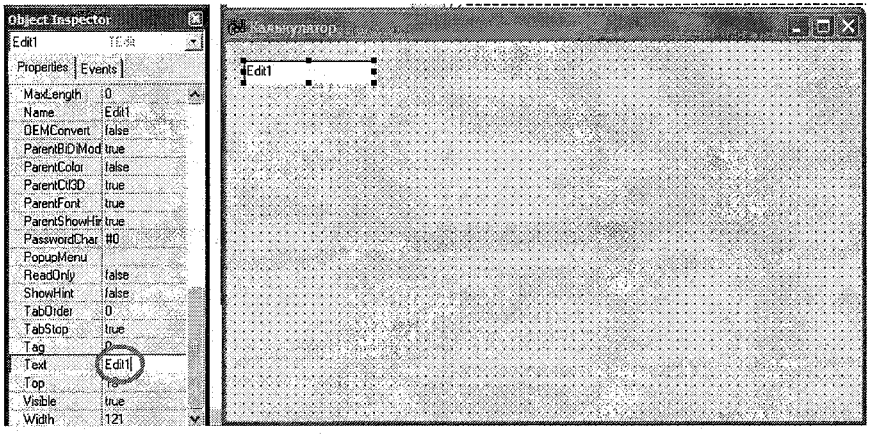


Рис. 6. Окна редактирования.

Затем добавьте еще один компонент Edit, две кнопки (Button) и надпись (Label) командами, выделенными на рис.7.

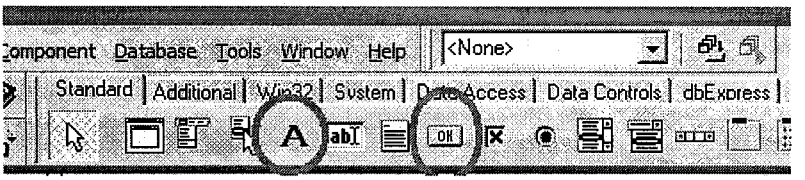


Рис. 7.

Удерживая курсор на соответствующем объекте, расположите их на форме примерно таким образом, как показано на рис. 8.

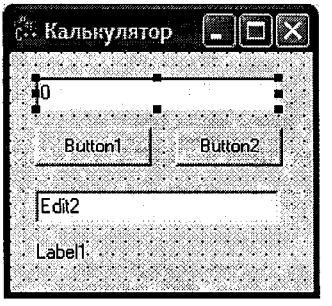


Рис. 8.

Если окно выделено, то размеры формы и компонентов можно изменить мышкой.

Измените свойства кнопок Button/Caption на «/» и «*», свойство Edit2 /Text на ноль, и свойство надписи Label1/Caption на «ответ: 0». Соответствующие формы будут представлены в виде рис. 9.

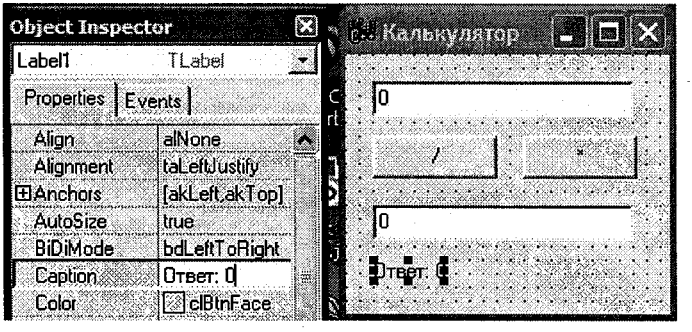


Рис. 9.

Чтобы изменить шрифт текста надписи, надо найти свойство Font в инспекторе и нажать на кнопку с тремя точками, выделенную на рис 10. В открывшемся окне следует выбрать требуемый шрифт.

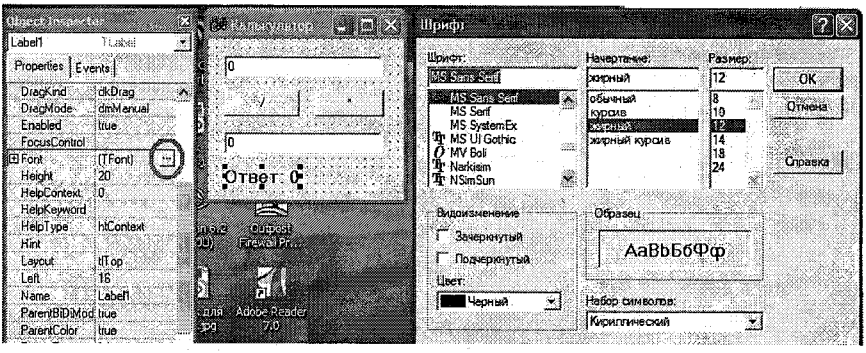


Рис. 10.

Оформив все формы и свойства объектов, переходим к программированию, Работа программы в нашем случае сводится к тому, что при нажатии одной из кнопок программа будет делить или умножать цифру из Edit1 на цифру из Edit2 и выводить результат в Label1. В C++ Builder код выполняется при наступлении события, в нашем случае – нажатие кнопки. Выделите Button1 (делить) и в Инспекторе нажмите вкладку Events, рис.11.

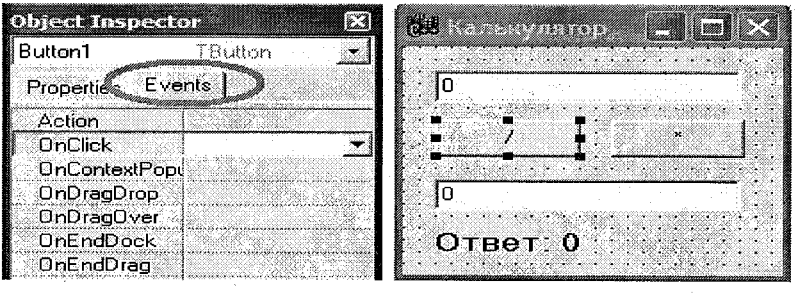


Рис. 11.

Здесь в списке событий, доступных для объекта Button1, нас интересует OnClick. Дважды щелкните в поле рядом с ним, и откроется заготовка процедуры Button1Click, рис.12, в которую надо записать программу расчета (между двумя скобками).

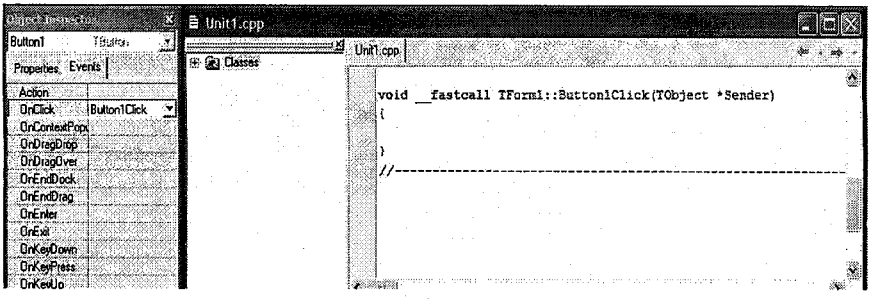


Рис. 12.

