Стерлитамакский филиал
ФГБОУ ВПО «Башкирский Государственный университет»

Интерактивное пособие

Программная реализация компьютерной графики

Авторы-составители:
Хасанова Светлана Леонидовна, к.ф.н., доцент СФ БашГУ,
Рассказова Анна Анатольевна, студент физико-математического факультета СФ БашГУ

Стерлитамак - 2015

Содержание курса

1 модуль

Раздел 1. Основные понятия компьютерной графики.

Раздел 2. Алгоритмы вывода графических примитивов.

2 модуль

Раздел 3. Алгоритмы вывода трехмерных объектов.

Раздел 4. Алгоритмы вывода фрактальной графики.

Раздел 1.
Основные понятия компьютерной графики.

§1. Основные понятия теории цвета.

§2. Основные понятия компьютерной графики.

§1. Основные понятия теории цвета

Цвет — качественная субъективная характеристика электромагнитного излучения оптического диапазона, определяемая на основании возникающего физиологического зрительного ощущения и зависящая от ряда физических, физиологических и психологических факторов. Восприятие цвета определяется индивидуальностью человека, а также спектральным составом, цветовым и яркостным контрастом с окружающими источниками света, а также несветящимися объектами.

Для того чтобы «увидеть» цвет, нужны три вещи:

- источник света;
- объект;
- ваш глаз (приемник излучения).

Так же цвет имеет несколько параметров:

- яркость (или интенсивность) пропорциональна сумме энергий всех составляющих цветового спектра света;
- цветность, связана с доминирующими длинами волн в этом спектре.

Яркость является количественной характеристикой цвета. С ее помощью мы можем сравнивать интенсивность излучения различных источников между собой.

В отличие от нее цветность имеет качественный характер.

Принцип действия большинства технических устройств, предназначенных для обработки, содержащейся в свете цветовой информации, и базируется на раздельном распознавании красной, зеленой и синей составляющих света. Именно поэтому, субъективность в восприятии цвета при обработке изображений крайне нежелательна. Для обеспечения одинакового воспроизведения одного и того же цвета видеомониторами, принтерами и сканерами разных фирм-изготовителей необходимо наличие объективных измерительных систем, позволяющих установить однозначное определение цветовых координат. Для этих целей разработаны специальные средства, включающие:

- цветовые модели;
- системы соответствия цветов;
- цветовые режимы.

Цветовые модели (или цветовые пространства) предоставляют средства для концептуального и количественного описания цвета.

Ознакомившись с основами концептуального представления цвета, вы сможете лучше понять соотношения между цветами при работе, например, с тоновыми кривыми или при выборе нужного цвета с помощью окон диалога или палитр.

Режим — это способ реализации определенной цветовой модели в рамках конкретной графической программы.

Излученный цвет — это свет, испускаемый активным источником. Примерами таких источников могут служить солнце, лампочка или экран монитора.

Отражение — физический процесс взаимодействия волн или частиц с поверхностью.

§2. Основные понятия компьютерной графики

В компьютерной графике с понятием разрешения обычно происходит больше всего путаницы, поскольку приходится иметь дело сразу с несколькими свойствами разных объектов. Следует четко различать: разрешение экрана, разрешение печатающего устройства и разрешение изображения. Все эти понятия относятся к разным объектам. Друг с другом эти виды разрешения никак не связаны пока не потребуется узнать, какой физический размер будет иметь картинка на экране монитора, отпечаток на бумаге или файл на жестком диске.

Разрешение экрана - это свойство компьютерной системы (зависит от монитора и видеокарты) и операционной системы (зависит от настроек ОС Windows). Разрешение экрана измеряется в пикселях (точках) и определяет размер изображения, которое может поместиться на экране целиком.

Разрешение принтера - это свойство принтера, выражающее количество отдельных точек, которые могут быть напечатаны на участке единичной длины. Оно измеряется в единицах dpi (точки на дюйм) и определяет размер изображения при заданном качестве или, наоборот, качество изображения при заданном размере.

Разрешение изображения - это свойство самого изображения. Оно тоже измеряется в точках на дюйм - dpi и задается при создании изображения в графическом редакторе или с помощью сканера. Так, для просмотра изображения на экране достаточно, чтобы оно имело разрешение 72 dpi, а для печати на принтере - не меньше как 300 dpi. Значение разрешения изображения хранится в файле изображения.

Физический размер изображения определяет размер рисунка по вертикали (высота) и горизонтали (ширина) может измеряться как в пикселях, так и в единицах длины (миллиметрах, сантиметрах, дюймах). Он задается при создании изображения и хранится вместе с файлом. Если изображение готовят для демонстрации на экране, то его ширину и высоту задают в пикселях, чтобы знать, какую часть экрана оно занимает. Если изображение готовят для печати, то его размер задают в единицах длины, чтобы знать, какую часть листа бумаги оно займет.

Физический размер и разрешение изображения неразрывно связаны друг с другом. При изменении разрешения автоматически меняется физический размер.

Графические форматы

Любое графическое изображение сохраняется в файле. Способ размещения графических данных при их сохранении в файле определяет графический формат файла. Различают форматы файлов растровых изображений и векторных изображений. Растровые изображения сохраняются в файле в виде прямоугольной таблицы, в каждой клеточке которой записан двоичный код цвета соответствующего пикселя. Такой файл хранит данные и о других свойствах графического изображения, а также алгоритме его сжатия.

Векторные изображения сохраняются в файле как перечень объектов и значений их свойств - координат, размеров, цветов и тому подобное.

Как растровых, так и векторных форматов графических файлов существует достаточно большое количество. Выбор того или другого формата для сохранения изображения зависит от целей и задач работы с изображением. Если нужна фотографическая точность воссоздания цветов, то преимущество отдают одному из растровых форматов. Логотипы, схемы, элементы оформления целесообразно хранить в векторных форматах. Формат файла влияет на объем памяти, который занимает этот файл. Графические редакторы позволяют пользователю самостоятельно избирать формат сохранения изображения. Если вы собираетесь работать с графическим изображением только в одном редакторе, целесообразно выбрать тот формат, какой редактор предлагает по умолчанию. Если же данные будут обрабатываться другими программами, стоит использовать один из универсальных форматов.

Существуют универсальные форматы графических файлов, которые одновременно поддерживают и векторные, и растровые изображения.

- Формат PDF (англ. Portable Document Format - портативный формат документа) разработан для работы с пакетом программ Acrobat. В этом формате могут быть сохранены изображения и векторного, и растрового формата, текст с большим количеством шрифтов, гипертекстовые ссылки и даже настройки печатающего устройства. Размеры файлов достаточно малы. Он позволяет только просмотр файлов, редактирование изображений в этом формате невозможно.

- Формат EPS (англ. Encapsulated PostScript - инкапсулированный постскриптум) - формат, который поддерживается программами для разных операционных систем. Рекомендуется для печати и создания иллюстраций в настольных издательских системах. Этот формат позволяет сохранить векторный контур, который будет ограничивать растровое изображение.

Форматы файлов растровой графики

Существует несколько десятков форматов файлов растровых изображений. У каждого из них есть свои позитивные качества, которые определяют целесообразность его использования при работе с теми или другими программами. Рассмотрим самые распространенные из них.

Достаточно распространенным является формат Bitmap (англ. Bit map image - битовая карта изображения). Файлы этого формата имеют расширение .BMP. Данный формат поддерживается практически всеми графическими редакторами растровой графики. Основным недостатком формата BMP является большой размер файлов из-за отсутствия их сжатия.

Для хранения многоцветных изображений используют формат JPEG (англ. Joint Photographic Expert Group - объединенная экспертная группа в отрасли фотографии), файлы которого имеют расширение.JPG или .JPEG. Позволяет сжать изображение с большим коэффициентом (до 500 раз) за счет необратимой потери части данных, что значительно ухудшает качества изображения. Чем меньше цветов имеет изображение, тем хуже эффект от использования формата JPEG, но для цветных фотографии на экране это малозаметно.

Формат GIF (англ. Graphics Interchange Format - графический формат для обмена) - самый уплотнённый из графических форматов, что не имеет потери данных и позволяет уменьшить размер файла в несколько раз. Файлы этого формата имеют расширение .GIF. В этом формате сохраняются и передаются малоцветные изображения (до 256 оттенков), например, рисованные иллюстрации. У формата GIF есть интересные особенности, которые позволяют сохранить прозрачность фона и анимацию изображения. GIF-формат также позволяет записывать изображение "через строку", благодаря чему, имея только часть файла, можно увидеть изображение полностью, но с меньшей разрешающей способностью.

Графический формат PNG (англ. Portable Network Graphic - мобильная сетевая графика) - формат графических файлов, аналогичный формату GIF, но который поддерживает намного больше цветов.

Для документов, которые передаются по сети Интернет, очень важным является незначительный размер файлов, поскольку от него зависит скорость доступа к информации. Поэтому при подготовке Web-страниц используют типы графических форматов, которые имеют высокий коэффициент сжатия данных: .JPEG, .GIF, .PNG.

Особенно высокие требования к качествам изображений предъявляются в полиграфии. В этой отрасли применяется специальный формат TIFF (англ. Tagged Image File Format - теговый (с пометками) формат файлов изображений). Файлы этого формата имеют расширение .TIF или .TIFF. Они обеспечивают сжатие с достаточным коэффициентом и возможность хранить в файле дополнительные данные, которые на рисунке расположены во вспомогательных слоях и содержат аннотации и примечания к рисунку.

Формат PSD (англ. PhotoShop Document). Файлы этого формата имеют расширение .PSD. Это формат программы Photoshop, который позволяет записывать растровое изображение со многими слоями, дополнительными цветовыми каналами, масками, т.е. этот формат может сохранить всё, что создал пользователь видимое на мониторе.

Форматы файлов векторной графики

Форматов файлов векторной графики существует намного меньше. Приведем примеры самых распространенных из них.

- WMF (англ. Windows MetaFile - метафайл Windows) - универсальный формат для Windows-дополнений. Используется для хранения коллекции графических изображений Microsoft Clip Gallery. Основные недостатки - искажение цвета, невозможность сохранения ряда дополнительных параметров объектов.
- CGM (англ. Computer Graphic Metafile - метафайл компьютерной графики) - широко использует стандартный формат векторных графических данных в сети Internet.
- CDR (англ. CorelDRaw files - файлы CorelDRaw) - формат, который используется в векторном графическом редакторе Corel Draw.
- AI - формат, который поддерживается векторным редактором Adobe Illustrator.

Раздел 2.
Алгоритмы вывода графических примитивов

§1. Алгоритмы вывода прямой линии.

§2. Алгоритм вывода окружности.

§3. Стиль линии. Перо.

§4. Стиль заполнения. Кисть. Текстура.

Примеры выполнения алгоритмов.

Лабораторные задания.

§1. Алгоритмы вывода прямой линии

Простейшим и, вместе с тем, наиболее универсальным растровым графическим примитивом является пиксель. Любое растровое изображение можно нарисовать по пикселям, но это сложно и долго. Необходимы больше сложные элементы, для которых рисуются сразу несколько пикселей.

Рассмотрим графические примитивы, которые используются наиболее часто в современных графических системах, — это линии и фигуры.

1.1. Алгоритмы вывода прямой линии

Рассмотрим растровые алгоритмы для отрезков прямой линии. Предположим, что заданы координаты (x1,y1 - x2,y2) концов отрезка прямой. Для вывода линии необходимо закрасить в определенный цвет все пиксели вдоль линии. Для того чтобы закрасить каждый пиксель, необходимо знать его координаты.

Наиболее просто нарисовать отрезок горизонтальной линии:

```
for (x=x1; x<=x2; x++)
//Пиксель (x, y1);
```

Вычисление текущих координат пикселя здесь выполняется как приращение по X (необходимо, чтобы x1 <=x2), а вывод пикселя обеспечивается функцией Пиксель().

Аналогично рисуется отрезок вертикали:

```
for  (y=y1;  y<=y2;  y++)
//Пиксель (x1, y);
```

Как видим, в цикле выполняются простейшие операции над целыми числами — приращение на единицу и проверка на "< =". Поэтому операция рисования отрезка выполняется быстро и просто. Ее используют как базовую операцию для других операций, например, в алгоритмах заполнения плоскости полигонов.

Можно поставить такой вопрос: какая линия рисуется быстрее — горизонталь или вертикаль? На первый взгляд — одинаково быстро. Если учитывать только математические аспекты, то скорость должна быть одной и той же при одинаковой длине линий, поскольку в обоих случаях выполняется равное количество идентичных операций.

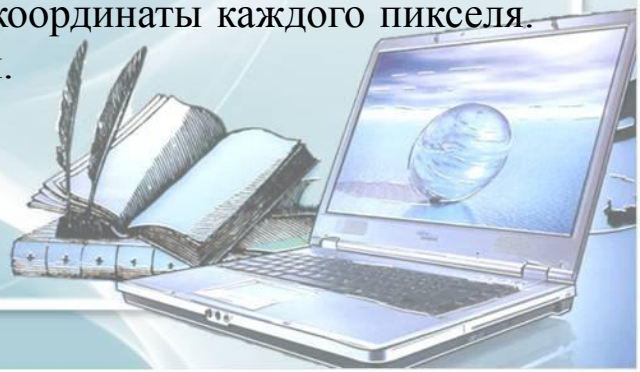
Однако если кроме расчета координат анализировать также вывод пикселей, то могут быть отличия. В растровых системах рисование пикселя обычно означает запись одного или нескольких бит в память, где сохраняется растр. И здесь уже не все равно — по строкам или по столбцам заполняется растр.

Необходимо учитывать логическую организацию памяти компьютера, в которой хранятся биты или байты растра. Даже для компьютеров одного типа (например, персональных компьютеров) для различных поколений процессоров и памяти скорость записи по соседним адресам может существенно отличаться от скорости записи по не соседним адресам.

В особенности это заметно, когда для растра используется виртуальная память с сохранением отдельных страниц на диске и (или) в оперативной памяти (RAM). При работе графических программ в среде операционной системы Windows часто случается так, что горизонтали рисуются быстрее вертикалей, поскольку в страницах памяти хранятся соседние байты. А может быть, что RAM достаточно, но скорости рисования все же различны.

Например, если используется черно-белый растр в формате один бит на пиксель, то для вертикали битовая маска одинакова для всех пикселей линии, а для горизонтали маску нужно изменять на каждом шаге. Здесь необходимо заметить, что рисование черно-белых горизонталей можно существенно ускорить, если записывать сразу восемь соседних пикселей— байт в памяти.

Горизонтали и вертикали представляют собой частный случай линий. Рассмотрим линию общего вида. Для нее также необходимо вычислять координаты каждого пикселя. Известно несколько методов расчетов координат точек линии.



1.1.1.Прямое вычисление координат

Пусть заданы координаты конечных точек отрезка прямой. Найдем координаты точки внутри отрезка (см рис 1)

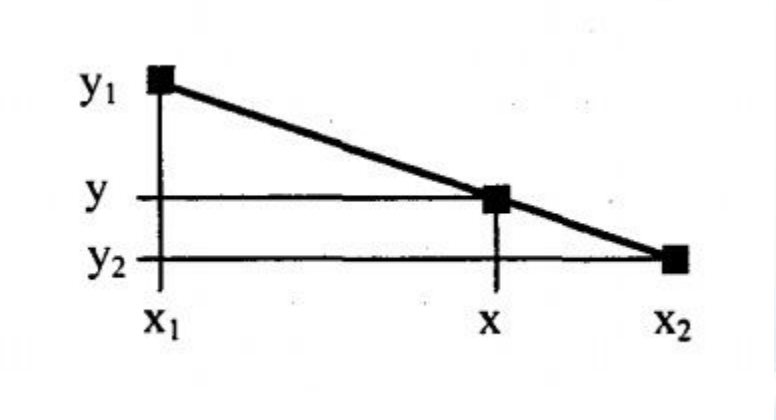


Рис.1. Отрезок прямой.

