11 класс

Проект 1 к главе 1 «Графика и визуализация»

Теоретический минимум

Графика на плоскости (2D)

Всем известно, что большую часть сведений и представлений об окружающем мире человек получает с помощью зрения. Далеко не всё из того что мы видим, можно без потери смысла описать в виде наборов цифр или текста. Поэтому очень важно существование средств представления информации «для зрения» и автоматических средств обработки такой информации.

Общее направление, в рамках которого решаются такие задачи, получило название компьютерной графики. Другими словами, ***компьютерной графикой*** *называют область деятельности, в которой компьютеры и программное обеспечение используются в качестве инструмента создания и обработки изображений.*

Из личного опыта вы прекрасно представляете себе что такое “компьютерная графика”. По сути, это набор средств формирования изображения с помощью компьютера - всех средств, включая способы и аппаратуру создания, получения и хранения цифрового изображения, его отображения, передачи и обработки.

Разнообразие этих средств огромно, графика - одно из самых динамичных (и коммерчески востребованных) приложений всей ИТ-отрасли.

Первый вопрос, который нужно решить при организации обработки графических данных — это представление и кодирование цвета.

В простейшем случае, когда возможных цветов всего два — используется один бит, состояние которого и задает цвет. Если же цветов становится больше, то такой подход уже не позволяет решить поставленную задачу.

Существует несколько способов кодирования цвета, применяемых при обработке графики.

Для описания градации одного цвета применяется обычное кодирование, в котором номер обозначает **градацию**. Чем больше значение, тем сильнее проявляется цвет. Для мониторов (в которых точка самостоятельно излучает свет) обычно 0 соответствует отсутствию цвета, а максимальное значение (например, 255) — максимальной светимости точки. Таким образом, появляется возможность задавать **оттенок** на монохромном мониторе.

В случае, когда используется печатающее устройство и чернильная точка либо есть, либо нет, оттенок задается некоторой матрицей (например, 4×4 точки), количество цветовых точек которой формирует оттенок.

В более сложных случаях, когда речь идет о кодировании сложного цвета с большим количеством оттенков, рассматривают разложение цвета на несколько отдельных компонентов, которые смешиваясь в одной точке образуют заданный цвет.

Для каждого конкретного изображения все, что передается одним из компонентов цвета, также называется *каналом*. Компоненты цвета и способ образования из них видимого оттенка и представляют собой **цветовую модель.**

Цветовые модели разрабатывались в психологии восприятия задолго до появления вычислительной техники. Существует большое количество цветовых моделей, которые создавались и вводились разными авторами для описания и исследования зрения человека. С появлением проекционной и печатающей аппаратуры, с учетом технических

требований были разработаны новые модели, учитывающие в первую очередь физические и технические аспекты формирования конкретного цвета.

Для формирования изображения на экране мы будем пользоваться **аддитивной цветовой моделью RGB** (рис. 1.3), в которой цвет образуется смешиванием трех компонентов:

- **R**ed — красного;
- **G**reen — зеленого;
- **B**lue — голубого.

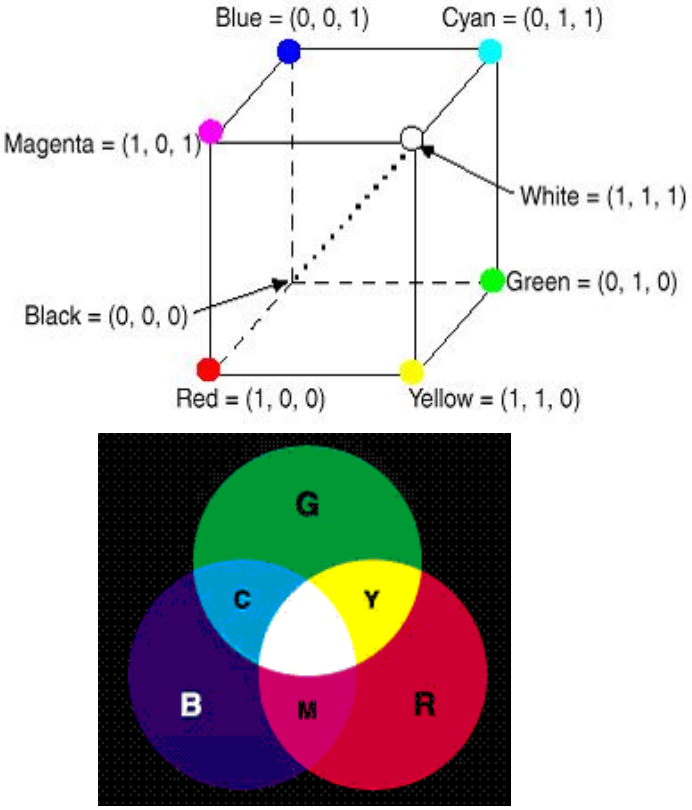


Рис. 1. Аддитивная цветовая модель RGB

Эта модель описывает цвет, который образуется из «суммы» света, излучаемого несколькими источниками. Эта модель является **аддитивной** (от лат. *additio* — прибавляю).

Самым популярным примером использования этой модели являются мониторы (в которых цвет каждого пикселя растра складывается из трех компонентов), проекторы и сканеры (которые чаще всего регистрируют отраженный свет).

Именно такая цветовая модель используется и в описании возможностей различных графических устройств. Цветовое пространство в этом случае описывают количеством битов, отводимых на сохранение цвета. Чаще всего используются режимы HighColor (16 бит, в соотношении 5:6:5 или 5:5:5) и TrueColor (24 бита, в соотношении 8:8:8).

Профессиональные программы обработки графической информации позволяют работать с расширенным представлением, когда на одну компоненту отводится не 8, а 16 бит.

Каждый компонент задается силой светимости, 0 соответствует отсутствию света. Таким образом, цвет 0-0-0 — это черный, цвет из равных долей каждого компонента — один из оттенков серого, а цвет с максимальными значениями компонентов — белый.