При нажатии кнопки делить ее надо привести к такому виду:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
float a, b, c; // переменные для расчетов
a = StrToFloat (Form1->Edit1->Text); // переменной a присваивается цифра из Edit1
b = StrToFloat (Form1->Edit2->Text); // тоже для Edit2
c = a/b; // расчет
Form1->Label1->Caption ="Ответ :"+FloatToStr(c); // вывод ответа
}
```

Так как свойство Text обьекта Edit является строковой переменной типа String, то ее нужно преобразовать в Float, чтобы можно было делить, умножать и другие операции. Преобразование это делается с помощью строковой переменной StrToFloat:

```
a = StrToFloat (Form1->Edit1->Text);
```

Чтобы вывести результат в надпись надо преобразовать переменные обратно:

```
Form1->Label1->Caption ="Ответ :"+FloatToStr(c);
```

Такую же процедуру надо сделать для операции умножения. При написании кода ОБЯЗАТЕЛЬНО соблюдать регистр!

Чтобы запустить программу нажмите **F9**. Результат работы программы будет выглядеть, как это представлено на рис.13.


Рис. 13

Если в процессе работы возникнет ошибка и надо будет остановить программу, нажмите **Ctrl+F2** или **Project->Program Reset**. Чтобы создать exe-файл, используйте команду: Project->Build Project1, и он будет лежать в папке с сохраненной программой. Так как в нашей программе по сути всего четыре строчки, здесь есть несколько недочетов:

1) ноль на ноль делить нельзя, а у нас нет проверки, что за символы вписаны,

2) если вместо цифр вписать текст, то тоже будет ошибка,

3) если число десятичное, то надо ставить только запятую, а не точку.

Полный листинг программы «Калькулятор»:

```
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
//---------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)// Заготовка
{
// Пустая форма
```

```
}
//Программа деления двух чисел
void __fastcall TForm1::Button1Click(TObject *Sender)
{
float a, b, c;
a = StrToFloat(Form1->Edit1->Text);
b = StrToFloat(Form1->Edit2->Text);
c = a/b;
Form1->Label1->Caption ="Результат :" +FloatToStr(c);
}
//Программа умножения двух чисел
void __fastcall TForm1::Button2Click(TObject *Sender)
{
float a, b, c;
a = StrToFloat(Form1->Edit1->Text);
b = StrToFloat(Form1->Edit2->Text);
c = a*b;
Form1->Label1->Caption ="Результат :" +FloatToStr(c);
}
//---------------------------------------------
```

Работа в графическом редакторе производится аналогично. В качестве примера рассмотрим программу рисования полукруга стрелок из работы №1.

```
include <vcl.h>
#pragma hdrstop
 #include<math.h>
#include "Unit1.h"
#pragma package(smart_init)
    #pragma resource "*.dfm"
    TForm1 *Form1;
    //---------------------------------------------------------------------
    __fastcall TForm1::TForm1(TComponent* Owner) :
TForm(Owner)
    {
    float x[4]={300.0, 300.0, 310.0, 290.0}, y[4]={-10.25, 20.25,
10.0, 10.0};
    int i, j;
```

```
float pi, phi, xx, yy, cos_phi,  sin_phi;
pi=4.0*atan(1.0); phi=6*pi/180;
cos_phi=cos(phi); sin_phi=sin(phi);
for(i=1; i<14; i++)
{
for(j=0; j<=3; j++)
{
xx=x[j]; yy=y[j];
x[j]=xx*cos_phi - yy*sin_phi;
y[j]=xx* sin_phi+ yy*cos_phi;
 }
Form1->Canvas->MoveTo(x[0], y[0]);
 for(j=1; j<=3; j++)
 {Canvas->Pcn->Color=clRed;
Form1->Canvas->LineTo(x[j], y[j]);
}
Form1->Canvas->LineTo(x[1], y[1]);
}
}
```

***Задание на выполнение работы***

1. Записать и опробовать программу «Калькулятор».
2. Записать программы для примеров 1-5 из введения, откомпилировать их и опробовать для разных параметров и представления данных.
3. Записать программу «стрелка», произвести отладку программы и запустить в работу.
4. Построить график функции. Задается преподавателем.

***Содержание отчета:***

1. Полные программы для рисования построенных объектов.
2. Результаты построений.

***Контрольные вопросы***

1. Назовите основные “рисующие функции”
2. Перечислите аргументы функций Arc() , Pie().
3. Назовите основные функции вывода текстовой информации в графическом режиме.

## ЛАБОРАТОРНАЯ РАБОТА № 3

## Растровая графика двухмерных графических объектов

**Цель работы**: научиться работать с графическими функциями для рисования дуг, окружностей, эллипсов и других графических примитивов.

### Содержание работы

Для режима рисования необходимо установить на компьютере графический режим работы. Он устанавливается графической функцией initgraph(), полный текст которой следующий:

```
{
void initgraph (void)
int driver, mode, errorcode, Xmax, Ymax ;
register int i;
driver = DETECT;
initgraph (&driver, &mode, " ");
Xmax = getmaxx( ); Ymax = getmaxy();
errorcode = graphresult();
if (errorcode!=grOk) /*если произошла ошибка*/
{ printf ("Ошибка:% s\n", grapherrormsg (errorcode));
printf ("Для останова нажмите любую клавишу \n");
getch();
exit (1); /*Завершение работы*/
}
```

Для облегчения работы (чтобы каждый раз не записывать эти строки) создайте специальный заголовочный файл zaggraf.c, содержащий указанный текст, и поместите его в директорию INCLUDE.

Для входа в графический режим необходимо:

1) В заголовок программы поместить строку: # include<zaggraf.c>

2) Вместо записи intgraph() - указать zaggraf();

Функции Турбо Си позволяют эффективно рисовать окружности, дуги, секторы, эллипсы. 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 xradius, int yradius);
void far getarccoords(srtuct arccoordstype far * arccoords);
```

Структура arccoordstype записана в файле GRAPHICS.H как:

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

Тип графического видеоадаптера определяет количество цветов, и какие цвета могут быть использованы в графическом режиме. Наибольшая разница существует между адаптерами EGA и CGA. Видеоадаптер имеет четыре цвета в палитре и четыре палитры. Это значит, что на экране одновременно могут быть четыре цвета. Цвета нумеруются от 0 до 3. Палитры также нумеруются от 0 до 3. Цвет номер 0 всегда совпадает с цветом фона. Драйвер VGA с цветами работает также как и драйвер CGA, только имеет более высокое разрешение.

Установка цвета фона производится функцией

```
void far setbkcolor(int color);
```

а изменение палитры_функцией

```
void far setpalette( int index, int color);
```

причем эта функция не применима для видеоадаптера CGA за исключением цвета фона, который всегда имеет index, равный нулю.

Функция circle() рисует на экране окружность с центром в точке с координатами (x, y) и радиусом radius (единица измерения-пиксель) текущим цветом. По умолчанию цвет устанавливается WHITE. Изменить цвет линии можно функцией setcolor() с прототипом

```
void far setcolor(int color);
```

К другим рисующим функциям относятся:

arc()- рисует дугу окружности,

drawpoly()-рисует контур многоугольника,
elipse()-рисует эллипс,
rectangle-рисует прямоугольник.
Прототипы рисующих функций приведены в разделе 1.

Для закрашивания (заполнения) замкнутого контура служит функция floodfill(), которая закрашивает область заданным цветом по заданному шаблону. Ее прототип

void far floodfill(int x, int y, bordercolor);

где x, y – координаты точки внутри контура заполнения, bordercolor-цвет контура.

Цвет и шаблон заполнения устанавливается функцией

void far setfillstyle(int pattern, int color);

Вид шаблона закрашивания и соответствующие ему МАКРОС и значение (pattern) приведены в табл.2. Обратите внимание, что все макросы записываются с прописной буквы с подчеркиванием.

Примеры использования рисующих функций.