Запишем соотношения катетов для подобных прямоугольных треугольников:

$$\frac{x - x_1}{y - y_1} = \frac{x_2 - x_1}{y_2 - y_1},$$

$$x = x_1 + (y - y_1)\frac{x_2 - x_1}{y_2 - y_1},$$

то есть $x = f(y)$

А также

$$y = F(x): y = y_1 + (x - x_1)\frac{y_2 - y_1}{x_2 - x_1}.$$

то есть $y = F(x)$

В зависимости от угла наклона прямой выполняется цикл по оси x или по y (см рис 2)

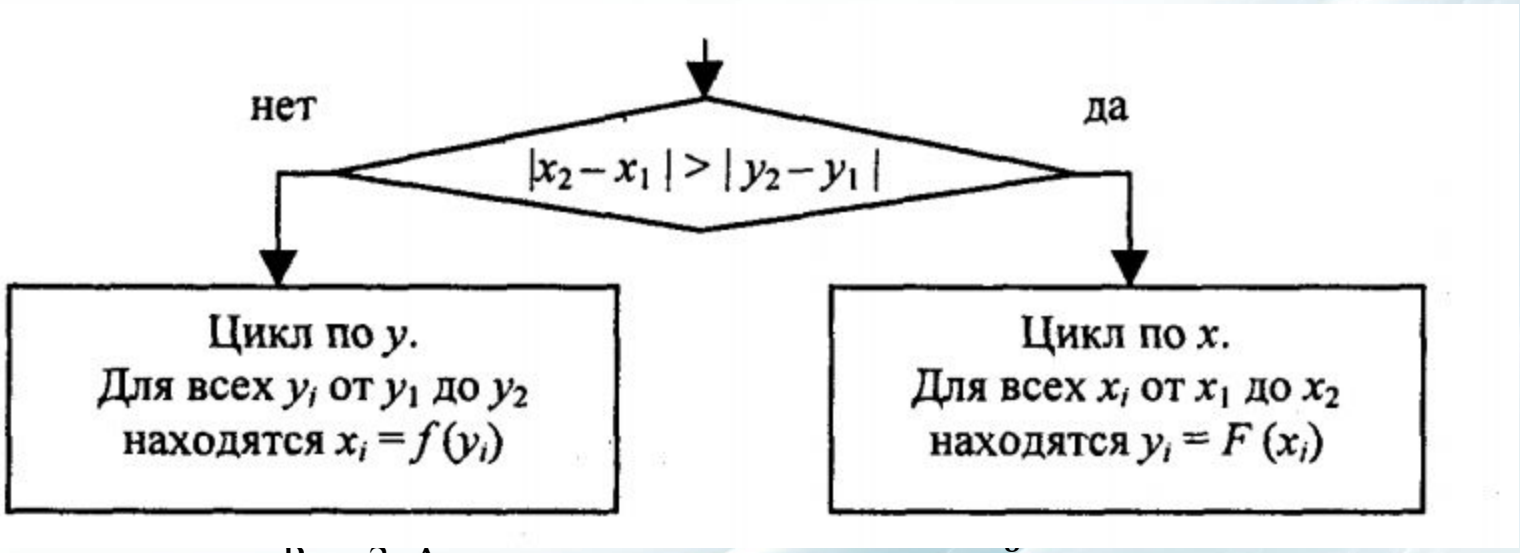


Рис. 2. Алгоритм вывода отрезка прямой линии

Приведем пример записи этого алгоритма на компьютерном языке программирования C,C++. Для сокращения текста рассмотрим фрагмент программы, где выполняется цикл по оси *x*, причем $x_1<x_2$:

```
for (x=x1; x<=x2; x++)
 {
  y = y1 + ((x-x1)*(y2-y1))/(x2-x1);
 //Пиксель(x, y);
 }
```

Здесь все операции выполняются над целыми числами. Двойные скобки не обходимы для того, чтобы деление выполнялось после умножения. Недостатки такой программы — в цикле выполняется много лишних операций, присутствуют операции деления и умножения. Это обуславливает малую скорость работы. Относительно лишних операций в цикле. Можно вынести вычисление $(y_2 - y_1)/(x_2 - x_1)$ из цикла, поскольку это значение не изменяется. Однако для этого уже необходимо использовать операции с плавающей точкой:

```
float k;
 k = (float)(y2-y1)/(float)(x2-x1);
 for (x=x1; x<=x2; x++)
 {
  y = y1 + (float)(x-x1)*k;
 //Пиксель(x, y);
 }
```

Попробуем еще уменьшить количество операций в цикле. Если раскрыть скобки в выражении y = y1+(x-x1)*k, то получим y = y1 + x*k-x1*k. Здесь значение (y=y1-x1*k) является константой - эти операции также вынесем из тела цикла.

```
float yy, k;
  k = (float) (y2-y1) / (float) (x2-x1);
  yy = (float) y1 - (float) x1*k;
  for (x=x1; x<=x2; x++)
  {
    y = yy + (float) x *k;
    // Пиксель (x, y);
  }
```

В цикле выполняются только две арифметические операции и преобразования x из целого в форму с плавающей точкой.

Если рассматривать цикл вычисления y_i по соответствующим значениям x_i=x1, x1+1,…,x2 как итеративный процесс, то можно поставить такой вопрос: чему равна разность (yi+1-yi)? Она равна yi+1-yi = x1+(xi+1-x1)k-x1-(xi -x1)k = (xi+1-xi)k = k; поскольку (xi+1-xi)=1. Разность (yi+1-yi) - константа, равная k. Исходя из этого, можно построить цикл таким образом:

```
float k;
k = (float)(y2-y1)/(float)(x2-x1);
y=y1;
for (x=x1;x<=x2;x++)
{
  //Пиксель (x, y);
  y+=k;
}
```

В теле цикла есть только одна операция для вычисления координаты y (если не учитывать <= и ++).

Достоинства прямого вычисления координат:

- Простота, ясность построения алгоритма.
- Возможность работы с нецелыми значениями координат отрезка.

Недостатки прямого вычисления координат :

- Использование операций с плавающей точкой или целочисленных операций умножения и деления обуславливает малую скорость. Однако это зависит от процессора и для различных типов компьютеров может быть по- разному. В современных компьютерах, в которых процессоры используют эффективные способы ускорения (например, конвейер арифметических операций с плавающей точкой), время выполнения целочисленных операций уже не намного меньше. Для старых компьютеров разница могла быть в десятки раз, поэтому и старались разрабатывать алгоритмы только на основе целочисленных операций.
- При вычислении координат путем добавления приращений может накапливаться ошибка вычислений координат.

1.1.2.Инкрементные алгоритмы

Брезенхэм предложил подход, позволяющий разрабатывать так называемые инкрементные алгоритмы растеризации. Основной целью при разработке таких алгоритмов было построение циклов вычисления координат на основе только целочисленных операций сложения/вычитания без использования умножения и деления. Были разработаны инкрементные алгоритмы не только для прямых, но и для кривых линий.

Инкрементные алгоритмы выполняются как последовательное вычисление координат соседних пикселов путем добавления приращений координат. Приращения рассчитываются на основе анализа функции погрешности.

В цикле выполняются только целочисленные операции сравнения и сложения/вычитания. Достигается повышение быстродействия для вычислений каждого пиксела по сравнению с прямым способом.

Один из вариантов алгоритма Брезенхэма:

xerr = 0, yerr =0;
dx = x2 - x1, dy = y2 - y1;
Если dx > 0, то incX = 1;
dx = 0, то incX = 0;
dx < 0, то incX = -1;
Если dy > 0, то incY = 1;
dy = 0, то incY = 0;
dy < 0, то incY = -1;
dx = |dx|, dy = |dy|;
Если dx > dy, то d = dx;
иначе d = dy;
x = x1, y = y1;
//Пиксель (x, y)
Выполнить цикл d раз:
{
xerr = xerr + dx;
yerr = yerr + dy;
Если xerr > d, то xerr = xerr - d
x = x + incX
Если yerr > d, то yerr = yerr - d
y = y + incY
//Пиксель (x, y)
}

Рассмотрим пример работы приведенного выше алгоритма Брезенхэма для отрезка (x1y1 - x2y2) = (2,3 - 8,6). Этот алгоритм восьмисвязный, то есть при вычислении приращений координат для перехода к соседнему пикселю возможны восемь случаев.

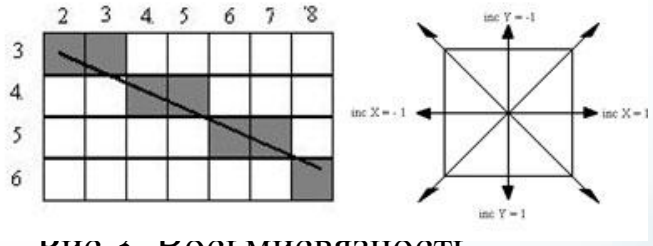


Рис.3. Восьмисвязность

Известны также четырехсвязные алгоритмы (рисунок 4).

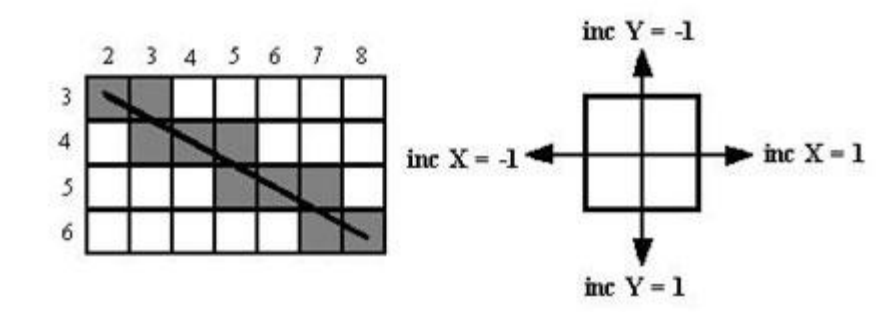


Рис.4. Четырехсвязность

Четырехсвязные алгоритмы проще, но они генерируют менее качественное изображение линий за большее количество тактов работы. Для приведенного примера четырехсвязный алгоритм работает 10 тактов, а восьмисвязный - только 7.

1.2. Алгоритм вывода окружности.

Для вывода контура круга можно использовать соотношение между координатами X и Y для точек круга $X^2 + Y^2 = R^2$ и построить алгоритм прямого вычисления координат. Однако тогда необходимо вычислять квадратный корень, а это в цифровом компьютере выполняется медленно.

```
void Circle (int xc, int yc, int raius)
{
  int x,y,dxt;
 long r2,dst,t,s,e,ca,cd,indx;
  r2=(long)radius*(long)radius;
   dst=4*r2;
  dxt=(double)radius/1.414213562373;
  t=0;
 s=-4*r2*(long)radius;
 e=(-s/2)-3*r2;    ca=-6*r2;    cd=-10*r2;
    x=0;
 y=radius;
 //Пиксель(xc,yc+radius);
 //Пиксель(xc,yc-radius);
//Пиксель(xc+radius,yc);
//Пиксель(xc-radius,yc);
```

```
 for (indx=1; indx<=dxt; indx++)
{
  x++;
  if (e>=0) e+=t+ca;
  else
  {
    y- - ;
    e+=t- s+cd;
    s+=dst;
  }
  t - = dst;
//Пиксель(xc+x,yc+y);
//Пиксель(xc+y,yc+x);
//Пиксель(xc+y,yc-x);
//Пиксель(xc+x,yc-y);
//Пиксель(xc-x,yc-y);
//Пиксель(xc-y,yc-x);
//Пиксель(xc-y,yc+x);
//Пиксель(xc-x,yc+y);
 }}
```

1.3. Алгоритм вывода эллипса.

Инкрементный алгоритм для эллипса подобен алгоритму для круга, но более сложный.

```
void Ellipse(int  xc,int  yc,int  enx,int eny)
{
int x,y;
long a2,b2,dds,ddt,dxt,t,s,e,ca,cd,indx;
int a,b;
a = abs(enx - xc);  b = abs(eny - yc);
a2 =   (long)a*(long)a;
b2 =   (long)b*(long)b;
dds = 4*a2;
ddt = 4*b2;
dxt =   (float)a2/sqrt(a2+b2);
t = 0;
s = -4*a2*b;
e =   (-s/2)   -  2*b2  -  a2;
ca = -6*b2;
cd = ca - 4*a2;
x = xc;
y =  yc  + b;
```

```
//Пиксель (x,y);
//Пиксель (x,2*yc-y);
 //Пиксель (2*xc-x,2*yc-y;
//Пиксель (2*xc-x,y);
 for(indx=1; indx<=dxt; indx++)
 {
   x++;
   if(e>=0)e+=t+ca;
   else
   {
     y--;
     e+=t-s+cd;
     s+=dds;
   }
   t-=ddt;
  //Пиксель (x,y);
 //Пиксель (x,2*yc-y);
 //Пиксель (2*xc-x,2*yc-y);
 //Пиксель (2*xc-x,y);
 }
 dxt=abs(y-yc);
```

```
  e-=t/2+s/2+b2+a2;
  ca= -6*a2;
  cd=ca-4*b2;
for (indx=1; indx<=dxt; indx++)
  {
    y--;
    if(e<=0) e+=-s+ca;
    else
    {
      x++;
      e+=-s+t+cd;
      t-=ddt;
    }
    s+=dds;
     //Пиксель ( (x,y);
     //Пиксель ((x,2*yc-y);
    //Пиксель ( (2*xc-x,2*yc-y);
    //Пиксель ((2*xc-x,y);
  }
}
```

В этом алгоритме использована симметрия эллипса по квадрантам (рисунок 5). Алгоритм состоит из двух циклов. Сначала от x = 0 до x = dxt, где

$$dxt = \frac{a^2}{\sqrt{a^2 + b^2}},$$

а потом цикл до точки x = a, y = 0.

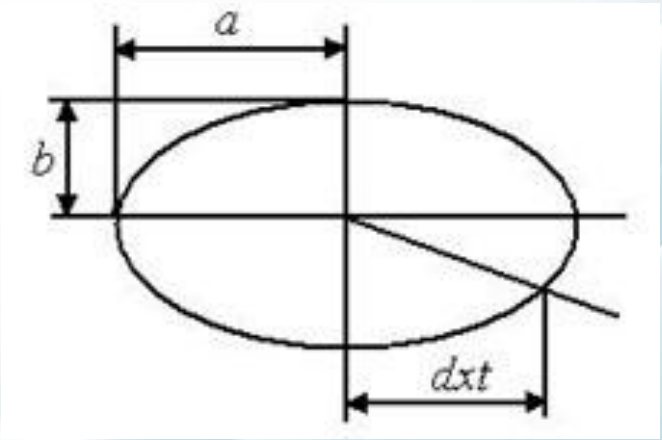


Рис. 5. Вывод эллипса

1.4. Кривая Безье

Разработана математиком Пьером Безье. Кривые и поверхности Безье были использованы в 60-х годах компанией "Рено" для компьютерного проектирования формы кузовов автомобилей. В настоящее время они широко используются в компьютерной графике.

Кривые Безье описываются в параметрической форме:

$$x = P_x(t),$$
$$y = P_y(t).$$

Значение t выступает как параметр, которому отвечают координаты отдельной точки линии. Параметрическая форма описания может быть более удобной для некоторых кривых, чем задание в виде функции $y = f(x)$. Это потому, что функция $f(x)$ может быть намного сложнее, чем $P_x(t)$ и $y = P_y(t)$, кроме того, $f(x)$ может быть неоднозначной.

Многочлены Безье для P_x и P_y имеют такой вид:

$$P_x(t) = \sum_{i=0}^{m} C_m^i t^i (1-t)^{m-i} x_i$$

$$P_y(t) = \sum_{i=0}^{m} C_m^i t^i (1-t)^{m-i} y_i$$

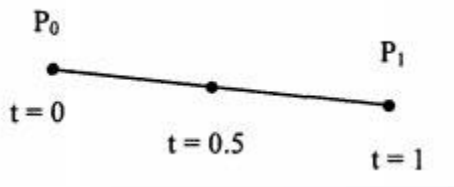
где $C_m^{\ i}$ - сочетание m по i (известное также по биному Ньютона), $C_m^{\ i} = m!/(i!(m-i)!)$, а x_i и y_i - координаты точек-ориентиров P_i.

Значение m можно рассматривать и как степень полинома, и как значение, которое на единицу меньше количества точек-ориентиров.

Рассмотрим кривые Безье, классифицируя их по значениям *m*.

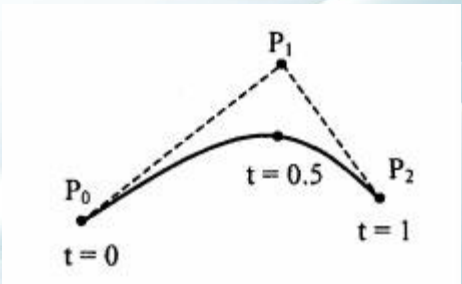
m = 1 (по двум точкам).

Кривая вырождается в отрезок прямой линии, определяемый концевыми точками P_0 и P_1 как показано на рисунке.



m = 2 (по трем точкам).