Второй вопрос, на который нужно ответить, создавая изображение с помощью компьютера, это вопрос о том что именно мы будем "расцвечивать" - а точнее, что как мы будем графическую информацию хранить и показывать? От этого будет зависеть и обработка и передача. Основных способов два:

Первый способ, которым чаще всего представляется графическая информация, получил название *растрового*. ***Растровая графика*** *— способ представления и хранения изображения в виде обозначения цвета точек (пикселей), находящихся в узлах прямоугольной равномерной координатной сетки —* ***растра.***

Если сетка достаточно плотная, то и точки получаются мелкие, поэтому человеческий глаз не воспринимает изображение как дискретное.

Основными параметрами изображения в растровой форме являются **разрешение линейное,** т. е. возможное количество точек на единицу площади, и **разрешение цветовое** — количество градаций цвета. Таким образом, различают разрешение линейное — количество столбцов по горизонтали и линий по вертикали, и цветовое/оттеночное — количество оттенков или цветов у каждой точки. Линейное разрешение описывают возможным количеством точек, а цветовое — в виде количества битов, отводимых на описание каждой отдельной точки.

Чем больше количество точек на единицу площади и количество цветов каждой точки, тем выше возможное качество изображения.

Второй способ - **векторная графика.** Растровая графика является чрезвычайно мощным способом представления и хранения изображений, особенно фотографических, но в ряде случаев прямое использование такого подхода неудобно. Это случаи, когда изображение создается из типовых элементов — **графических примитивов** (точек, прямых и кривых линий и т. п.). Представление таких элементов в виде точек лишает нас возможности менять параметры примитива без перерисовки изображения, ограничивает возможности геометрических преобразований, требует много места при хранении.

Для преодоления этих ограничений применяется подход, подразумевающий хранение и обработку изображения не в виде растра, а в виде некоторых описаний отдельных элементов. Элементами обычно являются математические объекты с заданными конкретными параметрами. Параметры позволяют выполнить визуализацию элементов на устройстве вывода (**растеризацию**), исходя из его характеристик и заданного «окна» просмотра.

Поскольку пространственное положение примитивов и способ отображения задаются с помощью координат, этот способ хранения и обработки изображений получил название **векторной графики.**

Одним из наиболее существенных достоинств векторной формы представления изображения является ее компактность и малая зависимость объема от размеров изображения.

К минусам этой формы представления относится отсутствие общих стандартов (практически у каждого редактора есть свои собственные форматы и особенности) и высокие требования к системным ресурсам, особенно — вычислительным.

Тем не менее, если мы синтезируем изображение программно, то чаще всего именно векторную графику мы будем использовать как основу для работы.

Здесь мы не будем затрагивать вопрос обработки изображений с помощью специализированных графических пакетов, а попробуем описать основу их работы - основные приемы и алгоритмы, которые используют при их создании.

Для решения этих задач мы воспользуемся уже знакомой средой программирования - PascalABC.Net

Обработка растровых изображений. Фильтры.

Как мы знаем, множество растровых изображений мы получаем из реального мира - с помощью фотографий, видеосъемки, обработки значений датчиков и т.д. Полученное изображение даже если оказывается достаточного качества (что бывает далеко не всегда) часто требует изменений в зависимости от вашего замысла или предпочтений.

Нам требуются средства, которые позволяют менять растровое изображение целиком. Основой очень значительной части таких средств являются ***фильтры***.

Фильтры - это способ изменения изображения, при котором оно "пропускается" через преобразование и обрабатывается поточечно. Самый известный способ такого преобразования - применение *матрицы-свертки*, в которой цвет каждой точки - это сумма цветов окружающих ее точек с некоторыми коэффициентами.

К каждой точке растра, находящейся в зоне действия фильтра, применяется некоторое действие для расчета ее нового цвета. Очень часто это действие основано на *весовой матрице,* которая называется *ядром преобразования*, то есть матрице с нечетной длиной и шириной, в которой указан вес (то есть доля участия) пикселя в итоговом значении. Матрица накладывается на изображение так, чтобы ее центр (именно поэтому длина и ширина выбраны нечетными) совпал с изменяемым пикселем. Параметры пикселей изменяемой зоны, попавшие под матрицу, суммируются с весом, указанным в матрице, результат нормируется (т. е. пересчитывается в доли от общего веса) и записывается в центральный пиксель. После этого матрица сдвигается на 1 пиксель и все повторяется.

Вот простой фильтр ("размытие"):

1	1	1
1	1	1
1	1	1

При наложении на изображение действовать это будет примерно так:

14	4	55	44	44	55
12	64	44	221	33	34
22	55	22	127	22	22
12	44	33	67	44	55
11	33	77	23	23	22
135	44	65	34	22	34

Цвет в центральной точке будет рассчитан как сумма всех цветов внутри синего квадрата, деленный на суммарный вес, с округлением до целого: (64*1+44*1+221*1+55*1+22*1+127*1+44*1+33*1+67*1)/9 ≈ 75

Такой расчет делается для каждого компонента цвета.

Реализуем средство наложения такого фильтра и посмотрим, чего можно таким образом добиться.

Первое, что нам понадобится - исходное изображение. Получение, сжатие, декодирование - сложные задачи, на решение которых здесь у нас нет времени. Воспользуемся готовыми средствами. В составе нашей среды есть основанные на средствах Windows библиотеки работы с изображениями.

```
uses ABCObjects,GraphABC;

var
     p : Picture; // Специальный объект класса Picture для работы с изображением
```

```
begin
 p := new Picture( 'example.jpg'); // Создание объекта из
файла

 p.Draw(0,0); // Отрисовка изображения в рабочем окне
 while true do  ; // Ждем закрытия окна
end.
```

Для работы с изображением нам понадобятся средства для доступа к отдельным пикселям изображения. Для этого у объекта Picture есть методы GetPixel и setPixel - то есть получения и записи отдельного пикселя по его координатам.

Метод Picture.GetPixel вернет объект Color со свойствами R,G,B - которые нам и нужны. Изменить эти свойства нельзя, поэтому новое значение мы “соберем” из компонентов с помощью функции RGB(r,g,b)

Теперь мы реализуем функцию наложения фильтра. Она будет получать объект-исходное изображение, объект-результат для изображения с наложенным фильтром и сам фильтр.