*Пример 1*

```
/*Рисование квадрата в центре экрана*/
.....
main()
{
--------------------
left=getmaxx()/2-50; /*  задается верхний*/
top=getmaxy()/2-50;  /*левый угол */
right=getmaxx()/2+50; /* задается нижний */
bottom=getmaxy()/2+50; /*правый угол*/
/*Рисование квадрата*/
rectangle(left, top, right, bottom);
getch();
}
```

*Пример 2*

```
/*Программа рисует круг в левом верхнем углу, который перемещается
по диагонали вниз*/
```

```
main()
{
int ym=480,  xm=640, r=50;
dx=((xm-2*r)/150);   dy=((ym-2*r)/150);
x=r;  y=r;
initgraph (void);
setviewport(1, 1, 640, 480, 0); / * задание окна вывода*/
setbkcolor(BLACK); /*задание цвета фона */
setcolor(WHITE); /* задание цвета примитива*/
clearviewport();  /* очистка окна вывода */
setwritemode(1); /* задание типа печати */
circle(r, r, r); /* рисование окружности */
getch();  /* ждать нажатия клавиши */
/* перемещение окружности по диагонали */
while((y<ym-r)&&(x<xm-r)
{
setcolor(WHITE);
circle(x, y, r);
setcolor(BLACK);
circle(x, y, r);
x=dx+x;
y=dy+y;
}
circle(xm-r, ym-r, r);
getch();
return 0;
}
```

***Задание на выполнение работы***

1. Записать полную программу для примера 1 с полной инициализацией графики, откомпилировать ее и опробовать для разных параметров прямоугольника.
2. Записать полную программу для примера 2, используя для инициализации графики модуль <zaggraf.c>, произвести отладку программы и запустить в работу.
3. Составить программу для зарисовки графического объекта, представленного на рис.14.
4. Нарисовать закрашенный эллипс.
5. Нарисовать закрашенный столбик bar3d().

### *Указания для построения*

Координаты прямоугольника (rectangle):10, 10, 639, 349, цвет любой.
Координаты закрашенного прямоугольника (bar):50, 50, 300, 300.
Координаты окна (setvievport()):100, 100, 200, 200.
В окно вписать несколько десятков концентрических окружностей, заполняющих все пространство окна.
Надпись над рисунком сделать с привязкой по центру шрифтом TRIPLEX_FONT.

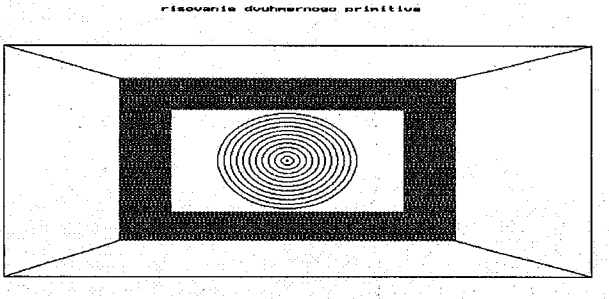

Рис. 14.

### *Содержание отчета:*

1. Полные программы для рисования трех объектов.
2. Результаты построений.

### *Контрольные вопросы*

1. Назовите основные “рисующие функции”
2. Перечислите аргументы функции rectangle().
3. В чем состоит различие функций bar() и bar3d()?
4. Назовите основные функции вывода текстовой информации в графическом режиме.
5. Как производится изображение букв в графическом режиме?

## ЛАБОРАТОРНАЯ РАБОТА № 4

## Применение рекурсии для построения графических объектов

**Цель работы:** Изучить основные методы составления программ для построения рекурсивных изображений простых геометрических объектов: треугольников, окружностей, заданных кривых.

### Содержание работы

Говорят, 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 обращение к такой функции. При входе в обычную функцию выход из нее всегда происходит раньше, чем повторный вход, но для рекурсивной функции это не обязательно.

Примером применения рекурсии является построение вложенных треугольников. Для этой цели будем использовать программу построения треугольников TRIA, описанную в работе №1. Вычертим один треугольник с вершинами A, B, C, координаты которых: xA, yA, xB, yB, xC, yC, но затем снова вызовем функцию TRIA с координатами средних точек его сторон:
xP = (xB+xC)/2, yP = (yB+yC)/2; xQ = (xC+xA)/2; yQ = (yC+yA)/2;
xR = (xA+xB)/2; yR = (yA+yB)/2. Так, что внутри треугольника появятся четыре малых треугольника (рис.15) Затем функция вызовет самою себя с координатами вершин малых треугольников в качестве аргументов.

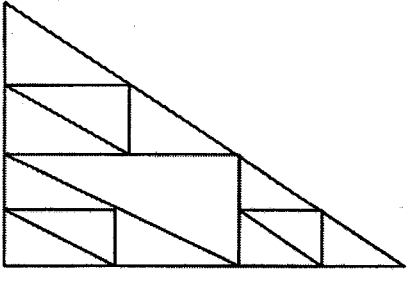

Рис. 15.

Данный процесс может быть проверен *n* раз. Число n называется глубиной рекурсии и включается в качестве аргумента в функцию. Например, если пользователь задает *n* = 7, то аргумент функции устанавливается равным n–1 для каждого рекурсивного обращения, т.е. при

достижении самого «глубокого уровня рекурсии» значение n становится равным нулю, что приводит к немедленному возврату в вызывающую функцию, т.е. в саму функцию TRIA с рекурсией приведена в приложении.

Следующим примером использования рекурсии является получение линейного рисунка, известного под именем *кривой Гильберта*. Эта кривая основана на изображении буквы П, вычерченной в виде трех сторон квадрата как показано на рис. 16. Существуют кривые Гильберта порядков 1, 2, ..., обозначаемые как H1, H2, .... Фигура H2 получается из четырех фигур H1, соединенных связками (толстые линии). В действительности все линии одинаковой толщины. Мы видим, что H2 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.

Применим ту же процедуру к каждой из четырех букв П, составляющих H2, т.е. каждую букву П в H2 заменим меньший H2, одновременно уменьшим длину связок так, чтобы их длины стали равными длине элементарного отрезка прямой линии, которые содержатся в трех малых фигурах H2, т.е. мы получили фигуру H3 (рис. 17).

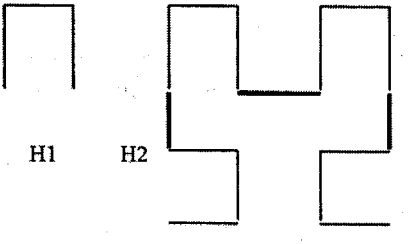

Рис. 16.

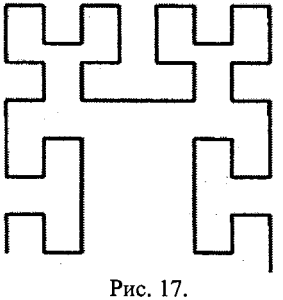
Рис. 17.

Теперь все элементарные отрезки в семь раз меньше, чем длина сторон квадрата, в который вписывается фигура H3. Отсюда получаем, что коэффициенты уменьшения для этих элементарных отрезков в фигурах H1, H2, H3, ..., образуют ряд чисел 1, 3, 7, ...., т.е. в общем случае коэффициенты уменьшения для фигуры Hn может быть вычислен по формуле $2^n - 1$.

Заметим, что связки в фигуре H2 вычерчиваются в тех же направлениях, как и три отрезка, образующих букву П в фигуре H1.

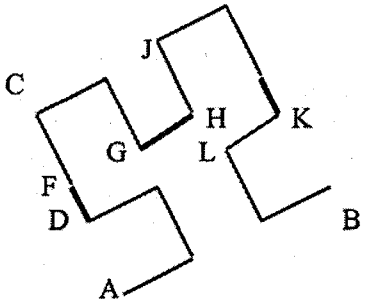

Рис. 18

При желании эти последние отрезки прямых линий можно рассмотреть как связки, соединяющие четыре точки, которые в свою очередь можно принять за кривую нулевого порядка.

Рассмотрим формирование рекурсивной функци Hilbert со следующими аргументами:

– координаты точек A, B, C (см рис.18);

– горизонтальные и вертикальные компоненты двух направленных связок, причем одна лежит на отрезке AB, а другая на отрезке AC.

Они задаются в виде векторов, т.е. как пара чисел (dx, dy), которые могут принимать положительные, отрицательные и нулевые значения, в зависимости от относительного положения точек A, B, C. Эти два вектора обозначим как dAB и dAC, а глубину рекурсии – за $n$. При $n = 0$ функция ничего делать не будет.