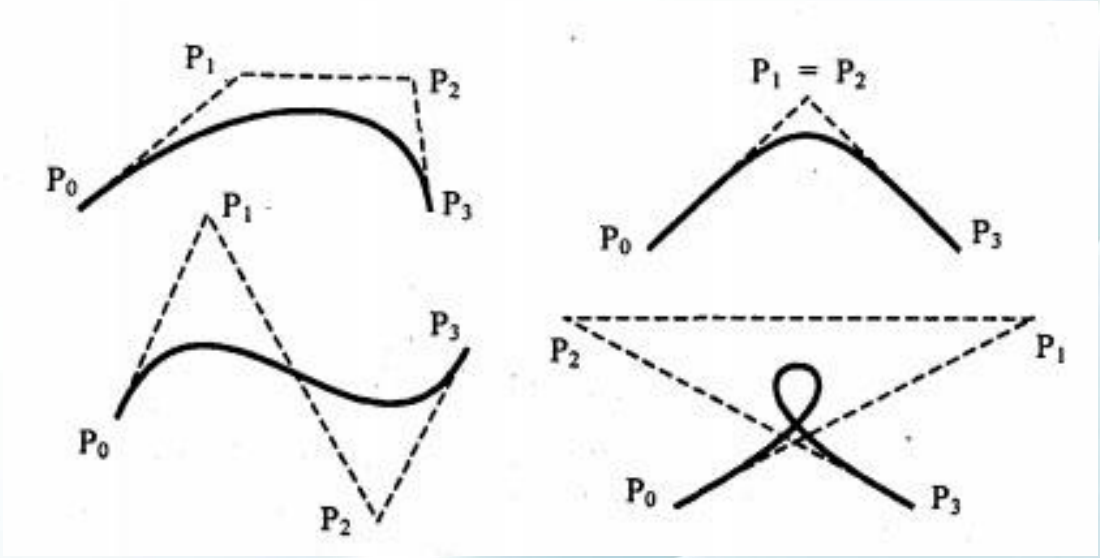
$$P(t) = (1 - t)^2 P_0 + 2t(1 - t) P_1 + t^2 P_2$$



m = 3 (по четырем точкам, кубическая).

Используется довольно часто, в особенности в сплайновых кривых.

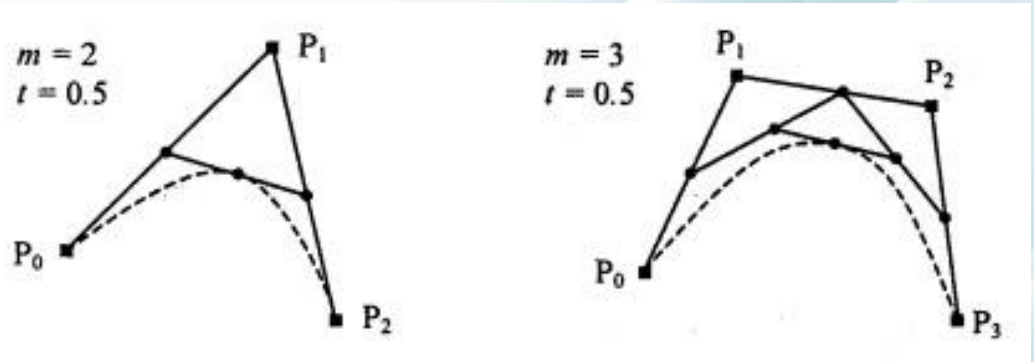
$$P(t) = (1 - t)^3 P_0 + 3t(1 - t)^2 P_1 + 3t(1 - t)^2 P_2 + t^3 P_3.$$



Геометрический алгоритм для кривой Безье

Этот алгоритм позволяет вычислить координаты *(x,y)* точки кривой Безье по значению параметра *t*.

1. Каждая сторона контура многоугольника, проходящего по точкам-ориентирам, делится пропорционально значению *t*.
2. Точки деления соединяются отрезками прямых и образуют новый многоугольник. Количество узлов нового контура на единицу меньше, чем количество узлов предыдущего контура.
3. Стороны нового контура снова делятся пропорционально значению *t*. И так далее. Это продолжается до тех пор, пока не будет получена единственная точка деления. Эта точка и будет точкой кривой Безье.



Приведем запись геометрического алгоритма на языке C++:

```
for (i=0; i<=m; i++)

R[i]=P[i];                    // Формируем вспомогательный массив R

for (j=m; j>0; j--)

for (i=0; i<j; i++)

R[i]=R[i]+t*(R[i+1]-R[i]);
```

Результат работы алгоритма – координаты одной точки кривой Безье – записываются в R[0]

1.5. Алгоритмы вывода фигур

Фигурой здесь будем считать плоский геометрический объект, который состоит из линий контура и точек заполнения, которые помещаются внутри контура. Контуров может быть несколько — например, если объект имеет внутри пустоты (рис. 6). В некоторых графических системах одним объектом может считаться и более сложная многоконтурная фигура - совокупность островов с пустотами.

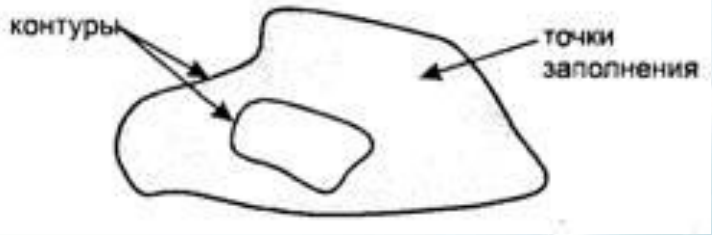
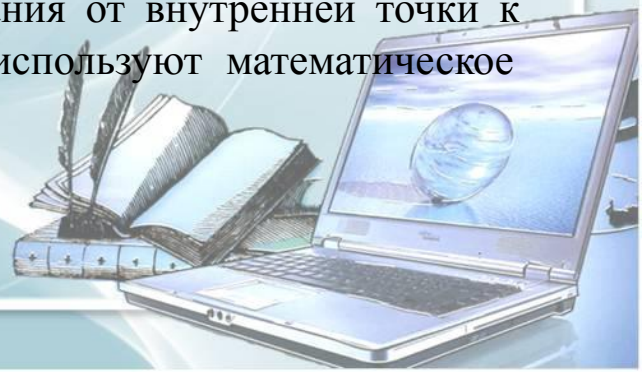


Рис.6. Пример фигуры

Графический вывод фигур делится на две задачи: вывод контура и вывод точек заполнения. Поскольку контур представляет собой линию, то вывод контура проводится на основе алгоритмов вывода линий. В зависимости от сложности контура, это могут быть отрезки прямых, кривых или произвольная последовательность соседних пикселей.

Для вывода точек заполнения известны методы, которые разделяются в зависимости от использования контура на два типа: алгоритмы закрашивания от внутренней точки к границам произвольного контура и алгоритмы, которые используют математическое описание контура.



1.5.1. Алгоритмы закрашивания

Рассмотрим алгоритмы закрашивания произвольного контура, который уже нарисован в растре. Сначала определяются координаты произвольного пиксела, находящегося внутри очерченного контура фигуры. Цвет этого пиксела изменяем на нужный цвет заполнения. Потом проводится анализ цветов всех соседних пикселов. Если цвет некоторого соседнего пиксела не равен цвету границы контура или цвету заполнения, то цвет этого пиксела изменяется на цвет заполнения. Потом анализируется цвет пикселов, соседних с предшествующими. И так далее, пока внутри контура все пикселы не перекрасятся в цвет заполнения.

Пикселы контура образуют границу, за которую нельзя выходить в ходе последовательного перебора всех соседних пикселов. Соседними могут считаться только четыре пиксела (сосед справа, слева, сверху и снизу — четырехсвязность), или восемь пикселов (восьми-связность). Не всякий контур может считаться границей закрашивания, например, для восьмисвязного алгоритма (рис. 7).

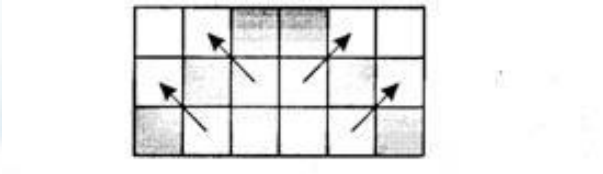


Рис. 7. Выход за границу контура на следующих шагах закрашивания

Простейший алгоритм закрашивания

Для всех алгоритмов закрашивания нужно задавать начальную точку внутри контура с координатами x_o, y_o.

Простейший алгоритм можно описать так:

Шаг 1. Определить x_o, y_o.

Шаг 2. Выполнить функцию ЗАКРАШИВАНИЕ(x_o, y_o)

Функцию ЗАКРАШИВАНИЕ() определим так:

```
Функция ЗАКРАШИВАНИЕ (x, y)
{
  Если цвет пикселя (x, y) не равен цвету границы, то
    { установить для пикселя (x, y) цвет заполнения
    ЗАКРАШИВАНИЕ (x+1, y);
    ЗАКРАШИВАНИЕ (x-1, y);
    ЗАКРАШИВАНИЕ (x, y+1);
    ЗАКРАШИВАНИЕ (x, y-1);
    }
}
```

Такое определение функции является рекурсивным. Рекурсия позволяет упростить запись некоторых алгоритмов. Но для этого алгоритма рекурсия порождает существенные проблемы - рекурсивные вызовы функции ЗАКРАШИВАНИЕ() делаются для каждого пикселя, что обычно приводит к переполнению стека в ходе выполнения компьютерной программы.

Практика показывает, что этот алгоритм неприменим для фигур площадью в тысячу и больше пикселей.

Можно построить подобный алгоритм и без рекурсии, если вместо стека компьютера использовать отдельные массивы. Тогда стек не переполняется.

Волновой алгоритм закрашивания

Алгоритм предназначался для расчета центра тяжести объектов по соответствующим изображениям. Идея была навеяна волновым алгоритмом поиска трассы на графе, известным в САПР электронных схем. Суть подобных алгоритмов состоит в том, что для начальной точки (вершины на графе) находятся соседние точки (другие вершины), которые отвечают двум условиям: во-первых - эти вершины связаны с начальной; во-вторых - эти вершины еще не отмечены, то есть они рассматриваются впервые. Соседние вершины текущей итерации отмечаются в массиве описания вершин, и каждая из них становится текущей точкой для поиска новых соседних вершин в следующей итерации. Если в специальном массиве отмечать каждую вершину номером итерации, то когда будет достигнута конечная точка, можно совершить обратный цикл - от конечной точки к начальной по убыванию номеров итераций. В ходе обратного цикла и находятся все кратчайшие пути (если их несколько) между двумя заданными точками на графе. Подобный алгоритм можно также использовать, например, для поиска всех нужных файлов на диске. Относительно закрашивания растровых фигур, то здесь вершинами графа являются пиксели, а отметка пройденных пикселей делается прямо в растре цветом закрашивания. Как видим, это почти полностью повторяет идею предыдущего простейшего алгоритма, однако здесь мы не будем использовать рекурсию. Это обуславливает совсем другую последовательность обработки пикселей при закрашивании.

Запишем волновой алгоритм закрашивания на языке C++ с использованием графических функций API Windows.

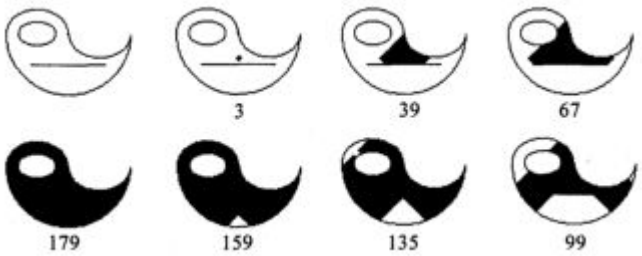
```
void WaveFill(HDC hdc,int xst, int yst)
{
 int numA,numB;
 POINT *stackA, *stackB;
 stackA=new POINT[10000];          // Открываем массивы
 stackB=new POINT[10000];          // для текущих фронтов волн
 numA=1;
 stackA[0].x=xst;                  // В массив stackA записываем
 stackA[0].y=yst;                  // координаты стартовой точки
 numB=0;                           // Массив stackB пока что пуст
 while (1)                         // Основной цикл
  {
     if (numA>0) OneStep(hdc,&numA,&numB,stackA,stackB);
     else break;
     if (numB>0) OneStep(hdc,&numB,&numA,stackB,stackA);
     else break;
  }
 delete[] stackB;
 delete[] stackA;
}
```

```
// ---------- Одна итерация (фронт) распространения волны  ----------
// Из массива Src[] читаются координаты пройденных точек
// для каждой точки находится соседняя и записывается
// в массив Dest[] - в этом массиве будет новый фронт

void OneStep(HDC hdc, int *numSrc,
int *numDest, POINT *Src, POINT* Dest)
{
 int x,y,i;
 *numDest=0;
 for(i=0;i<*numSrc;i++)
{ x=Src[i].x;
  y=Src[i].y;
                NearPix(hdc,x+1,y,numDest,Dest);
NearPix(hdc,x-1,y,numDest,Dest);
                NearPix(hdc,x,y+1,numDest,Dest);
NearPix(hdc,x,y-1,numDest,Dest);
 }
}
void NearPix(HDC hdc, int x, int y, int
*numStack, POINT *Stack)
{
   if (GetPixel(hdc,x,y)!=0)
 {
  SetPixel (hdc,x,y,0); // Пиксель закрашивания
  for (int e=0; e<1000; e++)
        for (int t=0; t<1000;t++)

  Stack[*numStack].x=x; // Координаты в массив
  Stack[*numStack].y=y;
    (*numStack)++; // Количество элементов в
массиве
  }
 }
```

Здесь цвет закрашивания и цвет контура - черный цвет (код 0). Пример работы алгоритма приведен на рисунке.



От начальной точки распространяется волна пикселей закрашивания в виде ромба. В одном цикле OneStep закрашиваются пиксели вдоль линии периметра ромба (или нескольких ромбов в зависимости от сложности фигуры). В качестве рабочих массивов для текущего сохранения координат пикселей фронтов волн использованы динамические массивы емкостью по 10 000 элементов. Максимальная емкость массивов обуславливается размерами контура и рассчитывается эмпирически.

Достаточно просто модифицировать приведенный алгоритм для случая отличающихся цветов контура и заполнения. Необходимо заметить, что этот алгоритм не является самым быстрым из известных алгоритмов закрашивания, особенно если для его реализации использовать медленную функцию SetPixel для рисования отдельных пикселей в программах для Windows. Большую скорость закрашивания обеспечивают алгоритмы, которые обрабатывают не отдельные пиксели, а сразу большие блоки из многих пикселей, которые образовывают прямоугольники или линии.

Алгоритм закрашивания линиями

Данный алгоритм получил широкое распространение в компьютерной графике. От приведенного ранее простейшего рекурсивного алгоритма он отличается тем, что на каждом шаге закрашивания рисуется горизонтальная линия, которая размещается между пикселями контура.

Алгоритм рекурсивный, но поскольку вызов функции осуществляется для линии, а не для каждого отдельного пикселя, то количество вложенных вызовов уменьшается пропорционально длине линии. Это уменьшает нагрузку на стековую память компьютера.

Приведем запись алгоритма на языке C++.

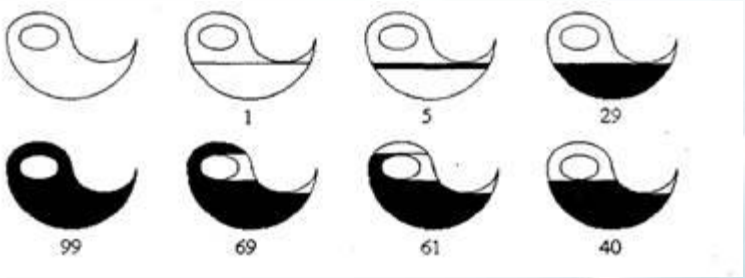
```
int LineFill (HDC hdc,int x,int y,int dir,int preXL,int preXR)
{
 int xl=x, xr=x;
 COLORREF clr,BORDER=RGB(0,0,0);
 do {
   clr=GetPixel(hdc,--xl,y);
    }
 while(clr!=BORDER); //BORDER - цвет контура
```

```
do {
  clr=GetPixel(hdc,++xr,y); }
  while(clr!=BORDER);
  xl++; xr--;                          // Левая и правая границы текущей горизонтали
  MoveToEx(hdc,xl,y,NULL);
  LineTo(hdc,xr+1,y);
  for(x=xl;x<=xr;x++)
  {
    clr=GetPixel(hdc,x,y+dir);
    if (clr!=BORDER)
    {
      x=LineFill(hdc,x,y+dir,dir,xl,xr);
    }
  }
  for(x=xl;x<preXL;x++)
  {
    clr=GetPixel(hdc,x,y-dir);
    if (clr!=BORDER)
      x=LineFill(hdc,x,y-dir,-dir,xl,xr);
  }
  for(x=preXR;x<xr;x++)
  {
    clr=GetPixel(hdc,x,y-dir);
    if (clr!=BORDER)
      x=LineFill(hdc,x,y-dir,-dir,xl,xr);
  }
  return xr;
}
```

В программе функция LineFill используется таким образом:

LineFill (xst, yst, 1, xst, xst);

Пример работы алгоритма закрашивания линиями приведен на рисунке.