Изначально, для простоты и понятности, фильтр будет целочисленной матрицей 5*5 (размер мы оговорим константой). Напомним - этот размер должен быть нечетным.

Мы будем накладывать эту матрицу целиком, начиная с точки (2,2) - чтобы она поместилась целиком. Перед наложением мы рассчитаем суммарный вес всех точек в матрице - чтобы выполнить нормирование. Результат применения - то есть цвет точки - будет называться ***откликом фильтра.***

```
const
 filterSize = 5;
 half = 2;

type
 filter = array[0..filterSize-1,0..filterSize-1] of real;

procedure filterApply( s,t : Picture; filterMatrix :
filter);
var
 i,j,ip,jp : integer;
 dv : real;
 rr,rg,rb : real;
 pnt : Color;
begin
 dv := 0; // Рассчитаем делитель - для нормирования
 for i:=0 to filterSize-1 do
    for j:=0 to filterSize-1 do
       dv := dv + filterMatrix[i,j];
 if dv = 0 then dv := 1;
 for ip:=half to s.Height-half-1 do
    for jp:=half to s.Width-half-1 do
    begin
       rr :=0;
       rg :=0;
       rb :=0;
       for i:= -half to half do // накладываем фильтр-
свертку - захватывая пиксели вокруг
            for j:= -half to half do
```

```
                begin
                   pnt := s.GetPixel(jp+j,ip+i);
                   rr := rr + pnt.R*filterMatrix[i+half,j+half];
                   rg := rg + pnt.G*filterMatrix[i+half,j+half];
                   rb := rb + pnt.B*filterMatrix[i+half,j+half];
                end;

t.SetPixel(jp,ip,RGB(round(rr/dv),round(rg/dv),round(rb/dv)) );
          end;
      end;
```

В основной программе ничего особенного происходить не будет:

```
begin
 p := new Picture( 'example.jpg');
 r := new Picture( p.Width ,p.Height );
 filterApply(p,r,blur);
 r.Draw(0,0);
 while true do
    ;

end
```

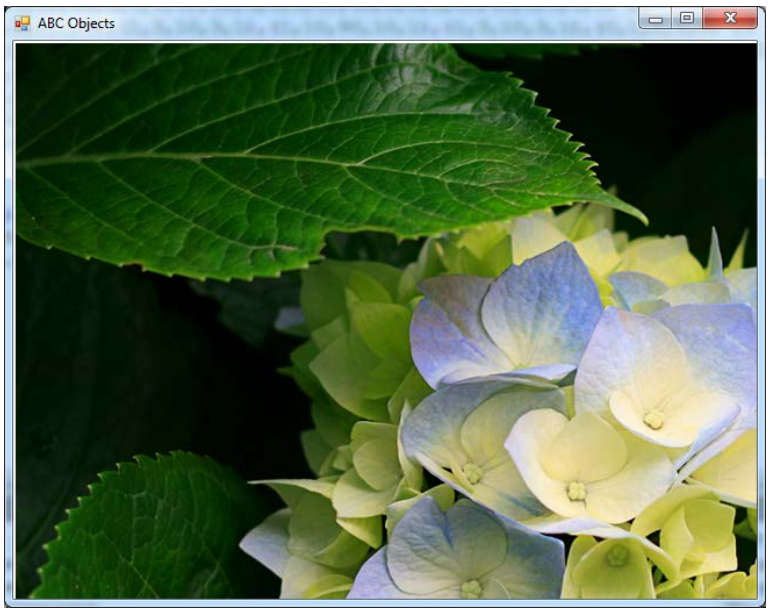
Имя фильтра blur мы пока не описали - это и есть матрица свертки. Содержание матрицы очень простое:

1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

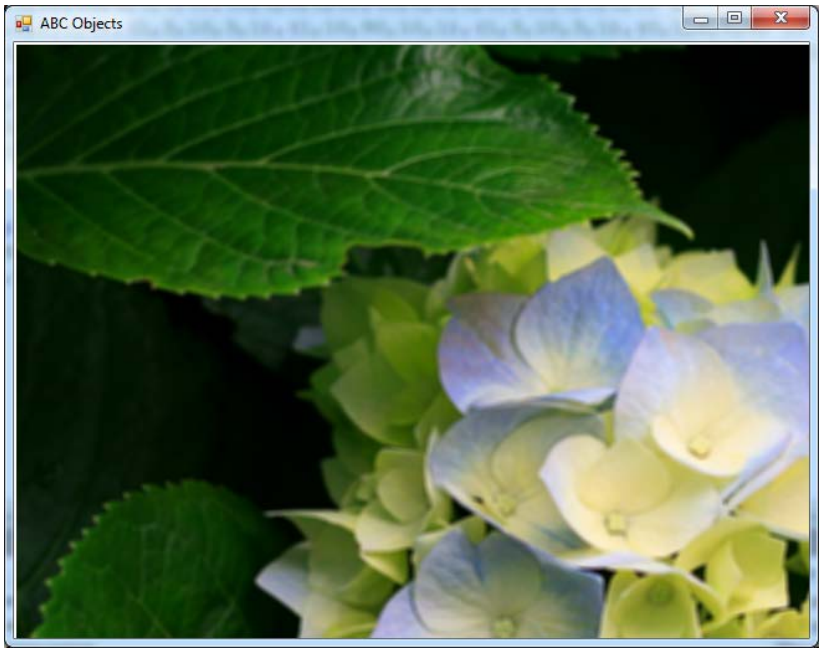
То есть в этом фильтре цвет по всем компонентам будет смесью окружающих точек. Опишем и применим его:

blur : filter := ((1,1,1,1,1),(1,1,1,1,1),(1,1,1,1,1),(1,1,1,1,1),(1,1,1,1,1));

Было:



Стало:



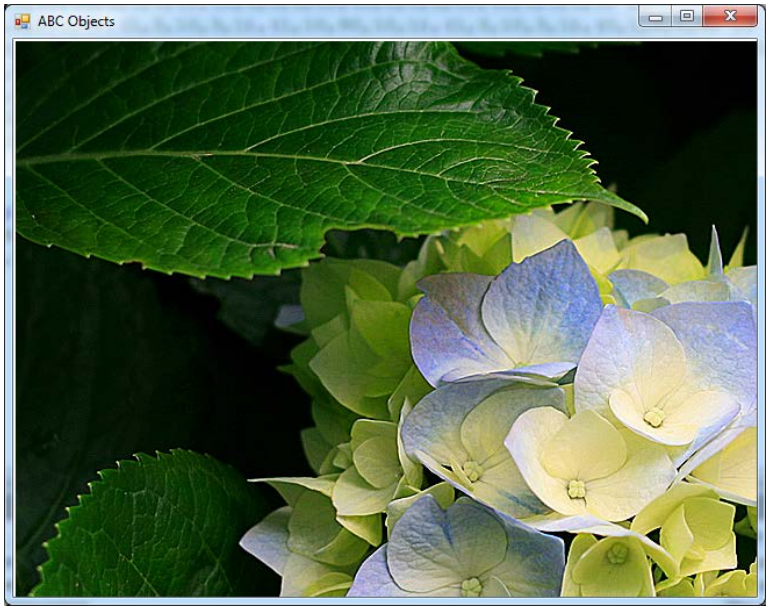
Вот еще несколько фильтров на пробу:
Мягкое размытие:

0	1	2	1	0
1	3	10	3	1
1	10	90	10	1
1	3	10	3	1
0	1	2	1	0

Резкость:

0	0	0	0	0
0	-0.25	-0.25	-0.25	0
0	-0.25	3	-0.25	0
0	-0.25	-0.25	-0.25	0
0	0	0	0	0

Результат применения:



У приведенных выше фильтров может быть несколько вариантов, которые вы можете без труда найти в сети и сравнить результаты. Отметим, что "сила" с которой действует фильтр будет зависеть не от величины коэффициентов, а от фактических размеров матрицы. Чем больше пикселов задействовано - тем заметнее эффект.

Теперь доработаем нашу функцию так, чтобы увеличить возможности применения фильтров. Для этого мы будем передавать (а не рассчитывать) коэффициент нормирования и величину смещения - для компенсации.

Функция будет выглядеть примерно так:

```
const
   filterSize = 5;
   half = 2;

type
   filter = array[0..filterSize-1,0..filterSize-1] of real;

procedure filterApply( s,t : Picture; filterMatrix : filter;
dv,off : integer);
var
   i,j,ip,jp : integer;
   rr,rg,rb : real;
   pnt : Color;
begin

   for ip:=half to s.Height-half-1 do
      for jp:=half to s.Width-half-1 do
      begin
         rr :=0;
         rg :=0;
         rb :=0;
         for i:= -half to half do
            for j:= -half to half do
```

```
                    begin
                       pnt := s.GetPixel(jp+j,ip+i);
                       rr := rr + pnt.R*filterMatrix[j+half,i+half];
                       rg := rg + pnt.G*filterMatrix[j+half,i+half];
                       rb := rb + pnt.B*filterMatrix[j+half,i+half];
                    end;

t.SetPixel(jp,ip,RGB(test(round(rr/dv)+off),test(round(rg/dv)+of
f),test(round(rb/dv)+off)) );
            end;
      end;
```

Обратите внимание, мы используем функцию test. Задача этой функции - убирать значения меньше 0 и больше 255. Здесь эта функция очень проста - на все значения меньше 0 возвращает 0, на все больше 255 - 255. Полагаем, читатели вполне в состоянии написать эту функцию сами.

Для повышения качества стоило бы не "обрубать" значения, а определить шкалу значений яркости и пересчитать все цвета пропорционально их положению на этой шкале.

С этими изменениями мы можем, например, повысить яркость изображения: просто добавив число ко всем компонентам цвета и применив нейтральный фильтр:

```
begin
        p := new Picture( 'example.jpg');
        r := new Picture( p.Width ,p.Height );
        filterApply(p,r,neutral,1,120);
        r.Draw(0,0);
        while true do
           ;

      end.
```

Еще один пример - инверсия цвета. Этот фильтр имеет только одно ненулевое значение - -1 в центре, а для "переворота" цвета добавляет смещение - 256.

Фильтров, приводящих к появлению самых разных эффектов- великое множество. Результаты зависят и от матрицы, и от того как она накладывается, и от способов создания отклика и т.д.

Задания.

Возможностей для экспериментирования у вас масса. Вот несколько вариантов:

1. Свертка накладывать на все пиксели, включая край. На край она накладывается частично - с пересчетом делителя нормирования.

2. Перед наложением край изображения масштабировать так, чтобы размеры стали больше на величину свертки.

3. Фильтр накладывать прямо на изображение - используются и уже обработанные и еще не обработанные точки.

Полученные результаты в виде работающих программ предъявите учителю для оценивания.

Задания для самостоятельной работы:

1. Размытие по Гауссиане может быть выполнено, в частности, с помощью такой свертки:

$$\begin{pmatrix} 1 & 2 & 3 & 2 & 1 \\ 2 & 4 & 5 & 4 & 2 \\ 3 & 5 & 6 & 5 & 3 \\ 2 & 4 & 5 & 4 & 2 \\ 1 & 2 & 3 & 2 & 1 \end{pmatrix}$$

   Примените его и сопоставьте результат с "обычным" размытием.
2. Один из практически важных фильтров - усреднение. В этом фильтре все значения в матрице сортируются и отклик фильтра - среднее значение. Реализуйте такой фильтр, проанализируйте его поведение для серых и цветных изображений.
3. Эффекта "Акварельности" изображения добиваются так: применяют фильтр усреднения, а потом к результату - фильтр резкости. Реализуйте этот комбинированный фильтр.

Графические примитивы. Растеризация.