Будем считать, что рис. 17 является вариацией изображения буквы П, повернутой на угол $30^0$ против часовой стрелки, позиция которой определяется тремя заданными точками A, B и C. Будем считать , что точка A является начальной, 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. 18 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 17. В общем случае пунктирные линии на рис. 17 не вычерчиваются. Вместо этого будем выполнять рекурсивное обращение к функции Hilbert для каждой из четырех букв П. Для вычерчивания трех отрезков связок DF, GH, IK будем обращаться к функции draw() или lineto().

Функция Hilbert и использующая ее функция sguare() приведена в приложении.

Можно реализовать множество привлекательных рисунков с помощью функций, которые решают две задачи: во-первых, они вычерчивают некоторую фигуру, скажем, окружность с центром и радиусом, заданным в качестве аргументов, во вторых, они рекурсивно обращаются несколько раз сами к себе для вычерчивания той же самой фигуры, но меньшей чем исходная. Меньшие фигуры располагаются на расстоянии больше радиуса первой окружности (как спутники на орбите) и т. д. (рис. 18).

Допустим, что наибольшая окружность имеет радиус $r$. Тогда остальные три окружности будут иметь радиусы $fr, f^2r, f^3r$. Радиус наибольшей орбиты вычисляется 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.

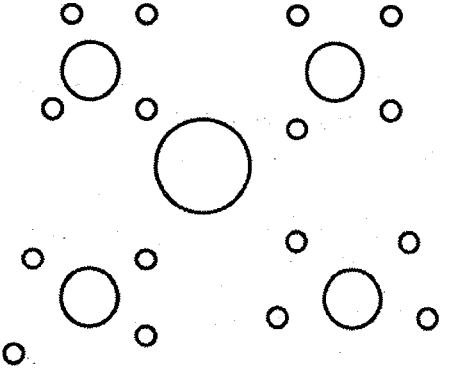

Рис. 18.

Это означает, что расстояние между центрами самой внутренней (наибольшей) окружности и самой внешней (наименьшей) окружности равно:

$$R + fR + f^2R.$$

Поскольку для вычисления значений радиуса $R$ нам необходимо соотнести это выражение с доступным полем на экране, то мы должны учесть размеры и самой маленькой окружности. Она хоть и маленькая, но все-таки имеет определенные размеры, следует добавить к выше приведенному ряду. Самым простейшим способом решения этой проблемы будет добавление следующего члена ряда, что означает отведения места для следующей орбиты, как если бы было на единицу больше. Следовательно, если $n$=4, то будем использовать ряд: $R + fR + f^2R + f^3R$.

В общем случае это расстояние равно:

$$S = R + fR + f^2R + \ldots + f^{n-1}R = R(1 + f + f^2 + \ldots + f^{n-1}) = r(1 - f^n)/(1 - f)$$

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:

$$R = 0.5 * y_{\max} * (1 - f)/(1 - p);$$

$$r = R/C$$

где $p = f^n$.

Описанные выше приемы создания рекурсивных изображений могут быть применены к другим фигурам, например квадратам, звездам и т.д.

***Задание на выполнение работы***

1. Ввести программу построения рекурсивного треугольника, провести ее отладку и запустить в работу для разных *n*.
2. Составить полную программу для построения кривой Hilbert и окружностей (Satellit) и исследовать ее для разных глубин рекурсии *n*.

***Содержание отчета***

1. Алгоритмы построения рекурсивных функций.
2. Программы их реализующие.

***Контрольные вопросы***

1. Какие функции называются рекурсивными?
2. Когда применяются рекурсивные функции?
3. Что такое глубина рекурсии?
4. Какие рисующие функции применяются для построения рекурсий?

## ЛАБОРАТОРНАЯ РАБОТА № 5

## Формирование перспективных изображений

**Цель работы**: Приобретение навыков построения простейших каркасных моделей в трехмерной области

### Содержание работы

В компьютерной графике, наряду с другими моделями, широко используются каркасные или проволочные модели [3]. Особенностью каркасной модели является то, что для ее описания используются геометрические объекты первого порядка: линии или ребра. Они применимы для моделирования объектов, представляющих собой полиэдры, т.е.замкнутые многогранники произвольной формы, ограниченные плоскими гранями или объекты, получаемые перемещением образующей , которая фиксируется в некоторых положениях. Модель 3D в этом случае содержит список координат вершин полиэдра с указанием связей между ними, т.е. ребер (рис. 20).

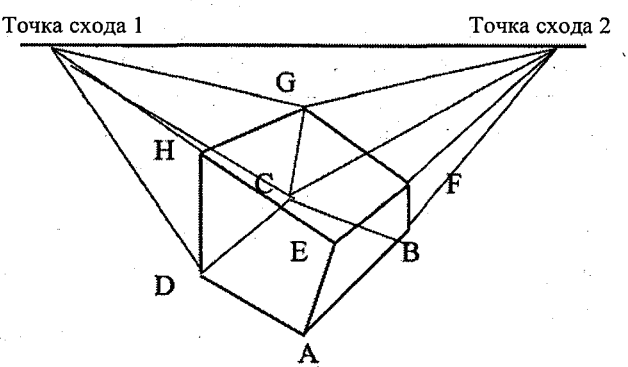


Рис.20

На рис.20 двухмерное изображение куба представлено вместе с некоторыми дополнительными линиями. На этом изображении отрезки AB и AD не параллельны нижнему или верхнему краю листа бумаги, так что можно было бы утверждать, что они горизонтальны.

Однако они обозначают ребра куба в трехмерном пространстве, поэтому в принципе их можно назвать горизонтальными. По этой же причине можно утверждать, что два отрезка AB и CD параллельны. По этой терминологии параллельные горизонтальные линии встречаются в так называемой *точке схода*. Все точки схода лежат на одной линии – линии горизонта. Линия горизонта и точка схода реально не существуют в трехмерном пространстве, но они помогают получить реалистические изображения. Такие изображения называются *перспективными*.

Очевидно, что картинка будет зависеть от положения глаза, так как "эффект перспективы" будет зависеть от расстояния между глазом и объектом. Чем дальше расположен глаз от объекта, тем более параллельными нам будут казаться ребра и тем более реалистичным будет изображен объект.

Объект в двух- и трехмерных пространствах может быть описан координатами точек $(X, Y)$ и $(x, y, z)$ соответственно. При необходимости получения перспективной проекции задается большое количество точек $P(x, y, z)$, принадлежащих объекту, для которых предстоит вычислить координаты точек $P'(X, Y)$ на картинке. Для этого нужно только преобразовать координаты точки $P$ из так называемых мировых координат $(x, y, z)$ в экранные координаты $(X, Y)$ ее центральной проекции $P'$. Будем предполагать, что экран расположен между объектом и глазом (точка $E$ на рис.21). Для каждой точки $P$ объекта прямая линия $PE$ перемещает экран в точку $P'$ .

Это отображение удобно выполнять в два этапа [5]:

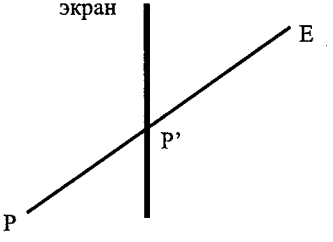


Рис. 21

1) Видовое преобразование – точка $P$ остается на своем месте, но система мировых координат переходит в систему *видовых координат*.

2) Перспективное преобразование – точное преобразование точки $P$ в точку $P'$, объединенное с переходом из системы трехмерных видовых координат в систему двухмерных экранных координат.

Мировые координаты (xw, yw, zw)

|

Видовое преобразование: Видовые координаты(xe, ye, ze)

|

Перспективное преобразование: Экранные координаты(X, Y).

Видовое преобразование показано на рис. 22
Для выполнения видовых преобразований должны быть заданы точка наблюдения (E), совпадающая с глазом, и объект (O). Желательно, чтобы система мировых координат была правой. Объект наблюдается из точки *E* .