Так же, но немного быстрее, работает функция FloodFill API Windows. Разница в быстродействии обусловлена тем, что для LineFill мы использовали стандартные функции для интерфейса прикладных программ, а FloodFill оптимизирована на системном уровне.

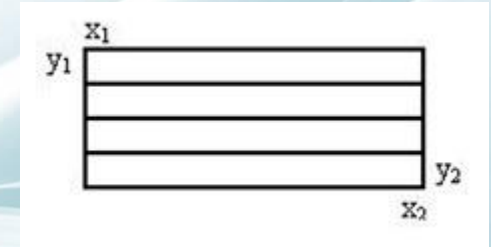
Алгоритмы заполнения, которые используют математическое описание контура.

Математическим описанием контура фигуры может служить уравнение *y=f(x)* для окружности, эллипса или другой кривой. Для многоугольника (полигона) контур задается множеством координат вершин (x_i, y_i). Возможны и другие формы описания контура. В любом случае алгоритмы данного класса не предусматривают обязательное предварительное создание пикселей контура растра - контур может совсем не выводиться в растр. Рассмотрим некоторые из подобных алгоритмов заполнения.

Заполнение прямоугольников

Среди всех фигур прямоугольник заполнять наиболее просто. Если прямоугольник задан координатами противоположных углов, например, левого верхнего (x_1,y_1) и правого нижнего (x_2,y_2), тогда алгоритм может заключаться в последовательном рисовании горизонтальных линий заданного цвета.

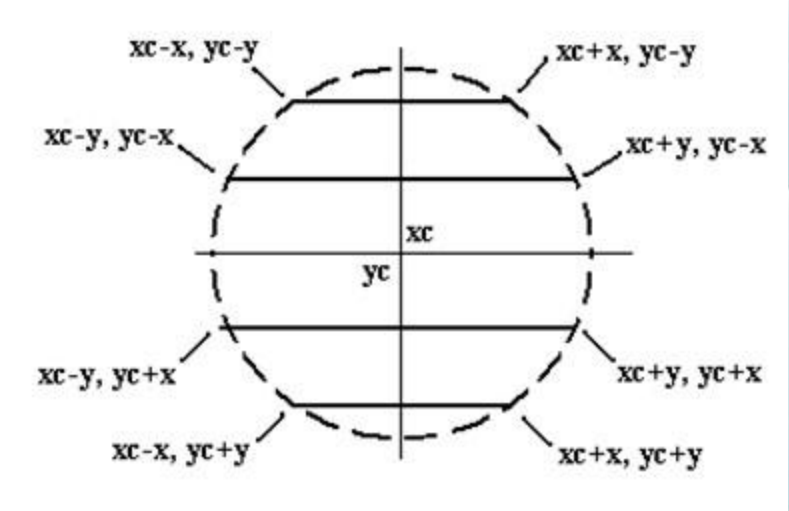
```
for (y=y1; y<=y2;y++)
  // Рисуем горизонтальную линию
  // с координатами (x1, y) - (x2, y)
```



Заполнение круга

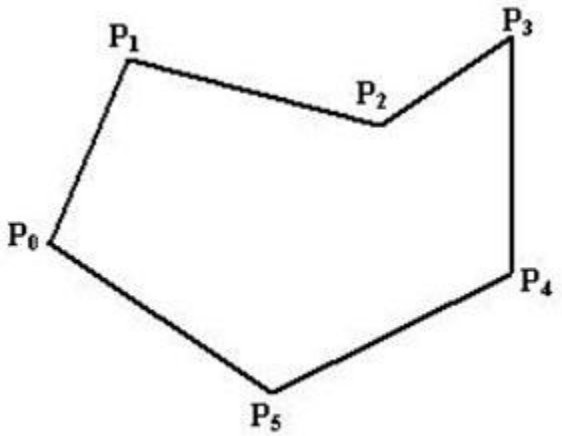
Для заполнения круга можно использовать алгоритм вывода контура (окружности). В процессе выполнения этого алгоритма последовательно вычисляются координаты пикселей контура в границах одного октанта. Для заполнения надлежит выводить горизонтали, которые соединяют пары точек на контуре, расположенные симметрично относительно оси y.

Так же может быть построен и алгоритм заполнения эллипса



Заполнение полигонов

Контур полигона определяется вершинами, которые соединены отрезками прямых. Это - векторная форма задания фигуры.



Рассмотрим один из наиболее популярных алгоритмов заполнения полигона. Его основная идея - закрашивание фигуры отрезками прямых линий. Наиболее удобно использовать горизонтали. Алгоритм представляет собою цикл вдоль оси *y*, в ходе этого цикла выполняется поиск точек сечения линии контура с соответствующими горизонталями. Этот алгоритм получил название XY

1. Найти min$\{y_i\}$ и max$\{y_i\}$ среди всех вершин P_i.
2. Выполнить цикл по y от y = min до y = max.

{

3. Нахождение точек пересечения всех отрезков контура с горизонталью y. Координаты x_j точек сечения записать в массив.
4. Сортировка массива $\{x_j\}$ по возрастанию x.
5. Вывод горизонтальных отрезков с координатами

$(x_0, y) - (x_1, y)$

$(x_2, y) - (x_3, y)$

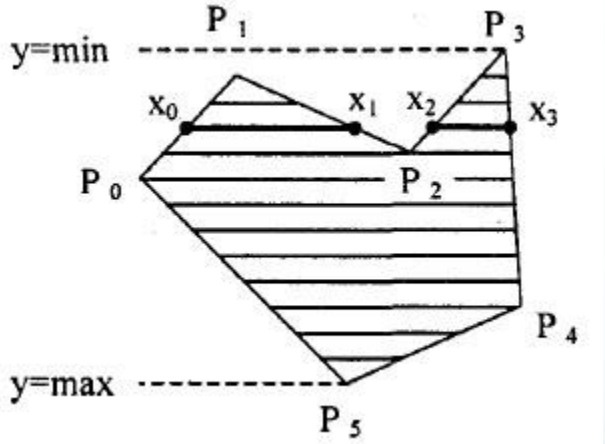
$(x_2k, y) - (x_{2k+1}, y)$

Каждый отрезок выводится цветом заполнения

}

В этом алгоритме использовано свойство топологии контура фигуры. Оно состоит в том, что любая прямая линия пересекает любой замкнутый контур четное количество раз.

Для выпуклых фигур точек пересечения с любой прямой всегда две. Таким образом, на шаге 3 этого алгоритма в массив $\{x_j\}$ всегда должно записываться парное число точек сечения.



При нахождении точек пересечения горизонтали с контуром необходимо принимать во внимание особые точки. Если горизонталь имеет координату (y), совпадающую с y_i, вершины P_i, тогда надлежит анализировать то, как горизонталь проходит через вершину. Если горизонталь при этом пересекает контур, как, например, в вершинах P_0 или P_4, то в массив записывается одна точка сечения. Если горизонталь касается вершины контура (в этом случае вершина соответствует локальному минимуму или максимуму как, например, в вершинах P_1, P_2, P_3 или P_5), тогда координата точки касания или не записывается, или записывается в массив два раза. Это условие четности количества точек пересечения, хранящихся в массиве {xj}.

Процедура определения точек пересечения контура с горизонтальной разверткой, учитывая анализ на локальный максимум, может быть достаточно сложной. Это замедляет работу. Радикальным решением для упрощения поиска точек сечения может быть смещение координат вершин контура или горизонталей заполнения таким образом, чтобы ни одна горизонталь не попала в вершину. Смещение можно выполнять различными способами, например, ввести в растр дробные координаты для горизонталей: $y_{min}+ 0.5$, $y_{min}+ 1.5$, ..., $y_{max}- 0.5$. Но такое упрощение процедуры нахождения точек пересечения приводит к искажению формы полигона.

Для определения координат (x) точек пересечения для каждой горизонтали необходимо перебирать все *n* отрезков контура. Координата пересечения отрезка $p_i - p_k$ с горизонталью *y* равна

$$x = x_i + (y_k - y)\,(x_k - x_i) / (y_k - y_i).$$

Количество тактов работы этого алгоритма:

$$N_{тактов} \approx (y_{max} - y_{min})\, N_{гор},$$

где y_{max}, y_{min} - диапазон координат y, $N_{гор}$ - число тактов, необходимых для одной горизонтали.

Оценим величину $N_{гор}$ как пропорциональную числу вершин

$$N_{гор} \approx k * n,$$

где *k* - коэффициент пропорциональности, *n* - число вершин полигона.

Возможны модификации приведенного алгоритма для ускорения его работы. Например, можно принять во внимание то, что каждая горизонталь в большинстве случаев пересекает небольшое количество ребер контура. Поэтому если при поиске точек сечения делать предварительный отбор ребер, которые находятся вокруг каждой горизонтали, то можно добиться уменьшения количества тактов работы с $N_{гор} = k * n$ до $k * n_p$, где n_p - число отобранных ребер. Например, разделим диапазон ymin- ymax пополам. Если для диапазона от y_{min} до $y_{ср}$ составить список отрезков (или вершин), которые попадают в этот диапазон, то в этот список будет включено приблизительно вдвое меньшее количество, чем для всего диапазона от y_{min} до y_{max}. Почему приблизительно - ибо это зависит от формы полигона. Таким образом, при работе алгоритма для каждой горизонтали в диапазоне y_{min} до $y_{ср}$ уже нужно не k * n тактов, а ~ k * n/2. Аналогично для диапазона $y_{ср}$- y_{max} также можно составить список ребер, который также будет почти вдвое меньшим.

Если принять, что подсчеты для каждой горизонтали теперь выполняются за вдвое меньшее количество тактов, а именно за $(N_{гор}/2)$, то общее число тактов:

$$N_{тактов} \approx (y_{max} - y_{min}) N_{гор}/2 + N_{доп},$$

где $N_{доп}$ - количество тактов для создания списка ребер.

§3. Стиль линии. Перо.

Для описания различных по виду изображений на основе линий используют термин стиль линий или перо. Термин перо иногда делает более понятной суть алгоритма вывода линий для некоторых стилей — в особенности для толстых линий. Например, если для тонкой непрерывной линии перо соответствует одному пикселю, то для толстых линий перо можно представить себе как фигуру или отрезок линии, который скользит вдоль оси линии, оставляя за собой след.

3.1. Алгоритмы вывода толстой линии

Взяв за основу любой алгоритм вывода обычных тонких линий, запишем его в следующем обобщенном виде:

Вывод пикселя (x, y)

Можно представить себе такой алгоритм, как цикл, в котором определяются координаты (x,y) каждого пикселя. Этот алгоритм можно модифицировать для вывода толстой линии следующим образом:

Вывод фигуры (или линии) пера с центром (x, y)

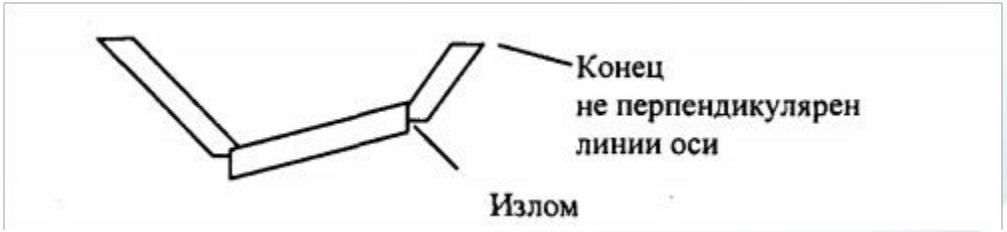
Вместо вывода отдельного пикселя стоит вывод фигуры или линии, соответствующей перу — прямоугольник, круг, отрезок прямой.

Такой подход к разработке алгоритмов толстых линий имеет преимущества и недостатки. Преимущество— можно прямо использовать эффективные алгоритмы для вычисления координат точек линии оси, например, алгоритмы Брезенхэма. Недостаток — неэффективность для некоторых форм пера. Для перьев, которые соответствуют фигурам с заполнением, количество тактов работы алгоритма пропорционально квадрату толщины линии. При этом большинство пикселей многократно закрашивается в одних и тех же точках.

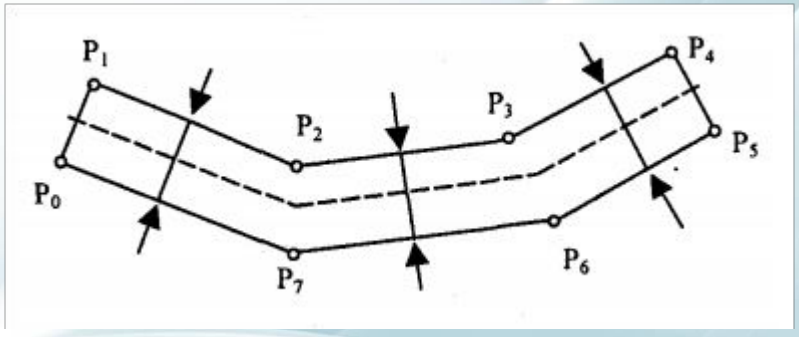
Такие алгоритмы более эффективны для перьев в виде отрезков линий. В этом случае каждый пиксель рисуется только один раз. Но здесь важным является наклон изображаемой линии. Ширина пера зависит от наклона. Очевидно, горизонтальное перо не может рисовать толстую горизонтальную линию.

Для вывода толстых линий с помощью пера в качестве отрезка линии чаще всего используются отрезки горизонтальной или вертикальной линий, реже — диагональные отрезки под углом 45 градусов. Целесообразность использования такого способа определяется большой скоростью вывода горизонтальных и вертикальных отрезков прямой. Для того чтобы достигнуть минимального количества тактов вывода, толстые линии, которые по наклону ближе к вертикальным, рисуют горизонтальным пером, а пологие линии — вертикальным пером.

При выводе толстых линий с использованием пера-отрезка линии заметны разрывы в углах разрывы в углах линий и плохие концы.



Для решения таких проблем иногда используют другие алгоритмы вывода толстых линий. Например, вывод толстой полилинии можно выполнить как рисование полигона с заполнением.



Очевидный недостаток такого подхода — меньшая скорость вывода, поскольку заполнение полигона— это существенно более трудоемкая задача, чем вывод линий, а кроме того, нужно еще определять координаты его вершин.

Третья разновидность алгоритмов вывода толстых линий — рисование толстой линии последовательным наложением нескольких тонких линий, смещенных одна относительно второй.

3.2. Алгоритмы вывода пунктирной линии

Алгоритм для рисования тонкой пунктирной линии можно получить из алгоритма вывода тонкой непрерывной линии:

Вывод пиксела (x, y)

заменой процедуры вывода пикселя более сложной конструкцией:

Проверка значения счетчика C :

Если C удовлетворяет некоторым условиям, то

Вывод пиксела (x, y)

значение C увеличивается на единицу

В таком алгоритме используется новая переменная (C)— счетчик пикселей линии. Если значение C удовлетворяет некоторому логическому условию, то рисуется пиксель заданного цвета с текущими координатами *(x, y)*. Логическое условие будет определять стиль линии. Например, если условием будет четность значения C, то получим линию из одиночных точек. Для рисования пунктирной линии можно анализировать остаток от деления C на S. Например, если рисовать пиксели линии только тогда, когда C mod S < S/2, то получим пунктирную линию с длиной штрихов S/2 и с шагом S.

При выводе полилиний, которые состоят из отрезков прямых, или сплайновых кривых, необходимо предотвратить обнуление значения счетчика в начале каждого отрезка и обеспечить продолжение непрерывного приращения вдоль всей сложной линии. Иначе будут нестыковки пунктира.

Использование переменной-счетчика затруднено при генерации пунктирных линий в алгоритмах, которые используют симметрию, например, при выводе круга или эллипса. В этом случае будут нестыковки пунктира на границах октантов или квадрантов.

3.3. Алгоритм вывода толстой пунктирной линии

Объединив алгоритм для вывода толстой непрерывной линии и алгоритм для тонкой пунктирной линии, можно получить следующий алгоритм:

Проверка значения счетчика (С) :

Если (С) удовлетворяет условию , то вывод фигуры (линии) пера с центром в (x, y)

C = C + 1;

Такой алгоритм достаточно прост. На практике используются и более изощренные алгоритмы.

§4. Стиль заполнения. Кисть. Текстура

При выводе фигур могут использоваться различные стили заполнения. Для обозначения стилей заполнения, отличных от сплошного стиля, используют такие понятия, как кисть и текстура. Их можно считать синонимами, однако понятие текстуры обычно используется применительно к трехмерным объектам, а кисть — для изображения двумерных объектов.

Текстура— это стиль заполнения, закрашивание, которое имитирует сложную рельефную объемную поверхность, выполненную из какого-то материала.

Для описания алгоритмов заполнения фигур с определенным стилем используем тот же способ, что и для описания алгоритмов рисования линий.