Следующая важная задача, которую решают при разработке средств машинной графики - это отрисовка примитивов. Практически все современные аппаратные средства в машинной графике - растровые, то есть так или иначе опираются на координатную сетку пикселей. Но любой графический примитив - например, отрезок, - состоит (а точнее - проходит) из нескольких пикселей. Возникает вопрос - какие из них нужно закрасить?

Приведем пример одного из основных алгоритмов, используемых в машинной графике в этом случае, алгоритма построения отрезка. На его примере можно увидеть некоторые основные приемы решения таких задач.

Построение отрезка для нас означает заполнение тех точек растра, которые находятся на отрезке, соединяющем точки с заданными координатами. Очевидно, что просто вычислить эти координаты точно нельзя — поскольку не может быть «дробных» точек. Для простоты мы будем считать, что координаты определены именно в точках растра (т.е. их не нужно пересчитывать).

Самый распространенный алгоритм рисования отрезков получил название *алгоритма Брезенхема*.

Основная идея этого алгоритма — не использовать вычисления с плавающей точкой, а просто «пройти» по одной из осей и определить для каждой координаты в пределах линии нужно ли для нее «сдвинуть» значение второй координаты.

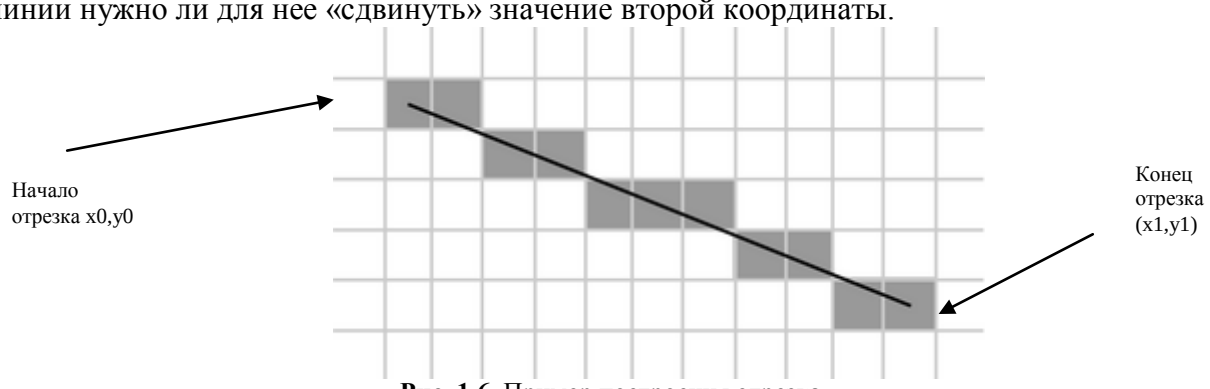


Рис. 1.6. Пример построения отрезка

Например, рассмотрим алгоритм для рисования линии на рис. 1.6, приняв координаты концов отрезка x_0, y_0 и x_1, y_1. Чтобы упростить понимание алгоритма, сначала запишем его на псевдокоде с использованием дробных чисел, считая, что исходное изображение — объект Image, все пиксели в нем — внутренняя матрица Pixel.

```
Deltax = x1 – x0
Deltay = y1 – y0
error = 0
deltaErr = Deltay/Deltax
y = y0
for x = x0 to x1
 Image.Pixel[x,y] = 1
 error = error + deltaErr
 if error > 0.5
      y = y +1
      error = error – 1
```

Таким образом, мы вычислили коэффициент наклона, и увеличивали *y* тогда, когда прошли больше половины пикселя (то есть накопили ошибку в переменной error больше, чем 0,5). Чтобы перейти к целым числам просто умножим дробные числа на Deltax и на 2 (там, где использовался коэффициент 0,5):

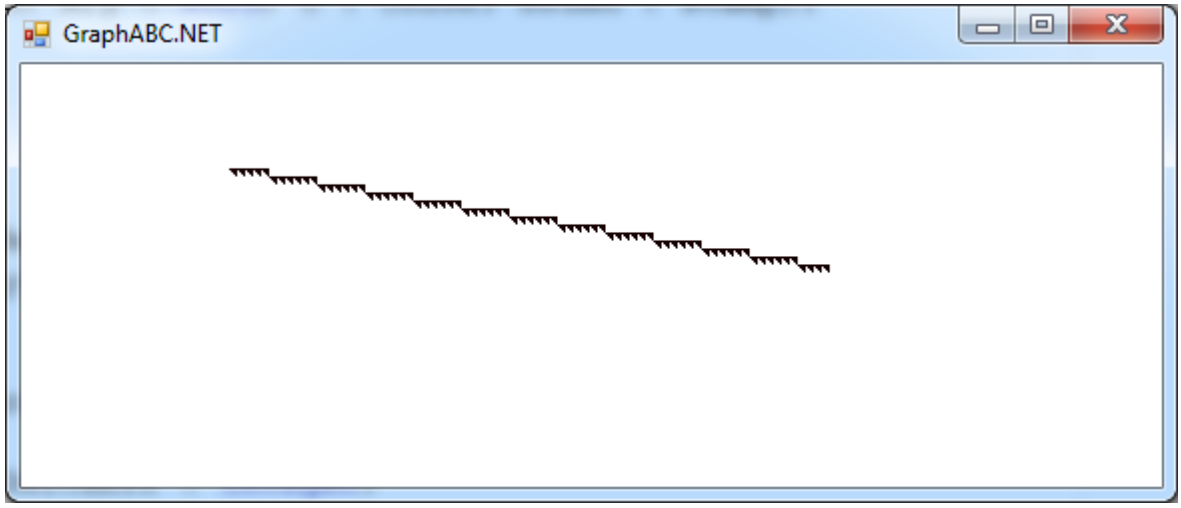
```
Deltax = x1 – x0
Deltay = y1 – y0
error = 0
deltaErr = Deltay
y = y0
for x = x0 to x1
 Image.Pixel[x,y] = 1
 error = error + deltaErr
 if 2*error > Deltax
      y = y +1
      error = error – Deltax
```

Чтобы получить полный алгоритм нужно учесть и другие случаи наклона отрезка. Обычно их учитывают меняя местами координаты и направления сдвига точек. Аналогично можно реализовать алгоритмы построения окружностей и эллипсов с осями параллельными осям координат.