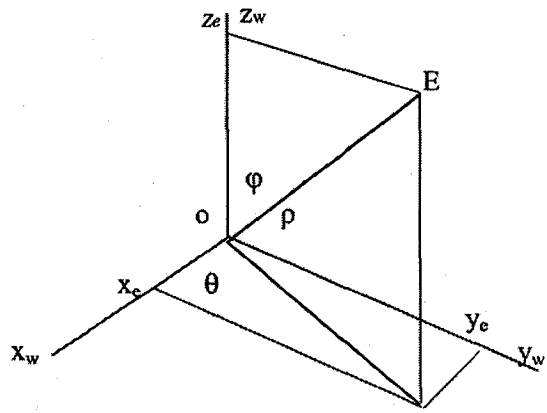


Рис. 22

Точка E задана в сферических координатах, т.е. мировые координаты могут быть вычислены как

$$xe = \rho \sin\varphi\cos\theta; \quad ye = \rho \sin\varphi\sin\theta; \quad z_e = \rho\cos\varphi. \qquad (1)$$

Из точки *E* можно наблюдать точки объекта только внутри некоторого конуса, ось которого совпадает с линией *EO*, а вершина- с точкой *E*.

Если заданы ортогональные координаты *xe, ye , ze* точки *E*, то можно вычислить ее сферические координаты по методике, изложенной в [5].

Конечной задачей является вычисление экранных координат *X, Y*, для которых оси *X* и *Y* лежат в плоскости экрана, расположенной между точками *E* и *O* и перпендикулярной направлению наблюде-

ния *EO*. Начало системы видовых координат располагается в точке наблюдения *E*. При направлении взгляда из *E* в *O* положительная полуось *xe* направлена вправо, а положительная полуось *ye* – вверх. Направление оси *ze* выбирается таким образом, что значения координат увеличиваются по мере удаления от точки наблюдения.

Видовое преобразование может быть записано в форме

$$\mathrm{I}\, xe, ye, ze \,\mathrm{I} = \mathrm{I}\, xw, yw, zw \,\mathrm{I}^* \,\mathrm{I}\, V \,\mathrm{I}, \qquad (2)$$

где *V*-матрица видового преобразования размером 4x4. Матрица *V* получается путем перемножения четырех матриц преобразований [5]:

1) Матрицы T-переноса системы координат из точки O в точку E.

2) Матрицы $R_z$ - поворота системы координат вокруг оси z.

3) Матрицы $R_x$- поворота системы координат вокруг оси x.

4) Матрицы Myz- изменения направления оси x. (Последняя матрица необходима ввиду того, что система мировых координат имеет правостороннюю ориентацию, а система видовых – левостороннюю).

Опуская математические выкладки, запишем матрицу *V*

$$\begin{vmatrix} \sin\theta & -\cos\varphi\cos\theta & -\sin\varphi\cos\theta & 0 \\ \cos\theta & -\cos\varphi\sin\theta & -\sin\varphi\cos\theta & 0 \\ V=0 & \sin\varphi & -\cos\varphi & 0 \\ 0 & 0 & \rho & \mathrm{I} \end{vmatrix} \qquad (3)$$

Итак, если заданы сферические координаты ρ, φ, θ для точки наблюдения, то положение точки в системе видовых координат можно вычислить по значениям ее мировых координат, используя только уравнения (2) и (3).

**Перспективные преобразования**

Следующим шагом будет преобразование видовых координат в экранные. Здесь мировые координаты уже не будут затрагиваться, поэтому их обозначать будем без индексов *(x, y, z)*.

Взаимосвязь видовых и экранных координат показана на рис. 23. На рис. 23 выбрана точка *Q* , видовые координаты которой (0, 0, *d*) для некоторого положительного числа *d*. Плоскость $z = d$

определяет экран – тоже плоскость, проходящую через точку $Q$ и перпендикулярную оси $z$. Экранные координаты определяются точкой привязки начала координат к точке $Q$, а оси $X$ и $Y$ имеют такое же направление, что и видовые координаты.

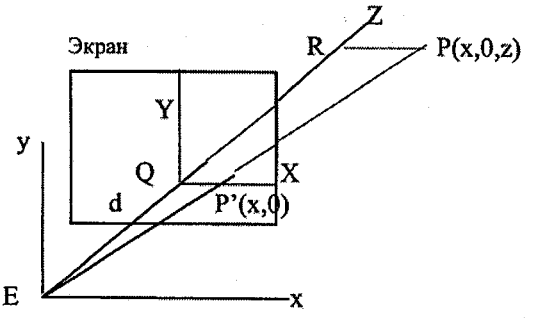


Рис. 23.

Для каждой точки $P$ объекта точка изображения $P'$ определяется как точка пересечения прямой $PE$ и экрана. Чтобы упростить выкладки будем считать, что точка $P$ имеет нулевую у-координату, хотя полученные далее соотношения справедливы для любой у-координаты. Из рис.23 видно, что треугольники $EPR$ и $EP'Q$ подобны. Следовательно,

$$P'Q/EQ = PR/ER.$$

Отсюда имеем:

$$X/d = x/z, \quad X = d*(x/z), \tag{4}$$

$$Y/d = y/z, \quad Y = d*(y/z). \tag{5}$$

Вначале мы предположили, что точка $O$ начала системы мировых координат совпадает с центром объекта (рис.22). Поскольку ось $z$ видовой системы координат совпадает с прямой линией $EQ$, которая пересекает экран в точке $Q$, то начало $Q$ системы экранных координат будет находиться в центре изображения. Если мы хотим, чтобы это начало координат располагалось в нижнем левом углу экрана, а размеры экрана составляли 2*c1 по горизонтали и 2*c2 по вертикали, то уравнения (4) и (5) надо заменить на

$$X = d*(x/z) + c1,$$

$$Y = d*(y/z) + c2.$$

### *Задание на выполнение работы*

1. Создать перспективную модель куба с длиной ребра 2h для произвольных сферических параметров:
   ρ = EQ – расстояние до точки наблюдения;
   θ – угол в горизонтальном направлении от оси x;
   φ – угол, измеренный по вертикали от оси z;
   d – расстояние между экраном и точкой наблюдения.
   (Макет программы приведен в приложении).
2. Исследовать видоизменение модели от указанных параметров и коэффициентов c1 и c2, определяющих размеры экрана.
3. Преобразовать каркасную модель в модель твердого тела путем внесения изменений в программу.

### *Указания для выполнения*

Точка *O* начала системы мировых координат выбирается в центре куба. Длина каждого ребра равна 2*h*.

Тогда координаты вершин куба будут иметь следующие координаты:

*A(h, -h, -h)  B(h, h, -h)  C(-h, h, -h)  D(-h, -h, -h)*
*E(h, -h, h)  F(h, h, h)  G(-h, h, h)  H(-h, -h, h).*

Для перемещения пера в заданную точку используется функция *mv(x, y, z)*, а для вычерчивания ребра- *dw(x, y, z)*, аналогичные *moveto(x, y)* и *lineto(x, y)*.

Функция *perspective()* реализует как видовое, так и перспективное преобразование. Функция *coeff* () реализует вычисления составляющих матрицы *V* и других вспомогательных коэффициентов, не зависящих от координат точек.

### *Содержание отчета*

1. Распечатку программы с нужными комментариями.
2. Виды полученных моделей.

### *Контрольные вопросы*

1. Что из себя представляет проволочная модель?
2. Какой математический аппарат используется для перспективных и проектных преобразований?
3. Напишите матрицу переноса системы координат в трехмерном пространстве.
4. Опишите различие видовых, мировых и экранных координат.
5. Какие изменения необходимо предусмотреть в программе для преобразования проволочной модели в модель твердого тела?

## ЛАБОРАТОРНАЯ РАБОТА № 6

## Построение полигонов сложной формы

**Цель работы**: Практическое использование функций fillpoly() и drawpoly(), а так же преобразования системы координат для построения полигонов сложной формы.