Мы уже ранее рассмотрели некоторые алгоритмы заполнения, и вы, наверное, согласитесь, что описание всех разновидностей подобных алгоритмов можно дать с помощью такой обобщенной схемы:

4.1. Вывод пикселя заполнения цвета C с координатами (x, y)

Например, в алгоритме вывода полигонов пиксели заполнения рисуются в теле цикла горизонталей, а все другие операции предназначены для подсчета координат *(x, y)* этих пикселей. Сплошное заполнение означает, что цвет (C) всех пикселей одинаков, то есть C = const.

Нам нужно как-то изменять цвет пикселей заполнения, чтобы получить определенный узор. Преобразуем алгоритм заполнения следующим образом:

$$C = f(x, y)$$

4.2. Вывод пикселя заполнения (x, y) цветом C

Функция *f(x, y)* будет определять стиль заполнения. Аргументами функции цвета являются координаты текущего пикселя заполнения. Однако в отдельных случаях эти аргументы не нужны. Например, если цвет C вычислять как случайное значение в определенных границах: *C = random (),* то можно создать иллюзию шершавой матовой поверхности (рис.8).

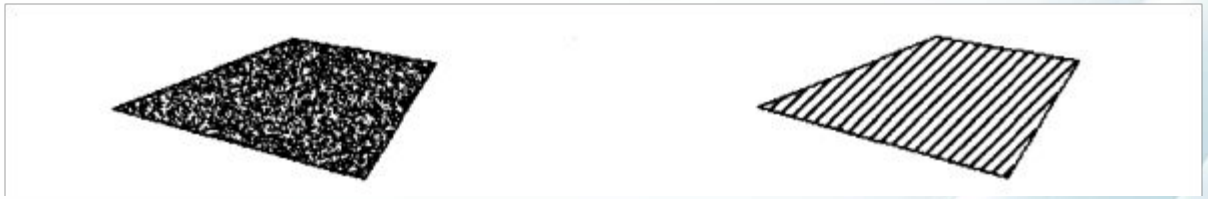


Рис.8. Матовая поверхность

Рис.9. Штриховка

Другой стиль заполнения — штриховой (рис. 9). Для него функцию цвета также можно записать в аналитической форме:

$$f(x, y) = \begin{cases} C_ш, & \text{если } (x + y) \bmod S < T, \\ C_ф & \text{— в других случаях,} \end{cases}$$

где S — период, а T — толщина штрихов, $C_ш$ — цвет штрихов, $C_ф$ — цвет фона.

Если не рисовать пиксели фона, то можно создать иллюзию полупрозрачной фигуры. Подобную функцию можно записать и для других типов штриховки. Аналитическая форма описания стиля заполнение позволяет достаточно просто изменять размеры штрихов при изменениях масштаба показа, например, для обеспечения режима WYSIWYG.

Зачастую при использовании кистей и текстур используется копирование небольших растровых изображений. Такой алгоритм заполнения можно описать вышеупомянутой общей схемой, если строку $C = f(x,y)$ заменить двумя другими строками:

- Координаты пикселя заполнения (x,y) преобразуем в растровые координаты образца кисти (xt/ yt)
- По координатам (xt, yt) определяем цвет (C) пиксела в образце кисти Вывод пиксела заполнения цвета C с координатами (x,y)

Преобразование координат пикселя заполнения (x,y) в координаты внутри образца кисти можно выполнить таким образом:

$$xt = x \bmod m, \qquad yt = y \bmod n,$$

где m, n — размеры растра кисти соответственно по горизонтали и вертикали. При этом координаты (xt, yt) будут в диапазоне xt- 0...m - 1, yt= 0...n - при любых значениях x и y. Таким образом, обеспечивается циклическое копирование фрагментов кисти внутри области заполнения фигуры

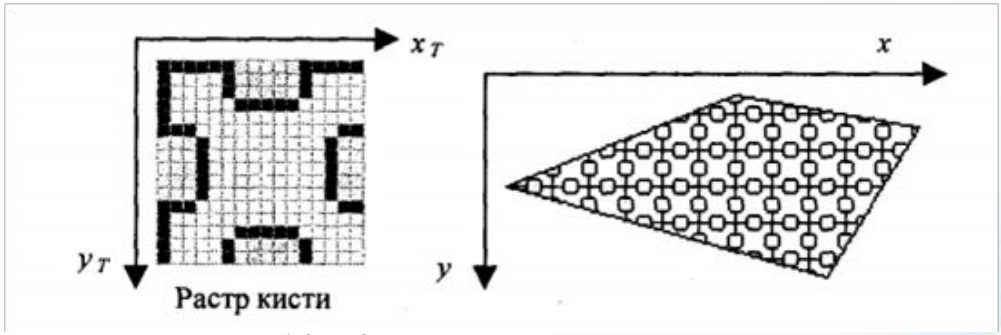


Рис.10. Копирование растра кисти

Удобно, когда размеры кисти равны степени двойки. В этом случае вместо операций взятия остатка (mod) можно использовать более быстродействующие для цифровых компьютеров поразрядные двоичные операции. Приведен пример вычисления остатка от деления на 16.

Биты двоичного кода числа *X*:

```
x x . . . x x x x x
X mod 16 = 0 0 . . . 0 x x x x
```

Здесь можно использовать поразрядную операцию & (И).

Еще один пример. Если необходимо вычислить X mod 256, а значение X записано в регистре AX микропроцессора (совместимого с 80x86), то в качестве результата достаточно взять содержимое младшей байтовой части этого регистра — AL. Растровые текстуры и кисти широко используются в современной компьютерной графике, в том числе и в 3D-графике. Для отображения трехмерных объектов, часто используются полигональные поверхности, каждая грань отображается с наложенной текстурой. Поскольку объекты обычно показываются с разных ракурсов — повороты, изменения размеров и тому подобное, то необходимо соответственно трансформировать и каждую грань с текстурой. Для этого используются проективные текстуры. Общая схема алгоритма заполнения контуров полигонов для проективных текстур такая же, как и приведенная выше. Однако растровый образец здесь представляет всю грань, а преобразование координат из *(x, y)* в *(xt yt)* более сложное, например, аффинное:

$$x_T = A\,x + B\,y + C,$$
$$y_T = D\,x + E\,y + F,$$

где коэффициенты *A, B, ... , F* — константы при пересчете координат всех пикселей для отдельной текстурированной грани. Такое преобразование координат можно использовать, если привязать текстуру к грани по трем опорным точкам. Пример наложения проективной текстуры приведен на рис. 11.

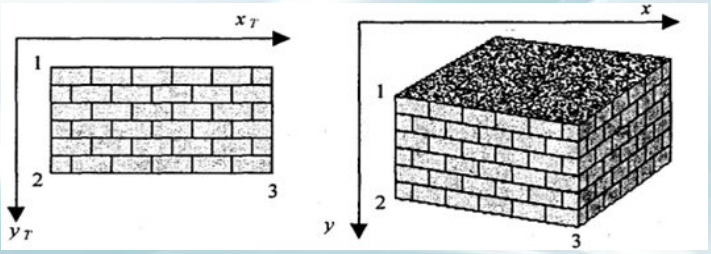


Рис. 11. Наложение проективной текстуры на две грани объекта

Привязывание по трем точкам соответствует уравнениям:

$$x_{Ti} = A\,x_i + B\,y_i + C,$$
$$y_{Ti} = D\,x_i + E\,y_i + F,$$

где $i = 1,2,3$. По известным координатам $(xti\ ,\ yti)$ и $(xi\ ,\ yi)$ можно найти коэффициенты A, B, ... , F, если решить систему линейных уравнений. Эта система распадается на две независимые системы третьего порядка. Для упрощения записи заменим xti на Xi, а $y$ti на Yi.

$$X_1 = A\,x_1 + B\,y_1 + C,$$
$$X_2 = A\,x_2 + B\,y_2 + C,$$
$$X_3 = A\,x_3 + B\,y_3 + C,$$

$$Y_1 = D\,x_1 + E\,y_1 + F,$$
$$Y_2 = D\,x_2 + E\,y_2 + F,$$
$$Y_3 = D\,x_3 + E\,y_3 + F.$$

Для решения систем линейных уравнений известно множество способов, используем способ, основанный на вычислении определителей. Решение первой системы для коэффициентов A,B и C можно записать в виде:

$$A = detA / det,$$
$$B = detB / det,$$
$$C = detC / det,$$

где, det это главный определитель системы, а определители detA, detB, detC получаются заменой соответствующих столбцов в det столбцом свободных членов:

$$detA = \begin{Vmatrix} X_1 & y_1 & 1 \\ X_2 & y_2 & 1 \\ X_3 & y_3 & 1 \end{Vmatrix}, \quad detB = \begin{Vmatrix} x_1 & X_1 & 1 \\ x_2 & X_2 & 1 \\ x_3 & X_3 & 1 \end{Vmatrix}, \quad detC = \begin{Vmatrix} x_1 & y_1 & X_1 \\ x_2 & y_2 & X_2 \\ x_3 & y_3 & X_3 \end{Vmatrix}$$

Если главный определитель равен нулю, то это означает, что решение системы невозможно. Это может быть, например, тогда, когда все три точки (x1, y1),(x2,y2) и (x3,y3) располагаются вдоль прямой линии (грань вид с торца). Однако в этом случае рисовать текстуру и не нужно. Вычислить определитель третьей степени можно, например, по "правилу Саррюса" (_). Для этого справа нужно дописать первые два столбца, а затем сложить (вычесть) произведения по диагоналям:

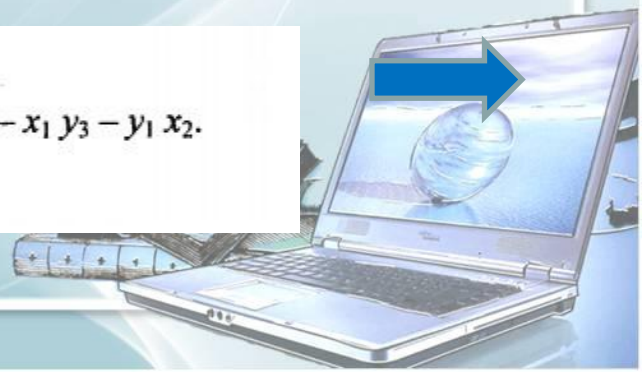
$$\begin{Vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{Vmatrix} = \begin{Vmatrix} a_{11} & a_{12} & a_{13} & a_{11} & a_{12} \\ a_{21} & a_{22} & a_{23} & a_{21} & a_{22} \\ a_{31} & a_{32} & a_{33} & a_{31} & a_{32} \end{Vmatrix} =$$

$$- \quad - \quad - \quad + \quad + \quad +$$

$$= a_{11}\,a_{22}\,a_{33} + a_{12}\,a_{23}\,a_{31} + a_{13}\,a_{21}\,a_{32} - a_{13}\,a_{22}\,a_{31} - a_{11}\,a_{23}\,a_{32} - a_{12}\,a_{21}\,a_{33}.$$

вычислим главный определитель:

$$det = \begin{Vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{Vmatrix} = x_1\,y_2 + y_1\,x_3 + x_2 y_3 - y_2 x_3 - x_1\,y_3 - y_1\,x_2.$$



Преобразуем выражение так, чтобы уменьшить число умножений:

det = x1 (y2 - y3) + x2 (y3 - y1) + x3 (y1 - y3).

Аналогично вычисляются определители detA и detB. Определитель detC является самым сложным из всех. Но его вычислять не обязательно. Запишем решение системы в следующем виде:

$$A = detA / det = (X_1 (y_2 - y_3) + X_2 (y_3 - y_1) + X_3 (y_1 - y_2)) / det,$$
$$B = detB / det = (x_1 (X_2 - X_3) + x_2 (X_3 - X_1) + x_3 (X_1 - X_2)) / det,$$
$$C = X_1 - A\, x_1 - B\, y_1 .$$

Таким же способом решаем систему уравнений для D,E и F.

$$D = (Y_1 (y_2 - y_3) + Y_2 (y_3 - y_1) + Y_3 (y_1 - y_2)) / det;$$
$$E = (x_1 (Y_2 - Y_3) + x_2 (Y_3 - Y_1) + x_3 (Y_1 - Y_2)) / det;$$
$$F = Y_1 - D\, x_1 - E\, y_1 .$$

Заметьте, что здесь главный определитель det совпадает с определителем первой системы уравнений — для A, B и C. Наложение текстур в перспективной проекции сложнее, чем для аксонометрической проекции. Рассмотрим рис.12 на котором изображен текстурированный прямоугольник.

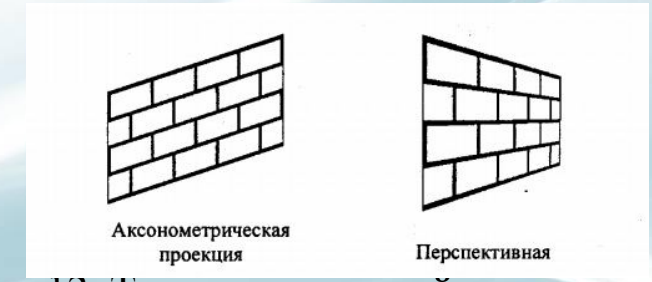
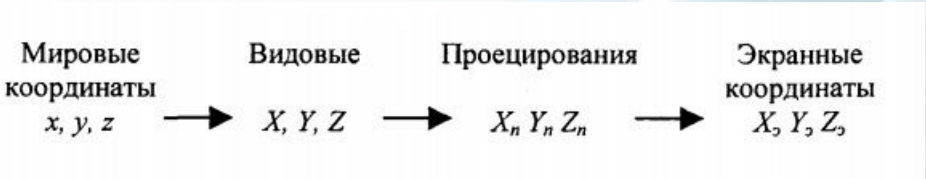


Рис. 12. Текстурированный прямоугольник

Текстурирование в перспективной проекции

Прямоугольник в аксонометрической (параллельной) проекции всегда выглядит как параллелограмм, поскольку для этой проекции сохраняется параллельность прямых и отношение длин. В перспективной (центральной) проекции это уже не параллелограмм и не трапеция (в косоугольной — трапеция), поскольку параллельность и отношение длин здесь не сохраняются. А что сохраняется? Как изображать плоские грани? В этой книге мы рассматриваем проекции на плоскость. Для таких проекций прямые линии остаются прямыми линиями, поэтому грани можно выводить как полигоны.

Здесь уместно вспомнить, как формируется изображение в некоторой проекции средствами компьютерной графики.



Если считать, что точки текстуры должны соответствовать точкам на объекте, то координаты текстуры должны связываться с мировыми координатами. Однако поскольку для аксонометрической проекции в цепочке от мировых координат до экранных все преобразования линейны, то вполне допустимо связать координаты текстуры с экранными координатами одним аффинным преобразованием.

Для перспективной проекции так делать нельзя. Преобразование координат из видовых координат в координаты плоскости проецирования не линейно. Поэтому экранные координаты вначале следует преобразовать в такие, которые линейно связаны с мировыми — это могут быть, скажем, видовые. А затем видовые координаты (или непосредственно мировые) связать с координатами текстуры аффинным преобразованием, используя, например, метод трех точек.

Рассмотрим, как можно выводить в перспективной проекции полигон с текстурой. Будем использовать алгоритм заполнения полигона горизонтальными линиями, уже рассмотренный нами. На рис. 13 изображена одна из горизонталей (AB). Вершины полигона (1-2-3-4) здесь заданы экранными двумерными координатами. Для краткости изложения положим, что экранные координаты совпадают с координатами в плоскости проецирования. В ходе вывода полигона для связи с текстурой будем вычислять видовые координаты произвольной точки (P) этого полигона. Для этого будем использовать в качестве базовой такую операцию: по известным видовым координатам концов отрезка находим видовые координаты точки отрезка, заданной координатами в плоскости проецирования.

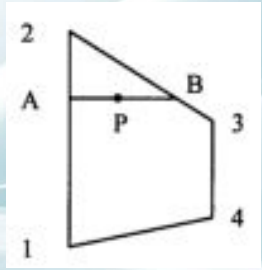


Рис. 13. Полигон

Для определения видовых координат X, Y, Z точки A должны быть известны видовые координаты концов отрезка (1-2). Тогда справедливы соотношения:

$$\frac{X - X_1}{X_2 - X_1} = \frac{Y - Y_1}{Y_2 - Y_1} = \frac{Z - Z_1}{Z_2 - Z_1}.$$

Выберем пропорцию, связывающую координаты X и Z. Тогда: X = a + Zb, где

a = X1 - Z1(X2-X1)/(Z2-Z1), B = (X2-X1)/(Z2-Z1).