Алгоритм можно использовать и для большего разнообразия линий, например, задав шаблон в виде битовой карты (рисовать точку или не рисовать), получать пунктирные линии, использовать не просто точку, а фигуру сложной формы (кисть) и т. д.

Задание.

Используем этот алгоритм для решения такой задачи: написать процедуру построения линии, нарисованной треугольниками или любыми фигурами, вписанными в размер 4*4 пикселя. Примерно так:



Типовые встроенные функции такого сделать не позволяют: даже рисование произвольной кистью приведет к появлению сплошной линии.
Но мы знаем, как строится линия в принципе - используем это для рисования линии не пикселями, а штампами 4*4.
Сначала напишем процедуру рисования штампом:

```
type
   stamp = array[0..3,0..3] of byte;

procedure DoStamp(s:Picture; x,y : word; c : Color; brush :
stamp);
var
   i,j : byte;
begin
   for i:=0 to 3 do
      for j:=0 to 3 do
         if brush[j,i]=1 then
           s.SetPixel(x+j,y+i,c);
end;
```

Как видно, ничего сложного - штамп будет задан массивом 4*4, в котором там где нужна точка цвета "c" стоит 1.

Теперь рисование линии:

```
procedure MyLine(s : picture; x0,y0,x1,y1 : Word; c : Color;
brush : stamp);
var
   Deltax, Deltay, error, DeltaErr : integer;
   x ,y, stepx, stepy : integer;
begin
   if (x1 < x0 ) then
      stepx := -4
   else
      stepx := 4;
   if (y1 < y0 ) then
      stepy := -4
   else
      stepy := 4;
```

```
   Deltax := abs(x1 - x0);
   Deltay := abs(y1 - y0);
   error  := 0;
   deltaErr := Deltay;
   y := y0;
   x := x0;
   while x <> x1 do
   begin
      DoStamp(s,x,y,c,brush);
     error := error + deltaErr;
       if 2*error > Deltax then
        begin
            y := y +stepy;
            error := error - Deltax;
       end;
       x := x + stepx;
     end;
end;
```

Единственное существенное отличие от алгоритма на псевдокоде, это учет того что точки могут быть заданы в любом порядке - это определит направление шага по каждой координате. Шаг равен 4 - поскольку это размер штампа.

Основная программа как и в случае с фильтром проста:

```
var
   p : Picture;
   triangle : stamp := (
(1,0,0,0),(1,1,0,0),(1,1,1,0),(1,1,1,1));
begin
  p := new Picture( 500,500);
  MyLine(p,400,100,100,50,rgb(20,0,0),triangle);
  p.Draw(0,0);
  while true do
      ;
     end.
```

Как и в прошлом примере, простора для экспериментов много: размер и форма штампа, использование фона, режимы наложения и т.д.

Рисование отрезков - далеко не единственный алгоритм Брезенхема. Аналогично строятся, например, эллипсы и дуги эллипсов.

Основы векторной графики. Преобразования координат.

Исторически первые задачи, которые решались в компьютерной (машинной) графике — это задачи создания изображения (графика, чертежа и т. д.). При подготовке таких рисунков очень часто возникает задача преобразования координат для отображения, преобразования объектов — масштабирования, поворота и т. д.

Такие задачи решаются методами аналитической геометрии с помощью преобразования координат.

Одна из базовых задач, которую мы решим таким способом это пересчет координат из математической системы координат - в экранную. Как вы знаете, в математике 0 - в центре или слева снизу, оси направлены слева направо и снизу вверх. На экране точка отсчета сверху слева, и ось у направлена вниз. Придется пересчитать.

Предположим, что у нас есть область для отображения, заданная координатами углов (X_0, Y_0), (X_1, Y_1). Нам нужно отобразить в ней объект (например, график функции), координаты которого укладываются в пределы (x_0, y_0), (x_1, y_1).

Тогда предварительно рассчитаем коэффициенты: $kx = (X_1 - X_0)/(x_1 - x_0)$ и $ky = (Y_1 - Y_0)/(y_1 - y_0)$.

Координаты точки на экране можно будет рассчитать так:

$$\begin{cases} X = [(x - x_0) \cdot kx + X_0]; \\ Y = [(y - y_0) \cdot ky + Y_0]. \end{cases}$$

Квадратными скобками мы обозначим операцию получения целой части числа. Если важно соблюдать масштаб, то коэффициент выбирается один — наименьший.

Фигуру, заданную координатами точек (всех или только ключевых) можно легко преобразовывать.

Сдвиг точки — это просто добавление чисел (координат вектора) к координатам.

$$\begin{cases} x' = x + a; \\ y' = y + b. \end{cases}$$

Увеличить или уменьшить фигуру (и не обязательно оставлять ее на месте) можно просто умножив координаты ее точек на коэффициент. Если фигура должна сохранить логическое положение на «листе», то координаты предварительно придется пересчитать в систему координат с центром в середине фигуры (x_0,y_0), определяемым как среднее арифметическое координат.

$$\begin{cases} x' = x_0 + k * (x - x_0); \\ y' = y_0 + k * (y - y_0) \end{cases}$$

Наконец, поворот точки на угол вокруг начала координат будет описываться так:

$$\begin{cases} x' = x * \cos\alpha - y * \sin\alpha; \\ y' = x * \sin\alpha + y * \cos\alpha \end{cases}$$

Если поворот нужен не вокруг начала - мы можем использовать перенос координат.

Важное свойство таких преобразований — сохранение формы фигуры. То есть линия преобразуется в линию, а поэтому можно пересчитать только координаты начала и конца отрезка, а промежуточные точки рассчитывать заново не надо.

Используя эту теорию, решим задачу синтеза достаточно сложного изображения: многолучевой звезды. Пусть для определенности лучей будет 7.

Начнем с постановки задачи. Звезда фактически представляет собой симметричный относительно центра многоугольник, в котором используются два радиуса: длинный и короткий. При этом "чередуются" они через равные углы. Вершины соединяются между собой.