### Содержание работы

Полигоны или многоугольники можно рисовать самыми различными способами, например, с помощью функций line() и lineto(). В Турбо Си имеются две функции для построения полигонов: drawpoly() и fillpoly(). Drawpoly() вычерчивает полигон обычным способом как любую ломаную, заданную совокупностью координат некоторого множества точек. Это может быть как сложная геометрическая фигура, так и график математической функции, заданной в табличном виде.

Fillpoly() отличается тем, что вычерчивает сам полигон и заполняет его внутреннюю область. Объявление fillpoly() в файле GRAPHICS.H выглядит так

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

Параметр numpoints обозначает количество пар координат (X, Y) точек ломаной, которые задаются через второй аргумент polypoints.

Если полигон имеет $n$ вершин (пронумерованных 0, 1, 2, ...$n$–1), то значение переменной numpoints должно быть равно $n$+1 и polipoints является начальным адресом следующей последовательности целых чисел

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

Нам необходимо замкнуть контур многоугольника самостоятельно, записав одну пару координат дважды (точка $X_0$, $Y_0$). В результате как первый аргумент numpoints, так и число пар координат должно быть равно $n$+1.Заметим, что здесь нужно задавать целочисленные пиксельные координаты.

Используем функцию fillpoly() для построения полигона, представленного на рис.24, называемого "магическим" треугольником. Данный полигон получается заполнением областей и граней треугольника, предложенного Эшером [5].

Чтобы организовать "заливку" (т.е. раскраску) полигона, надо использовать функцию, описанную 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 типов шаблонов заполнения, которые можно использовать (см. табл.3). Если этих шаблонов недостаточно, то можно определить свой собственный шаблон, используя функцию setfillpattern().

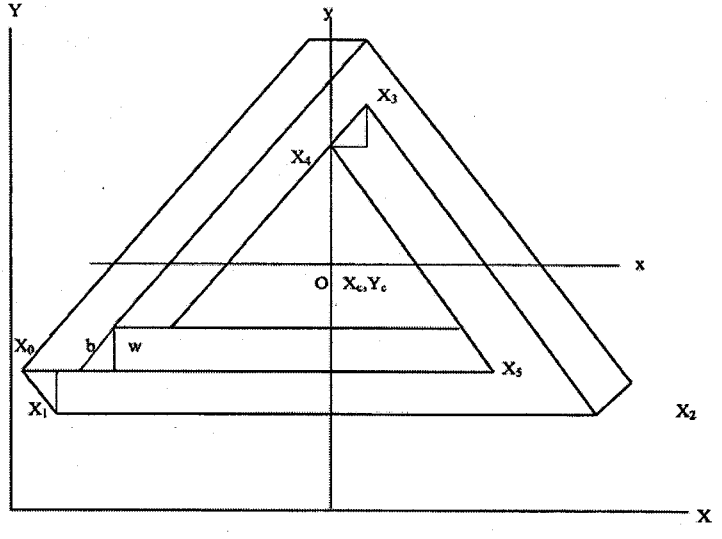


Рис.24

Чтобы построить "магический" треугольник используем тригонометрические функции и свойства равностороннего треугольника. Пусть оси координат проходят через центр (воображаемой) окружности ($x_c$ $y_c$), вписанной во внутренний треугольник, причем ось y направлена снизу вверх.

Радиус вписанной окружности

$$R = 4*R*\sin(A/2)*\sin(B/2)*\sin(C/2),$$

где R – радиус описанной окружности,

A, B, C – углы при вершинах треугольника.

Для равностороннего треугольника $\angle A=\angle B=\angle C=60^0$, следовательно, r = R/2.

Обозначим за *a* длину стороны внутреннего треугольника. Тогда

$$R=a\sqrt{2}*\sin(A)= a*\sqrt{3},\ r=0.5*a*\sqrt{3}/3.$$

Замечаем, что $\sin(120^0)= \sin(60^0)$, а $\cos(120^0) = -\cos(60^0)$. Это нам необходимо для того, чтобы, построив фактически один угол одного из трех треугольников, затем двумя поворотами по $120^0$ построить недостающие грани "магического" треугольника для шести точек. Для примера, приведенного на рис.24, при таком повороте точка $X_0$ перейдет на место точки $X_2$, а точка $X_1$ перейдет в точку Z (не вычисляется).

При составлении программы можно, следовательно, вычислить только $\cos(120^0)$ и $\sin(120^0)$ и значение $\sin(120^0)$ использовать еще для нахождения дополнительного параметра треугольника .

$$B = w/\sin(120^0).\sin(\alpha).$$

где *b* – длина стороны трех малых треугольников, которые отсекаются в вершинах наибольшего треугольника (см. рис.24),

w –толщина стороны каждого закрашиваемого треугольника,

**Рекомендации по составлению программы**

1) Произвести описание всех переменных, (переменная foregcolor=getcolor()).

2) Вычислить вспомогательные переменные:

sq3=sqr(3.0); ha=0.5*a; c= –0.5; s=0.5*sq3; b=w/s; hb=0.5*b.

3) Определяем координаты центра $X_c$, $Y_c$ и координаты вершин треугольника

$X_c = 0.5*X\_max$; $Y_c=0.5(Y\_max –r)$;

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].

4) Организуем два цикла: внешний (по i) для выбора трех цветов окраски и внутренний (по j) для преобразования координат в пиксельные

points[2*j]=($x_c$+xj),

points[2*j+1]=($y_c$+yj).

Координаты xj=x[j], yj=y[j] вычисляются по формулам поворота

x[j]=c*xj –s*yj; c= cos($\alpha$), s= sin($\alpha$).

y[j[=s*xj+c*yj.

5) Для замыкания контура ввести points[12]=points[0]; points[13]=points[1].

6) Установить fillpattern=INTERLEAVE_FILL;

setfillstyle(fillpattern, foregroundcolor );

fillpoly(7, points).

***Задание на выполнение работы***

1. Внимательно ознакомиться с содержанием работы и с геометрическими преобразованиями для построения "магического" треугольника.
2. Составить полную программу построения полигона.
3. Откомпилировать и отладить программу.
4. Внести изменения в программу для получения различных вариантов заливки полигона.

***Содержание отчета***

1. Краткое изложение содержания работы с необходимыми геометрическими построениями.
2. Распечатка программы с подробными комментариями.
3. Выводы по работе.

## ЛИТЕРАТУРА


1. Пореев В.Н. Компьютерная графика. – СПб.: БХВ-Петербрг, 2002. –432 с
2. Березин Б.И., Березин С.Б. Начальный курс Си и С++. – М.: ДИАЛОГ-МИФИ, 2001. – 288 с.
3. Культин Н.Б.Самоучитель С++ Builder. – СПб.: БХВ – Петербург, 2006. – 320 с.
4. Боресков А.В., Шикин Е.В., Шикина Г.Е. Компьютерная графика: первое знакомство/Под ред, Шикина Е.В. – М.: Финансы и статистика, 1996. – 176 с.
5. Аммерал Л. Принципы программирования в машинной графике / Пер. с англ.- М.:"Сол Систем", 1992. – 224 с.
6. Аммерал Л. Программирование графики на Турбо Си / Пер. с англ. – М.: «Сол Систем», 1992. – 221 с.
7. Шишкин А.Д.. Программирование на языке Си. Учебное пособие. СПб.:изд. РГГМУ, 2003. – 104 с.