Теперь запишем для перспективной проекции соотношение между видовыми координатами произвольной точки и координатой X в плоскости проецирования:

$$X_n = X\frac{Z_к - Z_{пл}}{Z_к - Z},$$

где ZK — это координата камеры (точки схода лучей проектирования), Zm - координата плоскости проецирования. Перепишем это равенство так:

X =c +Zd

где c = XnZk / (Zk - Zm), d = - Xn / (Zk - Zm). Теперь решим систему уравнений:

X = a + Zb,

X = c+ Zd.

Решением системы будет:

$$Z = \frac{c - a}{b - d},$$

После чего вычисляется X, например, по формуле X = a + Zb. Для определения координаты Y достаточно заменить везде X на Y. Здесь можно отметить, что вычисление видовых координат (X Y, Z) соответствует обратному проективному преобразованию.

Найдя видовые координаты точки A, мы можем точно так же вычислить видовые координаты и для точки B, лежащей на отрезке (3-4). Аналогично вычисляются видовые координаты точки P. Следует отметить, что, несмотря на то, что для преобразования координат необходимо вычислять дробно-линейные выражения, цикл вычислений можно сделать достаточно простым подобно инкрементным алгоритмам. Для тренировки можете попробовать выполнить данное упражнение.

Для точного наложения текстур на поверхности используются и более сложные преобразования координат. (!!!!)

Одна из проблем наложения текстур заключается в том, что преобразование растровых образцов (повороты, изменение размеров и тому подобное) приводят к ухудшению качества растров. Повороты растра добавляют ступенчатость (aliasing); увеличение размеров укрупняет пиксели, а уменьшение размеров растра приводит к потере многих пикселей образца текстуры, появляется муар. Для улучшения текстурованных изображений используют методы фильтрации (интерполяции) растров текстур. Также используются несколько образцов текстур для различных ракурсов показа (mipmaps) — компьютерная система во время отображения находит в памяти наиболее пригодный растровый образец.

Для использования текстур необходим достаточный объем памяти компьютера — количество растровых образцов может достигать десятков, сотен и более в зависимости от количества типов объектов и многообразия пространственных сцен. Чтобы как можно быстрее создавать изображение, необходимо сохранять текстуры в оперативной памяти.

Для экономии памяти, выделяемой для текстур, можно использовать блочное текстурирование. Текстура здесь уже не представляет всю грань целиком, а лишь отдельный фрагмент, который циклически повторяется в грани. Это напоминает процесс размножения рисунка кисти при закрашивании полигонов, рассмотренный нами в этом разделе.

Разумеется, далеко не для всех объектов можно использовать такой способ отображения, однако, например, для образцов современной массовой "коробочной" архитектуры в этом плане имеются практически неограниченные возможности.

Примеры выполнения алгоритмов на языке Pascal

1. Вывод окружности

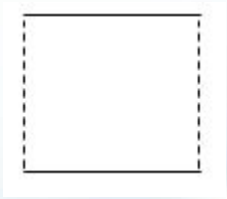
2. Вывод прямой линии

Лабораторные задания

№1. Вывести пунктирную прямую зелёного цвета:

№2. Вывести параллелограмм, залитый красным цветом:

№3. Вывести на экран квадрат имеющий 2 пунктирных ребра:



Раздел 3.
Алгоритмы вывода трехмерных объектов.

§ § 1§ 1. Модели описания поверхностей.
§ 2. Визуализация объемных изображений.
§ 3. Закрашивание поверхностей.

Лабораторные занятия.

§ 1. Модели описания поверхностей

1.1. Векторная полигональная модель

Для описания пространственных объектов здесь используются такие элементы: вершины; отрезки прямых (векторы); полилинии, полигоны; полигональные поверхности (рис.14).

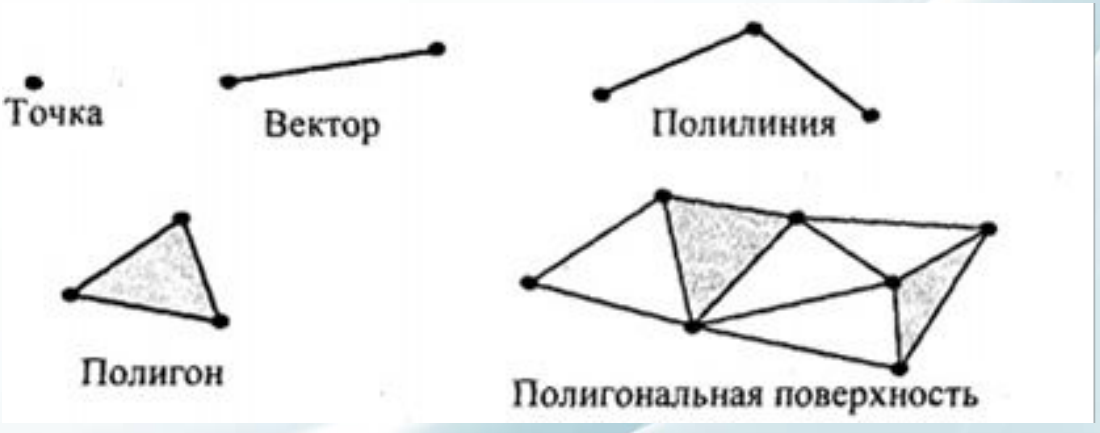


Рис.14. Базовые элементы векторно-полигональной модели

Вершина может моделировать отдельный точечный объект, размер которой не имеет значения, а также может использоваться в качестве конечных точек для линейных объектов и полигонов. Двумя вершинами задается вектор. Не сколько векторов составляют полилинию. Полилиния может моделировать отдельный линейный объект, толщина которого не учитывается, а также может представлять контур полигона. Полигон моделирует площадной объект. Один полигон может описывать плоскую грань объемного объекта. Несколько граней составляют объемный объект в виде полигональной поверхности— многогранник или незамкнутую поверхность (в литературе часто употребляется название "полигональная сетка").

Векторную полигональную модель можно считать наиболее распространенной в современных системах трехмерной КГ. Ее используют в системах автоматизированного проектирования, в компьютерных играх и тренажерах, в САПР, геоинформационных системах и тому подобное.

Обсудим структуры данных, которые используются в векторной полигональной модели. В качестве примера объекта будет куб. Рассмотрим, как можно организовать описание такого объекта в структурах данных.

Первый способ. Сохраняем все грани в отдельности:

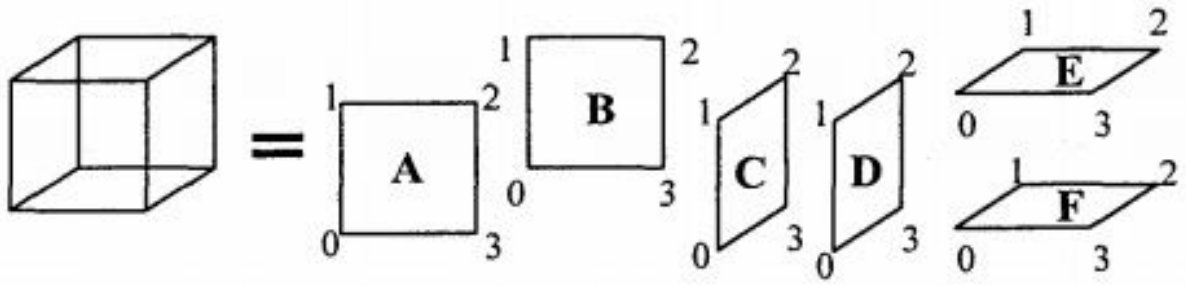


Рис. 4.3. Первый способ описания куба

Грань A = { (x_{A0},y_{A0},z_{A0}), (x_{A1},y_{A1},z_{A1}), (x_{A2},y_{A2},z_{A2}), (x_{A3},y_{A3},z_{A3}) }.

Грань B = { (x_{B0},y_{B0},z_{B0}), (x_{B1},y_{B1},z_{B1}), (x_{B2},y_{B2},z_{B2}), (x_{B3},y_{B3},z_{B3}) }.

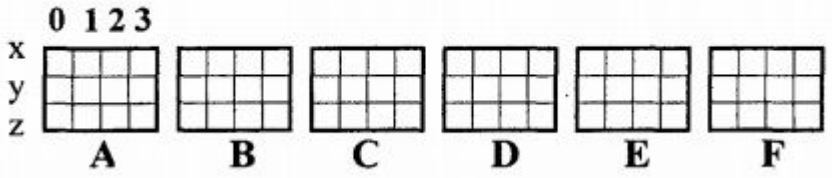
Грань C = { (x_{C0},y_{C0},z_{C0}), (x_{C1},y_{C1},z_{C1}), (x_{C2},y_{C2},z_{C2}), (x_{C3},y_{C3},z_{C3}) }.

Грань D = { (x_{D0},y_{D0},z_{D0}), (x_{D1},y_{D1},z_{D1}), (x_{D2},y_{D2},z_{D2}), (x_{D3},y_{D3},z_{D3}) }.

Грань E = { (x_{E0},y_{E0},z_{E0}), (x_{E1},y_{E1},z_{E1}), (x_{E2},y_{E2},z_{E2}), (x_{E3},y_{E3},z_{E3}) }.

Грань F = { (x_{F0},y_{F0},z_{F0}), (x_{F1},y_{F1},z_{F1}), (x_{F2},y_{F2},z_{F2}), (x_{F3},y_{F3},z_{F3}) }.

И схематично изобразим:



В компьютерной программе такой способ описания объекта можно реализовать разнообразно. Например, для каждой грани открыть в памяти отдельный массив. Можно все грани записывать в один массив-вектор. А можно использовать классы (языком C++) как для описания отдельных граней, так и объектов в целом. Можно создавать структуры, которые объединяют тройки *(x, y, z)*, или сохранять координаты отдельно. В значительной мере это относится уже к компетенции программиста, зависит от его вкуса. Принципиально это мало что изменяет — так или иначе в памяти необходимо сохранять координаты вершин граней плюс некоторую информацию в качестве накладных затрат.

Рассчитаем объем памяти, необходимый для описания куба следующим образом:

$$П_i = 6 * 4 * 3 * Р_в,$$

где $Р_в$ — разрядность чисел, необходимая для представления координат.

Здесь шесть граней описываются 24 вершинами. Такое представление избыточно — каждая вершина записана трижды. Не учитывается то, что у граней есть общие вершины.

Второй способ описания. Для такого варианта координаты восьми вершин сохраняются без повторов. Вершины пронумерованы, а каждая грань дается в виде списка индексов вершин (указателей на вершины).

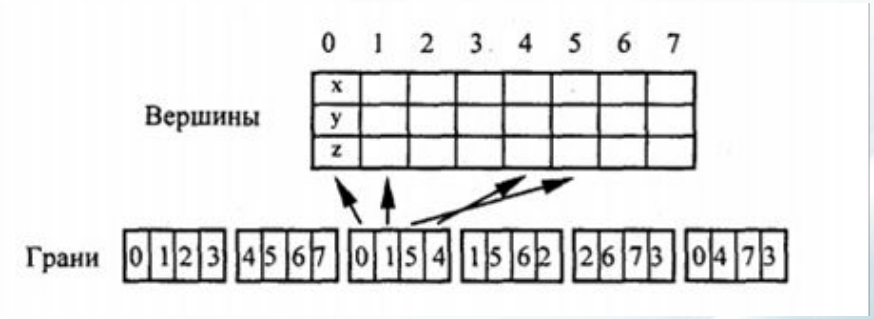


Рис.15. В массивах граней сохраняются индексы вершин

Оценим затраты памяти:

$$П_2 = 8 * 3 * Рв + 6 * 4 * Ри,$$

где *Рв* - разрядность координат вершин, *Ри* - разрядность индексов.

Третий способ описания. Этот способ (в литературе его иногда называют линейно-узловой моделью) основывается на иерархии: вершины-ребра-грани.

Оценим затраты памяти:

$$П_З = 8 * 3 * Р_в + 12 * 2 * Р_{инд.в} + 6 * 4 * Р_{инд.р,}$$

где $Р_в$ - разрядность координат, $Р_{инд.в}$ $Р_{инд.р}$ - разрядность индексов вершин и ребер соответственно.

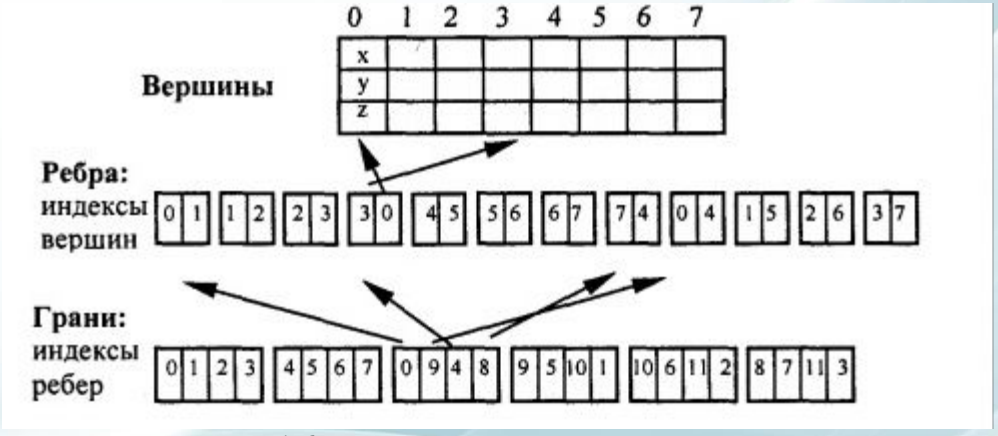


Рис. 16. Линейно-узловая модель

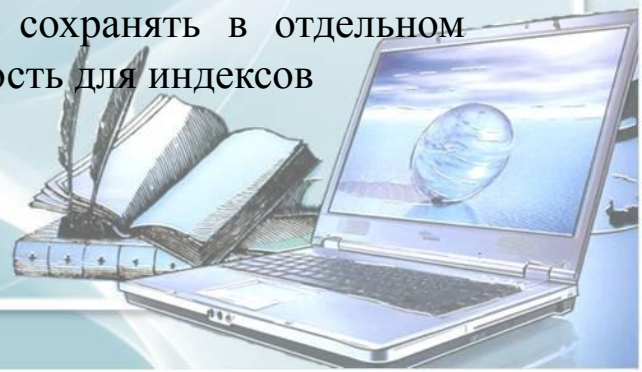
Для сравнения объемов памяти этих трех вариантов необходимо определиться с разрядностью данных. Предположим, что разрядность координат и индексов составляет четыре байта. Это соответствует, например, типу чисел с плавающей точкой float для координат и целому типу long для индексов (названия этих типов на компьютерном языке C, C++). Тогда затраты памяти в байтах составляют:

$$П_1 = 6 \times 4 \times 3 \times 4 = 288,$$
$$П_2 = 8 \times 3 \times 4 + 6 \times 4 \times 4 = 192,$$
$$П_3 = 8 \times 3 \times 4 + 12 \times 2 \times 4 + 6 \times 4 \times 4 = 288.$$

Пусть для координат отведено 8 байтов (тип с плавающей точкой double), а для индексов — 4 байта. Тогда:

$$П_1 = 6 \times 4 \times 3 \times 8 = 576,$$
$$П_2 = 8 \times 3 \times 8 + 6 \times 4 \times 4 = 288,$$
$$П_3 = 8 \times 3 \times 8 + 12 \times 2 \times 4 + 6 \times 4 \times 4 = 384.$$

Когда разрядность для координат больше, чем для индексов, то ощутимо преимущество второго и третьего вариантов. Наиболее экономичным можно считать второй вариант. Необходимо заметить, что такой вывод мы сделали для куба. Для других типов объектов соотношение вариантов может быть иным. Кроме того, необходимо учитывать такие варианты построения структур данных: использован ли единый массив для всех объектов, или же для каждого объекта предназначен отдельный массив (при объектно-ориен тированном стиле программирования каждый объект можно сохранять в отдельном классе). Это может обуславливать разную необходимую разрядность для индексов



Скорость вывода полигонов. Если для полигонов необходимо рисовать линию контура и точки заполнения, то первый и второй варианты близки по быстродействию — и контуры, и заполнения рисуются одинаково.

Отличия в том, что для второго варианта сначала надо выбирать индекс вершины, что замедляет процесс вывода. В обоих случаях для смежных граней повторно рисуется общая часть контура.

Для третьего варианта можно предусмотреть более совершенный способ рисования контура— каждая линия будет рисоваться только один раз, если в массивах описания ребер предусмотреть бит, означающий, что это ребро уже нарисовано. Это обуславливает преимущества третьего варианта по быстродействию.

Блокирование повторного рисования линий контуров смежных граней позволяет решить также проблему искажения стиля линий, если линии контуров не сплошные, а, например, пунктирные.

Топологический аспект. Представим, что имеется несколько смежных граней. Что будет, если изменить координаты одной вершины в структурах данных? Результат приведен на рис.17.