Пусть первая вершина на оси ординат, тогда остальные вершины мы сможем вычислить поворотом на нужный угол.

Для реализации на потребуется несколько вещей:

Во-первых, типы для хранения координат - математических и экранных

Во-вторых, функции для выполнения поворота точки на угол

В-третьих, средства рисования ломаной линии по точкам.

Первая часть ничего сложного из себя не представляет:

```
type
   pointR = record
```

```
      x,y : real;
   end;
   pointS = record
      x,y : word;
   end;
   screenCalc = record
      kx,ky : real;
      x0,y0 : word;
      xm0,ym0 : real;
   end;
```

Тут две простых структуры с координатами и структура для хранения данных пересчета (по формулам, которые мы приводили)

Теперь можно написать функции преобразований:

```
// Расчет коэффициентов преобразования в экранные координаты
function calcScreen( x0,y0, x1,y1 : word; xm0,ym0,xm1,ym1 : real
) :screenCalc;
begin
   Result.kx := (x1-x0)/(xm1-xm0);
   Result.ky := (y1-y0)/(ym1-ym0);
   Result.x0 := x0;
   Result.y0 := y0;
   Result.xm0 := xm0;
   Result.ym0 := ym0;
end;

// Перевод математических координат в экранные
function toScreen( math : pointR; screen : screenCalc  ) :
pointS ;
var
   res : pointS;
begin
   res.x := round((math.x-screen.xm0)*screen.kx+screen.x0);
   res.y := round((math.y-screen.ym0)*screen.ky+screen.y0);
   toScreen := res;
end;

// Поворот точки на угол (в радианах)
function Rotate( math : pointR; angle : real ) : PointR ;
var
   res : pointR;
begin
   res.x := math.x*cos(angle)-math.y*sin(angle);
   res.y := math.x * sin(angle) + math.y * cos(angle);
   rotate := res;
end;

// Масштабирование относительно центра координат
function Scale( math : pointR; k : real ) : PointR ;
var
   res : pointR;
begin
   res.x := math.x*k;
```

```
   res.y := math.y*k;
   Scale := res;
end;
```

Как видите, ничего сложного в этих функциях нет - все эти формулы вы видели либо тут, либо на уроках геометрии. Функция масштабирования нам понадобится чуть позже.

С хранением и отрисовкой линии чуть сложнее. В общем виде, мы не знаем сколько точек будет в ломаной. Эту задачу мы решим с помощью списка, добавив в раздел type:

```
   PList = ^PointList;
   PointList = record
     point : PointR;
     next  : PList;
   end;
```

И в функции:

```
function addPoint( coord : pointR; next : PList ) : PList;
begin
   new (Result);
   Result^.point := coord;
   Result^.next := next;
end;
```

И, наконец, функция отрисовки линии:

```
procedure Polyline( p : Picture; vertex : PList; screen
:screenCalc );
var
   test : PList;
   p1,p2 : pointS;
begin
  // Построение линии
    p1 := toScreen(vertex^.point,screen );//Первая точка списка
    test := vertex^.next; //Со второй точки
    while test <> nil do // Пройдем по всему списку
    begin
      p2 := p1;
      p1 := toScreen(test^.point,screen );
      p.Line(p2.x,p2.y,p1.x,p1.y);
      test := test^.next;
    end;
    p2 := toScreen(vertex^.point,screen );
    p.Line(p2.x,p2.y,p1.x,p1.y); // Замкнем ломаную
end;
```

Обратите внимание: мы перед собственно рисованием линии рассчитываем ее экранные координаты.

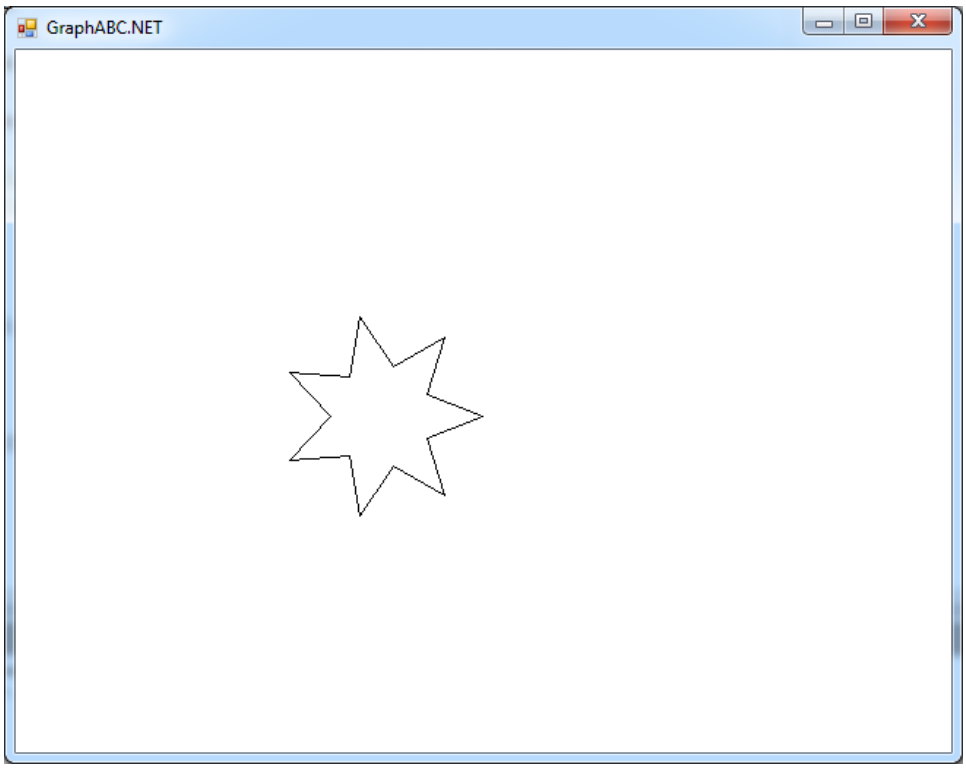
В секцию Const добавим определение количества точек-вершин:

```
const
  StarVertex = 14;
```