## *ПРИЛОЖЕНИЕ*

### Программа Circle.c

```
#include <graphics.h>
/*#include"grasptco.h"*/
/*Использование графических функций для рисования окружно-
стей*/
#include<zaggraf.c>
void main(void)
{
int zaggraf(void);
int zaggraf();
int driver, mode, errorcode;
register int i;
int X_max, Y_max;
float xC, yC, d, rmax, r, x_max, y_max;
printf("расстояние в дюймах например 10:");
scanf("%f", &d);
driver=DETECT;
initgraph(&driver, &mode, "");
errorcode=graphresult();
if(errorcode!=grOk) /*если произошла ошибка*/
{
printf("Ошибка:%s\n", grapherrormsg(errorcode));
printf("Для останова нажмите любую клавишу\n");
getch();
exit(1); /*Завершение работы программы*/
}
X_max=getmaxx(), Y_max=getmaxy();
xC=0.5*X_max;
yC=0.5*Y_max; rmax=200.5;
for(r=d;r<=rmax;r+=d)
circle(xC, yC, r);
getch();
closegraph();
}
```

## Программа Clipd1.c

```
/*Program Clipdemo*/
/* Рисование многоугольников, вписанных в квадрат*/
#include <graphics.h>
#include<zaggraf.c>
#include "math.h"
float xmin, xmax, ymin, ymax;
main()
{
int i;
float r, pi, alpha, phi0, phi, x0, y0, x1, y1, x2, y2, dx, dy;
pi=4.0*atan(1.0);
alpha=72.0*pi/180.0;phi0=0.0;
printf("Введите xmin, ymin, xmax, y max\n");
scanf("%f, %f, %f, %f", &xmin, &ymin, &xmax, &ymax);
printf("Введите X0, y0\n");
scanf("%f, %f", &x0, &y0);
zaggraf();
moveto(xmin, ymin);lineto(xmax, ymin);lineto(xmax, ymax);
lineto(xmin, ymax);lineto(xmin, ymin);
for(r=10.5;r<250.5;r+=20.5)
{
 x2=x0+r*cos(phi0);y2=y0+r*sin(phi0);
for(i=1;i<=5;i++)
{
 phi=phi0+i*alpha;
x1=x2;y1=y2;
x2=x0+r*cos(phi);y2=y0+r*sin(phi);
moveto(x1, y1);lineto(x2, y2);
}
}
getch();
closegraph();
 }
```

## Программа Clipdemo.c

```
/*clipdemo1*/
/*Рисование многоугольников */
/*Удаление линий, выходящих за пределы окна*/
#include <graphics.h>
#include<zaggraf.c>
#include "math.h"
float xmin=50.0, xmax=600.0, ymin=80.0, ymax=400.0;
main()
{
int i;
float r, pi, alpha, phi0, phi, x0, y0, x1, y1, x2, y2, dx, dy;
pi=4.0*atan(1.0);
alpha=72.0*pi/180.0;phi0=0.0;
x0=300.0;y0=270.0;
zaggraf();
moveto(xmin, ymin);lineto(xmax, ymin);lineto(xmax, ymax);
lineto(xmin, ymax);lineto(xmin, ymin);
for(r=10.5;r<400.5;r+=20.5)
{
 x2=x0+r*cos(phi0);y2=y0+r*sin(phi0);
for(i=1;i<=5;i++)
{
 phi=phi0+i*alpha;
x1=x2;y1=y2;
x2=x0+r*cos(phi);y2=y0+r*sin(phi);
clip(x1, y1, x2, y2);
}
}
 }
int code(x, y) float x, y;
 { return(x<xmin)<<3|(x>xmax)<<2|(y<ymin)<<1|(y>ymax);
 }
clip(x1, y1, x2, y2) float x1, y1, x2, y2;
{int c1=code(x1, y1), c2=code(x2, y2);float dx, dy;
while(c1|c2)
{ if(c1&c2) return;
```

```
dx=x2 –x1;dy=y2 –y1;
 if(c1)
 { if(x1<xmin){y1+=dy*(xmin –x1)/dx;x1=xmin;} else
  if(x1>xmax){y1+=dy*(xmax –x1)/dx;x1=xmax;}else
  if(y1<ymin){x1+=dx*(ymin –y1)/dy;y1=ymin;}else
  if(y1>ymax){x1+=dx*(ymax –y1)/dy;y1=ymax;}
  c1=code(x1, y1);
  } else
  { if(x2<xmin){y2+=dy*(xmin –x2)/dx;x2=xmin;}else
  if(x2>xmax){y2+=dy*(xmax –x2)/dx;x2=xmax;}else
  if(y2>ymin){x2+=dx*(ymin –y2)/dy;y2=ymin;}else
  if(y2>ymax){x2+=dx*(ymax –y2)/dy;y2=ymax;}
  c2=code(x2, y2);
  }
   }
   moveto(x1, y1);lineto(x2, y2);
   }
```

## Программа Hilbert.c

```
/*HILBERT*/
/*Построение рекурсий*/
#include <graphics.h>
#include"grasptco.c"
#include<zaggraf.c>
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);
 lineto(F.x, F.y);
 Hilbert(F, G, C, dAB, dAC, n –1);
 lineto(H.x, H.y);
 Hilbert(H, I, J, dAB, dAC, n –1);
 lineto(K.x, K.y);
 dAB.x= –dAB.x;dAB.y= –dAB.y;
 dAC.x= –dAC.x;dAC.y= –dAC.y;
 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;
C.x=xC;C.y=yC;
B.x=xB; B.y=yB;

 dAB.x=(xB –xA)/steps;
 dAB.y=(yB –yA)/steps;
 dAC.x=(xC –xA)/steps;
 dAC.y=(yC –yA)/steps;
 moveto(xA, yA);
 Hilbert(A, B, C, dAC, dAB, recdepth);
}
 main()
 { float xCenter, yCenter, h, xP, yP, xQ, yQ, xR, yR;
 printf("\Введите глубину рекурсии:");
 scanf("%d", &recdepth);
 steps=(1<<recdepth) –1;
 zaggraf();
 X_max=600; Y_max=500;
 xCenter=X_max/2;
 yCenter=Y_max/2;
 h=Y_max/30;
```

```
 xP=xR=xCenter –3*h;
 xQ=xCenter+3*h;
 yP=yQ=yCenter –4*h;
 yR=yCenter+4*h;
 setcolor(BLUE);
 square(xQ, yQ, xR, yR, xQ+8*h, yQ+6*h);
 setcolor(SLASH_FILL);
 square(xR, yR, xP, yP, xR –8*h, yR);
 setcolor(GREEN);
 square(xP, yP, xQ, yQ, xP, yP –6*h);
 }
```

## Программа Intria.c

```
/*Program INTRIA */
/*Построение рекурсивного треугольника*/
#include<stdio.h>
#include<graphics.h>
#include "grasptco.c"
#include<zaggraf.c>
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;
 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;
```

```
X_max=500;Y_max=300;
printf("\Глубина рекурсии(например 7):");
scanf("%d", &n);
zaggraf();
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();*/
}
```

## Программа magtria.c

```
#include <graphics.h>
#include<math.h>
#include<stdio.h>
/*МАГИЧЕСКИЙ ТРЕУГОЛЬНИК*/
#include<zaggraf.c>
main(void)
{
float a, b, ha, hb, c, sq3, xC, yC, x[6], y[6], xl, yl, r, w, s;
int points[14], i, l, foregroundcolor, fillpattern;
printf("Магический треугольник\n");
printf("\nРазмер стороны вн-го треуг-ка( например, 100:");
scanf("%f", &a);ha=0.5*a;
sq3=sqrt(3.0);r=ha*sq3/3.0;
c= –0.5;
s=0.5*sq3;
printf("\nВведите параметр толщины(напр.50:");
 scanf("%f", &w);
b=w/s;
hb=0.5*b;
zaggraf();
setcolor(GREEN);
xC=0.5*500; yC=0.5*(300 –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];
foregroundcolor=getcolor();
setcolor(RED);
for(i=0;i<3;i++)
{ for(l=0;l<6;l++)
 {xl=x[l];yl=y[l];
 points[2*l]=(xC+xl);
 points[2*l+1]=(yC+yl);
 x[l]=c*xl –s*yl;
 y[l]=s*xl+c*yl;
 }
 points[12]=points[0];
/* setbkcolor(BLUE);*/
 points[13]=points[1];
 fillpattern=
  (i==0? SOLID_FILL:
  i==1? INTERLEAVE_FILL:
  WIDE_DOT_FILL);
 setfillstyle(fillpattern, foregroundcolor);
  fillpoly(7, points);
  }
 getch();
 closegraph;
 }
```