Рис. 17. Результат изменения координат одной вершины

Поскольку для второго и третьего вариантов каждая вершина сохраняется в одном экземпляре, то изменение ее координат автоматически приводит к изменению всех граней, в описании которых сохраняется индекс этой вершины. Следует заметить, что подобного результата можно достичь и при структуре данных, соответствующей первому варианту. Можно предусмотреть поиск других вершин, координаты которых совпадают с координатами точки А. Иначе говоря, поддержка такой операции может быть обеспечена как структурами данных, так и алгоритмически. Однако когда нужно разъединить смежные грани, то для второго и третьего вариантов это сложнее, чем для первого — необходимо записать в массивы новую вершину, новые ребра и определить индексы в массивах граней. При разработке новой графической системы обычно приходится решать такой вопрос: какие операции реализовывать только алгоритмически, а какие обеспечивать структурами данных? Ответ на это можно дать, проанализировав много других факторов.

Достоинства векторной полигональной модели:

- удобство масштабирования объектов. При увеличении или уменьшении объекты выглядят более качественно, чем при растровых моделях описания. Диапазон масштабирования определяется точностью аппроксимации и разрядностью чисел для представления координат вершин;
- небольшой объем данных для описания простых поверхностей, которые адекватно аппроксимируются плоскими гранями;
- необходимость вычислять только координаты вершин при преобразованиях систем координат или перемещении объектов;
- аппаратная поддержка многих операций в современных графических видеосистемах, которая обуславливает достаточную скорость для анимации.

Недостатки векторной полигональной модели:

- сложные алгоритмы визуализации для создания реалистичных изображений; сложные алгоритмы выполнения топологических операций, таких, например, как разрезы;
- аппроксимация плоскими гранями приводит к погрешности моделирования. При моделировании поверхностей, которые имеют сложную фрактальную форму, обычно невозможно увеличивать количество граней из-за ограничений по быстродействию и объему памяти компьютера.

1.2. Воксельная модель

Воксельная модель — это трехмерный растр. Подобно тому, как пиксели располагаются на плоскости 2D - изображения, так и вокселы образовывают трехмерные объекты в определенном объеме (рис.18). Воксел — это элемент объема (voxel — volume element).

Как мы знаем, каждый пиксель должен иметь свой цвет. Каждый воксел также имеет свой цвет, а, кроме того, прозрачность. Полная прозрачность воксела означает пустоту соответствующей точки объема. При моделировании объема каждый воксел представляет элемент объема определенного размера. Чем больше вокселов в определенном объеме и меньше размер вокселов, тем точнее моделируются трехмерные объекты — увеличивается разрешающая способность.

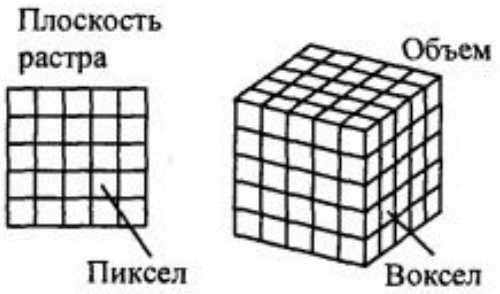


Рис. 18. Пикселы и вокселы

Достоинства воксельной модели:

- позволяет достаточно просто описывать сложные объекты и сцены; простая процедура отображения объемных сцен;
- простое выполнение топологических операций над отдельными объектами и сценой в целом. Например, просто выполняется показ разреза — для этого соответствующие вокселы можно сделать прозрачными.

Недостатки воксельной модели:

- большое количество информации, необходимой для представления объемных данных. Например, объем 256x256x256 имеет небольшую разрешающую способность, но требует свыше 16 миллионов вокселов;
- значительные затраты памяти ограничивают разрешающую способность, точность моделирования; большое количество вокселов обусловливает малую скорость создания изображений объемных сцен;
- как и для любого растра, возникают проблемы при увеличении или уменьшении изображения. Например, при увеличении ухудшается разрешающая способность изображения.

§ 2. Визуализация объемных изображений.

Любой объект, в том числе и объемный, может быть изображен различными способами. В одном случае необходимо показать внутреннюю структуру объектов, в другом — внешнюю форму объекта, в третьем — имитировать реальную действительность, в четвертом— поразить воображение зрителя чем-то неизвестным. Условно разделим способы визуализации по характеру изображений и по степени сложности соответствующих алгоритмов на такие уровни:

- Каркасная ("проволочная") модель.
- Показ поверхностей в виде многогранников с плоскими гранями или сплайнов с удалением невидимых точек.
- То же, что и для второго уровня, плюс сложное закрашивание объектов для имитации отражения света, затенения, прозрачности, использование текстур.

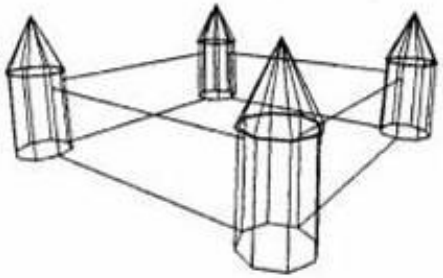


Рис. 19. Каркасная модель

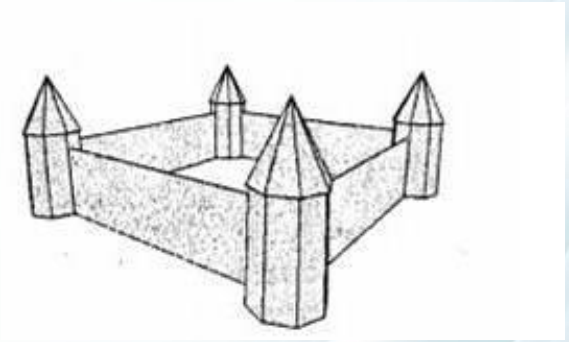


Рис. 20. Удаление невидимых точек

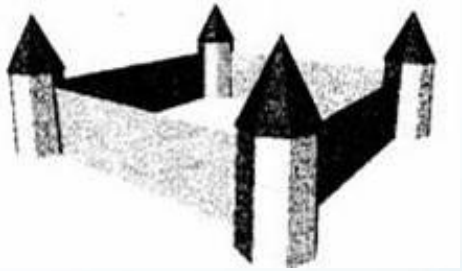


Рис. 21. Закрашивание граней с учетом освещения

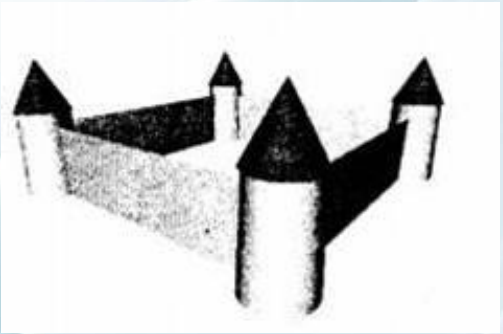
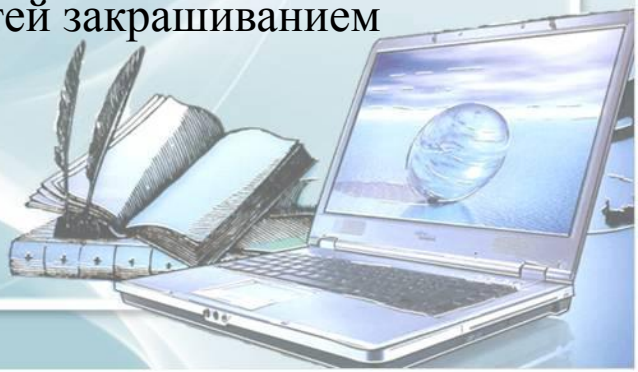


Рис. 22. Имитация гладких поверхностей закрашиванием



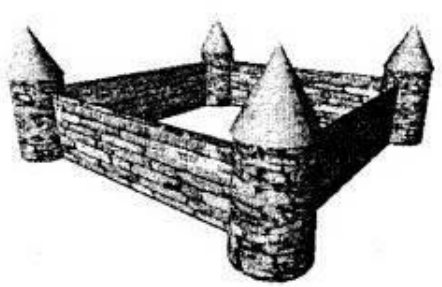


Рис. 23. Наложение текстуры

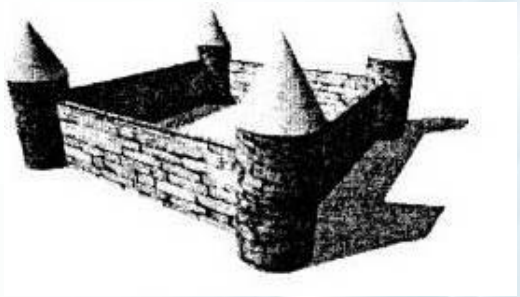


Рис. 24. Тени

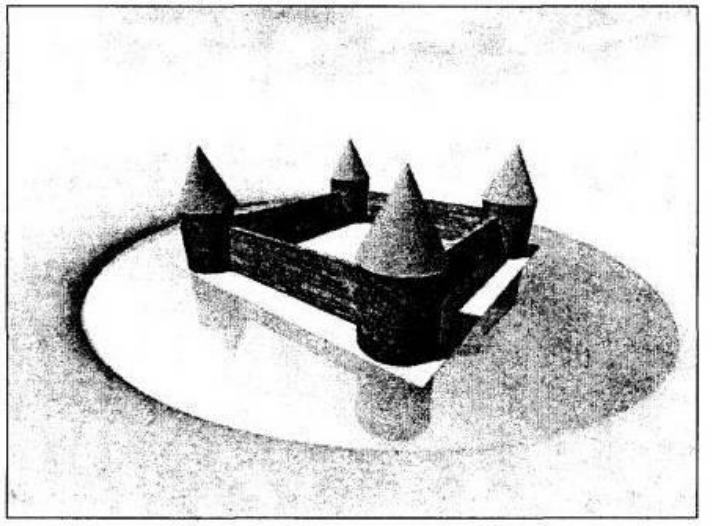


Рис. 25. Добавлены фон, матовые и зеркальные поверхности

Простейшая каркасная модель часто применяется в процессе редактировании объемных объектов. Визуализация второго уровня используется для упрощенного показа объемных объектов. Например, для графиков функции $z = f(x,y)$ (в виде рельефа поверхности) часто достаточно показать все грани сетки одним цветом, но зато необходимо обязательно удалить невидимые точки. Это более сложная процедура по сравнению с выводом каркасного изображения.

Сложность процесса графического вывода возрастает по мере приближения к некоторому идеалу для компьютерной графики — создания полной иллюзии естественных, живых, реалистичных изображений.

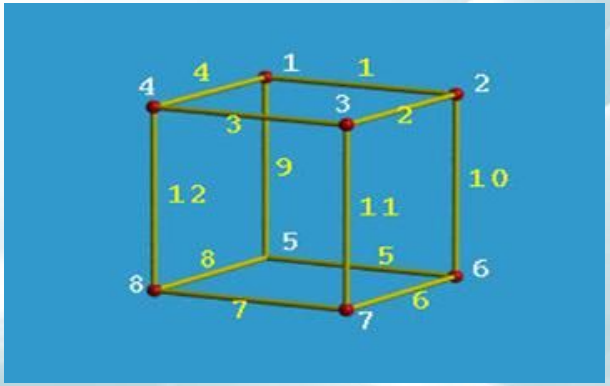
Усилия многих ученых и инженеров во всем мире направлены на разработку методов и средств достижения этой цели. В этом плане наиболее полно ощущается связь компьютерной графики с естественными науками, с дисциплинами, посвященными изучению окружающего нас мира.

Например, для создания реалистичных изображений необходимо учитывать законы оптики, описывающие свет и тень, отражение и преломление. Компьютерная графика находится на стыке многих дисциплин и разделов науки.

2.1. Каркасная визуализация

Каркас обычно состоит из отрезков прямых линий (соответствует многограннику), хотя можно строить каркас и на основе кривых, в частности сплайновых кривых Безье. Все ребра показанные в окне вывода, видны-как ближние, так и дальние.

Для построения каркасного изображения надо знать координаты всех вершин в мировой системе координат. Потом преобразовать координаты каждой вершины в экранные координаты в соответствии с выбранной проекцией. Затем выполнить вывод в плоскость экрана всех ребер как отрезков прямых или кривых, соединяющих вершины.



2.2. Показ с удалением невидимых точек

Мы будем рассматривать поверхности в виде многогранников или полигональных сеток. Известны такие методы показа с удалением невидимых точек: сортировка граней по глубине, метод плавающего горизонта, метод Z-буфера.

Сортировка граней по глубине

Это означает рисование полигонов граней в порядке от самых дальних к ближним. Этот метод не является универсальным, так как иногда нельзя четко различить, какая грань ближе (рис. 26). Известны модификации этого метода, которые позволяют корректно рисовать подобные грани. Метод сортировки по глубине эффективен для показа поверхностей, заданных функциями *z=f(x,y)* (рис.27).

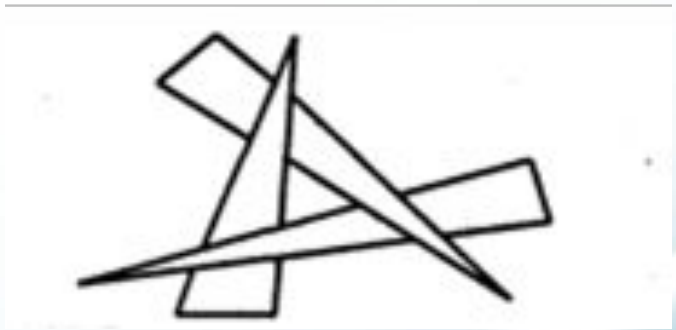


Рис. 26. В каком порядке рисовать эти грани?

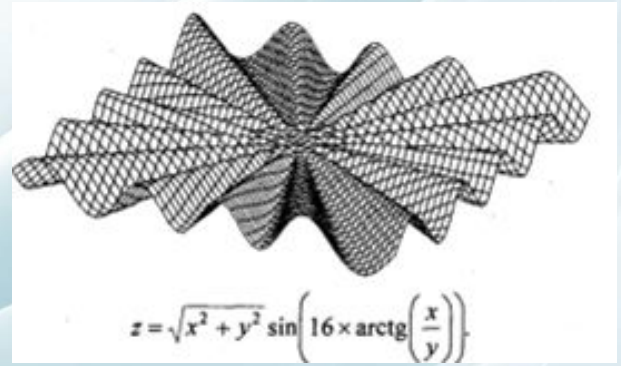
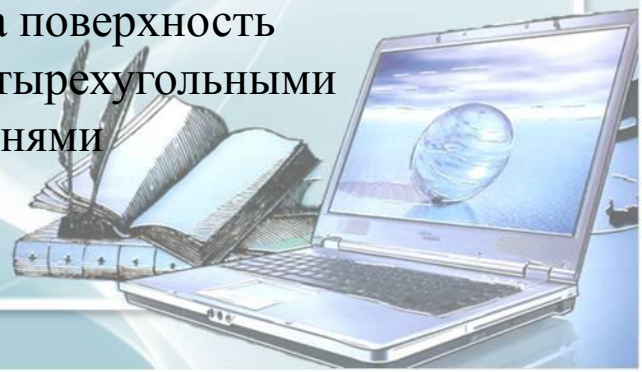


Рис. 27. Эта поверхность нарисована четырехугольными гранями



Метод плавающего горизонта

Здесь, в отличие от предыдущего метода, грани выводятся в последовательности от ближних к самым дальним. На каждом шаге границы граней образуют две ломаные линии — верхний горизонт и нижний горизонт.

В течение вывода каждой новой грани рисуется только то, что выше верхнего горизонта, и то, что ниже нижнего горизонта. Соответственно, каждая новая грань поднимает верхний и опускает нижний горизонты.

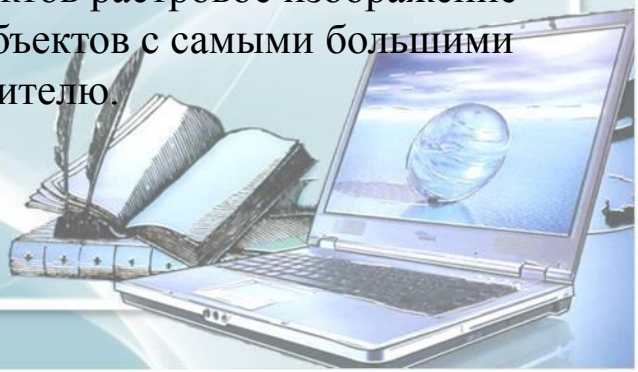
Этот метод часто используют для показа поверхностей, которые описываются функциями $z=f(x,y)$.