И теперь мы можем привести основную программу:

```
var
   star, test : PList;
   i : word;
   start1,start2 : pointR;
   screen : screenCalc;
   p : Picture;
begin

   start1.x := 10;
   start1.y := 0;
   start2.x := 5;
   start2.y := 0;

   star := addPoint(start1,nil);
   star := addPoint(Rotate(start2,2*Pi/StarVertex),star);

  for i:=3 to StarVertex do
   if i mod 2 = 1 then
    star := addPoint(Rotate(start1,(2*Pi/StarVertex)*(i-
1)),star)
   else
    star := addPoint(Rotate(start2,(2*Pi/StarVertex)*(i-
1)),star);

   screen := calcScreen(0,0,500,500,-36,-36,36,36);
   p := new Picture( 500,500);

   PolyLine(p,star,screen);

   p.Draw(0,0);
   while true do
      ;

end.
```

Результат исполнения:

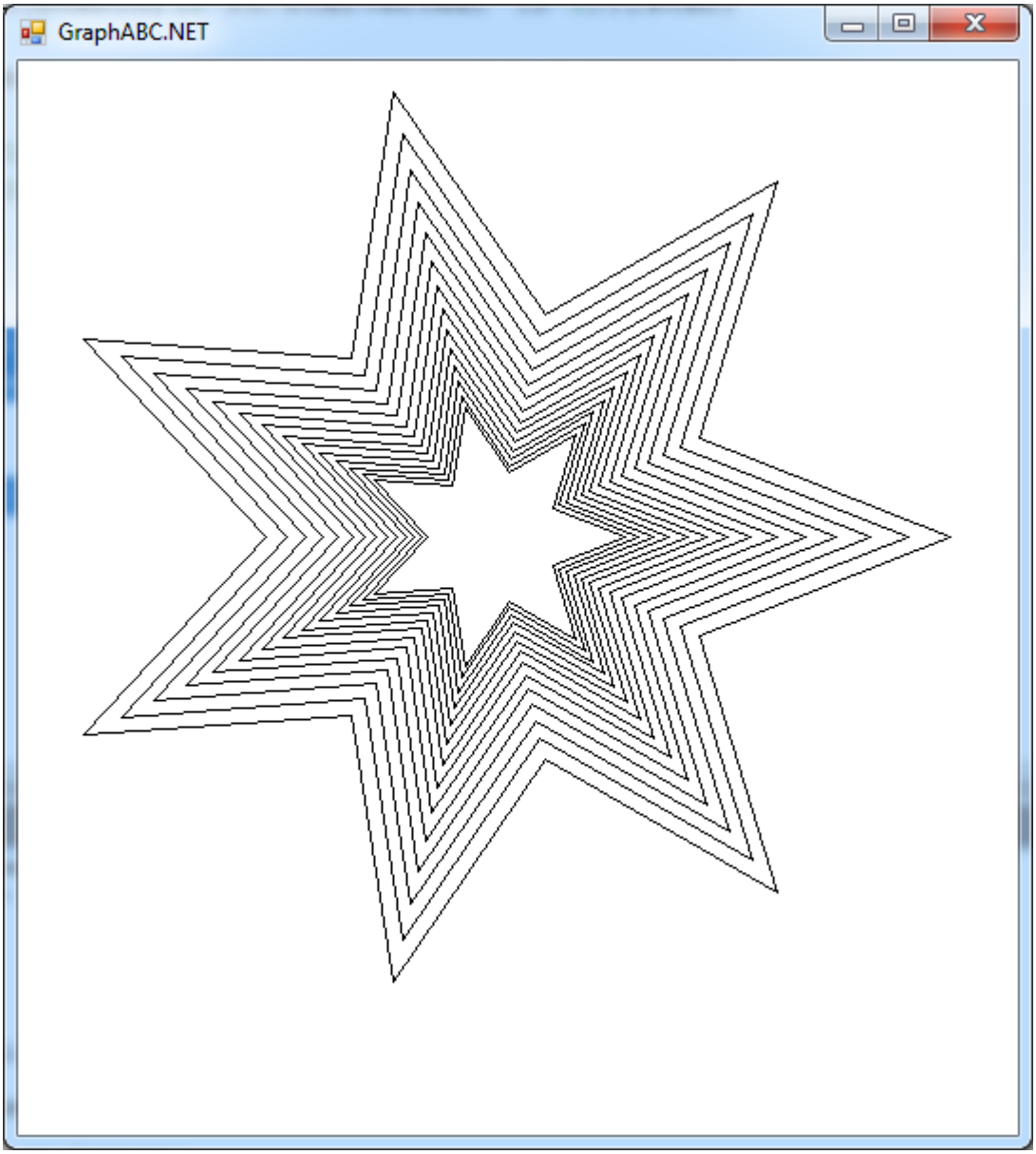


Усложним задачу - построим 14 таких звезд, вложенных друг в друга. Решить такую задачу средствами только растровой графики трудно, а векторной - ничего сложного.

Изменим часть, посвященную собственно рисованию линии - выполним после каждой отрисовки ее масштабирование:

```
for i:=1 to 14 do
   begin
    PolyLine(p,star,screen);
    test := star; // Пройдем по всему списку
    while test <> nil do
    begin
      test^.point := Scale(test^.point,1.1 );
      test := test^.next;
    end;
   end;
```

Результат исполнения:



Как видите, используя этот аппарат мы можем строить сложные изображения, опираясь на простые средства, без потери качества изображения выполняя целый ряд преобразований. Стоит заметить, что возможности эти будут напрямую зависеть от вашей математической подготовки.

Задания.

1. Нарисуйте с помощью описанных средств фигуры-стрелки по кругу, диаметром 10 математических единиц.
2. Реализуйте набор звезд, в котором будет плавно меняться цвет: каждая звезда рисуется одним своим цветом.
3. Реализуйте картинку из 10 вложенных квадратов. Вершины внутреннего квадрата - середины сторон наружного.

Результаты выполнения заданий предъявите учителю для оценивания.