## Программа strelka.c

```
/* Программа стрелка*/

#include <graphics.h>
#include<zaggraf.c>
#include"math.h"
float x[4]={300.0, 300.0, 310., 290.0}, y[4]={ –10.25, 20.25, 10.0,
10.0};
main(void)
{
int i, j;
```

```
float pi, phi, cos_phi, sin_phi, xx, yy;
pi=4.0*atan(1.0);phi=6*pi/180;
cos_phi=cos(phi);sin_phi=sin(phi);
zaggraf();
for(i=1;i<=14;i++)
{
 for(j=0;j<=3;j++)
 {
  xx=x[j];yy=y[j];
  x[j]=xx*cos_phi –yy*sin_phi;
  y[j]=xx*sin_phi+yy*cos_phi;
  }
  moveto(x[0], y[0]);
  for(j=1;j<=3;j++)
  lineto(x[j], y[j]);
  lineto(x[1], y[1]);
  }
  getch();
  closegraph();
   }
```

## Программа CUBE

```
/*Проволочная модель куба*/
#include "grasptco.c"
#include<graphics.h>
#include<zaggraf.c>
#include"math.h"
float v11, v12, v13, v21, v22, v23, v32, v33, v43,
screen_dist, c1, c2;
int main(void)
{ float rho, theta, phi, h=10.0;
printf("Расстояние до наблюдателя rho=EO:");
scanf("%f", &rho);
printf("\n3адайте два угла в градусах.\n");
printf("\nУгол theta измеряется по горизонтали от оси x:");
scanf("%f", &theta);
printf("Угол phi измеряется по вертикали от оси z:");
```

```
scanf("%f", &phi);
printf("Расстояние от точки наблюдения до экрана (например,
….):");
scanf("%f", &screen_dist);
printf("Коэффициент c1=");
scanf("%f", &c1);
printf("Коэффициент C2=");
scanf("%f", &c2);
coeff(rho, theta, phi);
zaggraf();
mv(h, –h, –h); dw(h, h, –h);
dw( –h, h, –h);
dw( –h, h, h);
dw( –h, –h, h);
dw(h, –h, h);
dw(h, –h, –h);
mv(h, h, –h); dw(h, h, h);
dw( –h, h, h);
mv(h, h, h); dw(h, –h, h);
mv(h, –h, –h);dw( –h, –h, –h);
dw( –h, h, –h);
mv( –h, –h, –h);dw( –h, –h, h);
return(0);
}
coeff(rho, theta, phi)
 float rho, theta, phi;
{ float th, ph, costh, sinth, cosph, sinph, factor;
factor=atan(1.0)/45.0;
/*Углы в радианах*/
th=theta*factor; ph=phi*factor;
costh=cos(th); sinth=sin(th);
cosph=cos(ph); sinph=sin(ph);
/*Элементы матрицы V см.(4.9) */
v11= –sinth; v12= –cosph*costh; v13= –sinph*costh;
v21=costh; v22= –cosph*sinth; v23= –sinph*sinth;
v32=sinph; v33= –cosph;    v43=rho;
return(0);
}
```

```
mv(x, y, z) float x, y, z;
{ float X, Y ;
perspective(x, y, z, &X, &Y);
moveto(X, Y);
return(0);
}
dw(x, y, z) float x, y, z;
{ float X, Y;
  perspective(x, y, z, &X, &Y);
 lineto( X, Y);
 return dw();
 }
perspective(x, y, z, pX, pY)
 float x, y, z, *pX, *pY;
{  float xe, ye, ze;
/* Координаты , вычисляемые по (2)*/
xe=v11*x+v21*y;
ye=v12*x+v22*y+v32*z;
ze=v13*x+v23*y+v33*z+v43;
/* Экранные координаты, вычисляемые по (3) и (4)*/
*pX=screen_dist*xe/ze+c1;
*pY=screen_dist*ye/ze+c2;
return perspective();
}
```

## Вспомогательная программа GRASPTCO

```
#include <graphics.h>
#include<conio.h>
int X_max, Y_max, foregrcolor, backgrcolor, colorsum;
float x_max=100.0, y_max=150.0, 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)
{ moveto(IX(x), IY(y));
 }
 void draw(float x, float y)
 { lineto(IX(x), IY(y));
 }
 void line_uc(float x1, float y1, float x2, float y2)
 { 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));
   }
```

**//Программа «График» для построения графика в среде C++Builder**

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
```

```
   : TForm(Owner)
{
}
#include "math.h"
float f(float x)
{return 2*sin(x)*exp(x/5);}
//---------------------------------------------------------------------------

void __fastcall TForm1::onPaint(TObject *Sender)
{
  float x1, x2, y1, y2;// границы изменения аргумента и значения
функции
  float x, y;// аргумент функции и значение функции в точке x
  float dx; //приращение аргумента
  int l, b ;//левый нижний угол области вывода
  int w, h; // ширина и высота области вывода
  int x0, y0; // начало координат
  float mx, my; // масштаб по осям X и Y
// Область вывода графика
  l=10;
  b=Form1 ->ClientHeight –20;
  h= Form1 ->ClientHeight –40;
  w=Form1 ->Width –20;
  x1=0; //нижняя граница диапазона аргумента
  x2=25; // верхняя граница диапазона аргумента
  dx=0.05;
  y1=f(x1); //минимальное значение функции
  y2=f(x2); // максимальное значение функции
  do
  {
  y=f(x);
  if(y<y1) y1=y;
  if(y>y2) y2=y;
  x+=dx;
  }
  while(x<=x2);
  my=(float)h/(abs)(y2 –y1); // масштаб по оси Y
  mx=w/(abs)(x2 –x1); //масштаб по оси X
```

```
//Оси
 x0=l+abs(x2 –x1);
 y0=b –abs(y1*my);
 Canvas –>MoveTo(x0, b); Canvas –>LineTo(x0, b –h);
 Canvas –>MoveTo(l, y0); Canvas –>LineTo (l+w, y0);
 Canvas –>TextOutA(x0+5, b –h, FloatToStrF(y2, ffGeneral, 6, 3));
 Canvas –>TextOutA(x0+5, b, FloatToStrF (y1, ffGeneral, 6, 3));
 Canvas –>TextOutA(x0+5, y0, 0);
 Canvas –>TextOutA(x0+150, y0 –150, "График функции
y=2sin(x)exp(x/5)");
 x=x1;
//Построение графика
 do
 {
 y=f(x);
 Canvas –>Pixels[x0+x*mx][y0 –y*my]=clRed;
 x+=dx;
 }
 while(x<=x2);
}
```

## СОДЕРЖАНИЕ

55=00

Учебное издание

А.Д. Шишкин, Е.А. Чернецова

# ПРАКТИКУМ
по дисциплине
«КОМПЬЮТЕРНАЯ ГРАФИКА»

Конспекты лекций

*Редакторы:* И.Г. Максимова, Л.В. Ковель
*Компьютерная верстка* Н.И. Афанасьевой

ЛР № 020309 от 30.19.96.

Подписано в печать 14.11.08. Формат 60×90 $^{1}/_{16}$. Гарнитура Times New Roman.
Бумага офсетная. Печать офсетная. Уч.-изд. л. 4,5. Тираж 250 экз. Заказ № 06/09.
РГГМУ, 195196, Санкт-Петербург, Малоохтинский пр., 98.
ЗАО «НПП «Система», 197045, Санкт-Петербург, Ушаковская наб., 17/1.