Метод Z-буфера

Метод основывается на использовании дополнительного массива, буфера в памяти в котором сохраняются координаты точек Z для каждого пикселя растра. Координата Z соответствует расстоянию точек пространственных объектов до плоскости проецирования. Например, она может быть экранной координатой Z в системе экранных координат (X, Y, Z), если ось Z перпендикулярна плоскости экрана. Рассмотрим алгоритм рисования объектов по этому методу. Пусть чем ближе точка в пространстве к плоскости проецирования, тем больше значение Z:

- сначала Z-буфер заполняется минимальными значениями;
- затем начинается вывод всех объектов, причем порядок вывода объектов не имеет значения – для каждого объекта выводятся все его пиксели в любом порядке;
- во время вывода каждого пикселя по его координатам (X, Y) находится текущее значение Z в Z-буфере;
- если рисуемый пиксель имеет большее значение Z, чем значение в Z-буфере, то, следовательно, эта точка ближе к наблюдателю. В этом случае пиксель действительно рисуется, а его Z-координата записывается в Z-буфер.

Таким образом, после рисования всех пикселей всех объектов растровое изображение будет состоять из пикселей, которые соответствуют точкам объектов с самыми большими значениями координат Z, т. е. видимые точки ближе всех к зрителю.



Этот метод прост и эффективен благодаря тому, что не требует сортировки объектов или их точек. При рисовании объектов, которые описываются многогранниками или полигональными сетками, манипуляции со значениями Z-буфера легко совместить с выводом пикселей заполнения полигонов плоских граней.

В настоящее время метод Z-буфера используется во многих графических 3d-акселераторах, которые аппаратно реализуют этот метод. Оптимально, если акселератор имеет собственную память для Z-буфера, доступ к которой осуществляется быстрее, чем к оперативной памяти компьютера.

Возможности аппаратной реализации используются разработчиками и пользователями компьютерной анимации, позволяя достичь большой скорости прорисовки кадров.

§ 3. Закрашивание поверхностей.

3.1. Метод Фонга

Аналогичен методу Гуро, но при использовании метода Фонга для определения цвета в каждой точке интерполируются не интенсивности отраженного света, а векторы нормалей.

- Определяются нормали к граням.
- По нормалям к граням определяются нормали в вершинах. В каждой точке закрашиваемой грани определяется интерполированный вектор нормали.
- По направлению векторов нормали определяется цвет точек грани в соответствии с выбранной моделью отражения света.

Метод Фонга сложнее, чем метод Гуро. Для каждой точки (пикселя) поверхности необходимо выполнять намного больше вычислительных операций. Тем не менее он дает значительно лучшие результаты, в особенности при имитации зеркальных поверхностей.

Общие черты и отличия методов Гуро и Фонга можно показать на примере цилиндрической поверхности, аппроксимированной многогранником. Пусть источник света находится позади нас.

Проанализируем закрашивания боковых граней цилиндра.

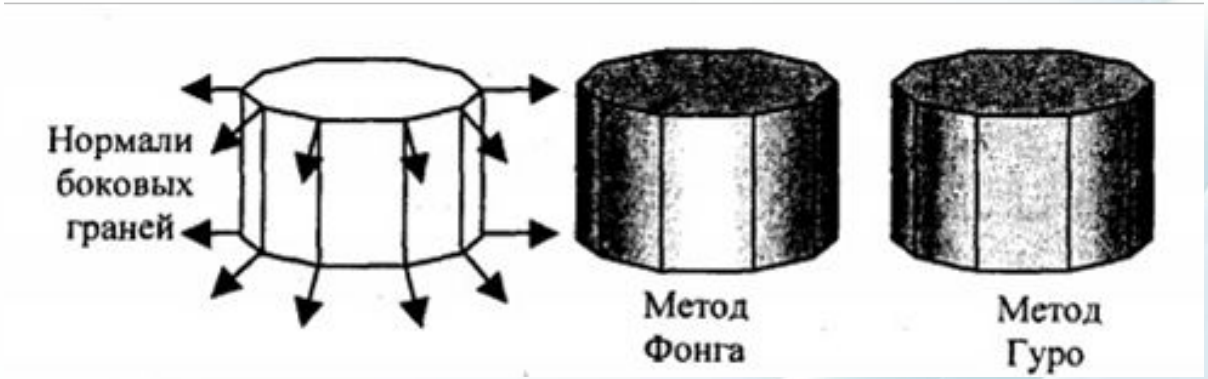


Рис. 28. Отличия закрашивания Фонга и Гуро

На рисунке на закрашенной поверхности показаны черным цветом ребра граней — это сделано для иллюстрации особенностей закрашивания, на самом деле после закрашивания никакого черного каркаса не будет, и поверхность выглядит гладкой.

Лабораторные занятия

№1. 3D max Edit Poly

№2. 3D №2. 3D max№2. 3D max создание стен, окон, дверей

№3. GIF анимация в 3№3. GIF анимация в 3D№3. GIF анимация в 3D max

№4. Интерфейс и примитивы

№5. Моделирование из примитивов

№6. Мягкая мебель 3№6. Мягкая мебель 3D№6. Мягкая мебель 3D max

№7. Освещение в 3№7. Освещение в 3D№7. Освещение в 3D max

№8. Основы анимации в 3№8. Основы анимации в 3D№8. Основы анимации в 3D max

№9. Редактор материалов 3№9. Редактор материалов 3D№9. Редактор материалов 3D max

Практическое задание

Раздел 4.
Алгоритмы вывода фрактальной графики

§1. Понятие фрактала.

§§2§2. Геометрические (конструктивные) фракталы.

§§3§3. Динамические (алгебраические) фракталы.

§4. Стохастические фракталы.

Приложение

§1. Понятие фрактала.

1.1. История

На рубеже XIX и XX веков изучение фракталов носило скорее эпизодический, нежели систематический характер, потому что раньше математики в основном изучали «хорошие» объекты, которые поддавались исследованию при помощи общих методов и теорий.

В 1872 году немецкий математик Карл Вейерштрасс построил пример непрерывной функции, которая нигде не дифференцируема. Однако его построение было целиком абстрактно и трудно для восприятия. Поэтому в 1904 году швед Хельге фон Кох придумал непрерывную кривую, которая нигде не имеет касательной, причем ее довольно просто нарисовать. Оказалось, что она обладает свойствами фрактала. Один из вариантов этой кривой носит название «снежинка Коха».

Идеи самоподобия фигур подхватил француз Поль Пьер Леви, будущий наставник Бенуа Мандельброта. В 1938 году вышла его статья «Плоские и пространственные кривые и поверхности, состоящие из частей, подобных целому», в которой описан еще один фрактал — С-кривая Леви. Все эти вышеперечисленные фракталы можно условно отнести к одному классу конструктивных (геометрических) фракталов.

Другой класс — динамические (алгебраические) фракталы, к которым относится и множество Мандельброта. Первые исследования в этом направлении относятся к началу XX века и связаны с именами французских математиков Гастона Жюлиа и Пьера Фату.

В 1918 году вышел почти двухсотстраничный труд Жюлиа, посвященный итерациям комплексных рациональных функций, в котором описаны множества Жюлиа — целое семейство фракталов, близко связанных с множеством Мандельброта.

Этот труд был удостоен приза Французской академии, однако в нем не содержалось ни одной иллюстрации, так что оценить красоту открытых объектов было невозможно. Несмотря на то что это работа прославила Жюлиа среди математиков того времени, о ней довольно быстро забыли.

Вновь внимание к работам Жюлиа и Фату обратилось лишь полвека спустя, с появлением компьютеров: именно они сделали видимыми богатство и красоту мира фракталов. Ведь Фату никогда не мог посмотреть на изображения, которые мы сейчас знаем как изображения множества Мандельброта, потому что необходимое количество вычислений невозможно провести вручную. Первым, кто использовал для этого компьютер был Бенуа Мандельброт .

В 1982 году вышла книга Мандельброта «Фрактальная геометрия природы», в которой автор собрал и систематизировал практически всю имевшуюся на тот момент информацию о фракталах и в легкой и доступной манере изложил ее.

Основной упор в своем изложении Мандельброт сделал не на тяжеловесные формулы и математические конструкции, а на геометрическую интуицию читателей.

Благодаря иллюстрациям, полученным при помощи компьютера, и историческим байкам, которыми автор умело разбавил научную составляющую монографии, книга стала бестселлером, а фракталы стали известны широкой публике. Их успех среди нематематиков во многом обусловлен тем, что с помощью весьма простых конструкций и формул, которые способен понять и старшеклассник, получаются удивительные по сложности и красоте изображения.

Когда персональные компьютеры стали достаточно мощными то появилось даже целое направление в искусстве — фрактальная живопись, причем заниматься ею мог практически любой владелец компьютера.

Сейчас в интернете можно легко найти множество сайтов, посвященных этой теме.

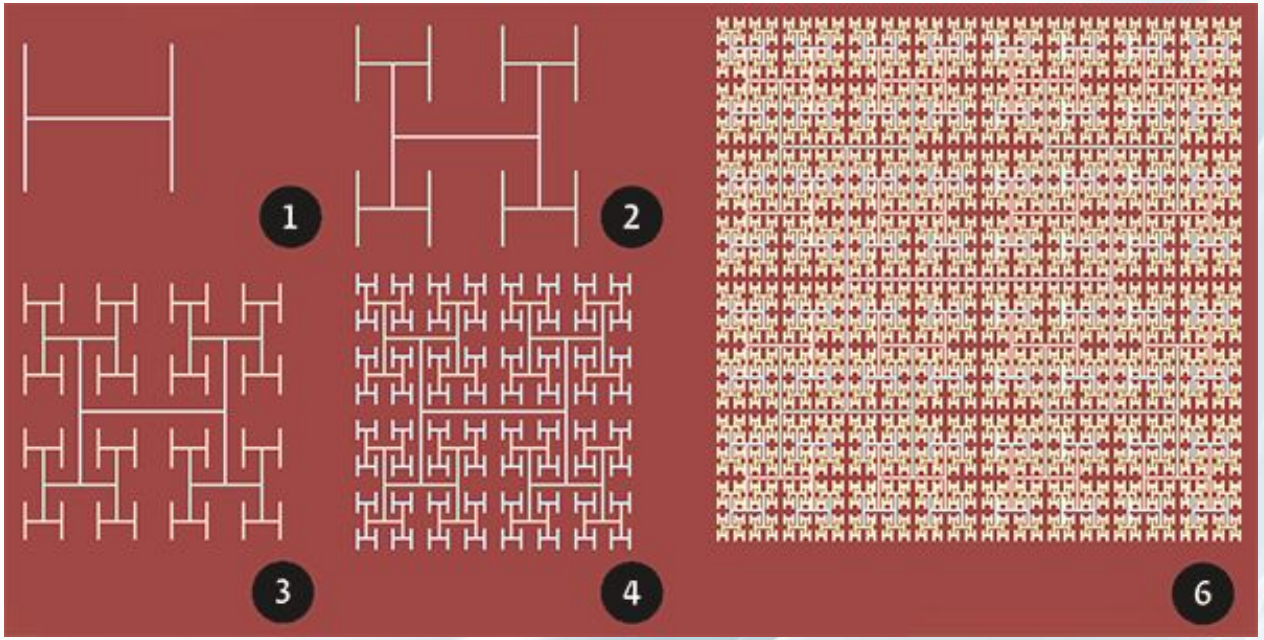
§2. Геометрические (конструктивные) фракталы.

2.1. H-фрактал

Всё начинается с фигуры в виде буквы H, у которой вертикальные и горизонтальные отрезки равны. Затем к каждому из 4 концов фигуры пририсовывается ее копия, уменьшенная в два раза. К каждому концу (их уже 16) пририсовывается копия буквы H, уменьшенная уже в 4 раза. И так далее. В пределе получится фрактал, который визуально почти заполняет некоторый квадрат. H-фрактал всюду плотен в нём. То есть в любой окрестности любой точки квадрата найдутся точки фрактала. Очень похоже на то, что происходит с T-квадратом. Это не случайно, ведь, если присмотреться, видно, что каждая буква H содержится в своем маленьком квадратике, который был дорисован на таком же шаге.

Можно сказать (и доказать), что H-фрактал заполняет свой квадрат (англ. space-filling curve). Поэтому его фрактальная размерность равна 2. Суммарная длина всех отрезков бесконечна.

Принцип построения Н-фрактала применяют при производстве электронных микросхем: если нужно, чтобы в сложной схеме большое число элементов получило один и тот же сигнал одновременно, то их можно расположить в концах отрезков подходящей итерации Н-фрактала и соединить соответствующим образом.



Если рисовать толстые буквы Н, состоящие из прямоугольников, а не из отрезков, то получится Дерево Мандельброта.

§3. Динамические (алгебраические) фракталы.

3.1. Множество Жюлиа

В множестве Жюлиа любая точка z комплексной плоскости имеет свой характер поведения (остается конечной, стремится к бесконечности, принимает фиксированные значения) при итерациях функции $f(z)$, а вся плоскость делится на части. При этом точки, лежащие на границах этих частей, обладают таким свойством: при сколь угодно малом смещении характер их поведения резко меняется (такие точки называют точками бифуркации). При этом множества точек, имеющих один конкретный тип поведения, а также множества бифуркационных точек часто имеют фрактальные свойства. Это и есть множества Жюлиа для функции $f(z)$.

Опишем в общих чертах процедуру рисования множества Жюлиа многочлена $z^2 + c$ для конкретного значения комплексного параметра $c = p + iq$.

Будем считать, что экран прямоугольный и состоит из a × b точек и что изображение будет покрашено в K + 1 цвет (то есть цвета пронумерованы от 0 до K, причем цвет номер 0 — черный, а для других цветов условимся, что чем больше номер цвета, тем быстрее «убегает на бесконечность» точка, которую мы покрасим в этот цвет). Еще необходимо выбрать область плоскости, которую выведем на экран (для начала подойдет квадрат {|Re z| ≤ 1,5, |Im z| ≤ 1,5}; его нужно разрезать на a × b прямоугольников, каждый из которых будет выступать в роли точки экрана), и радиус R круга D, точки снаружи которого будем считать «бесконечно далекими» (можно взять R = 10).

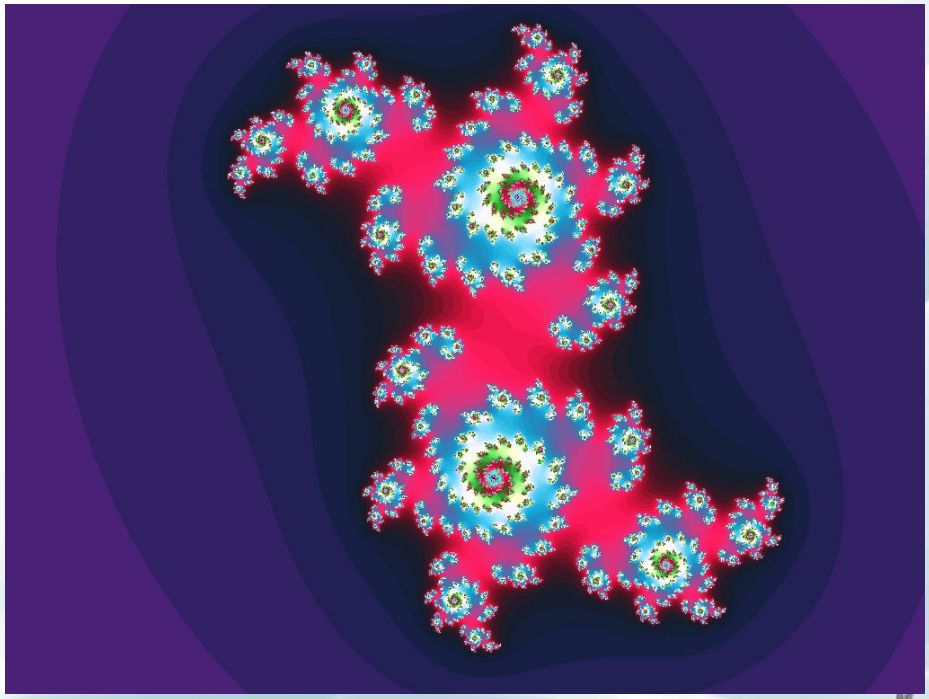
Для каждой точки $z_0 = (x_0 ; y_0)$ экрана (то есть центра соответствующего прямоугольника) нужно в цикле последовательно вычислять z_{k+1} по z_k, используя формулу

$$z_{k+1} = z_k^2 + c$$

(в координатах это выглядит так: $x_{k+1} = x_k^2 - y_k^2 + p$, $y_{k+1} = 2x_k y_k + q$). Признаком остановки цикла является выполнение условий: либо на k-м шаге точка z_k вышла из круга D (то есть верно неравенство $|z_k| = \sqrt{x_k^2 + y_k^2} > R$), и тогда точку z_0 нужно покрасить в цвет номер k, либо оказалось, что считаем, что точка z_0 лежит внутри множества Жюлиа, и красим ее в черный.

В результате работы программы на экран будет выведена квадратная область комплексной плоскости $\{|Re\ z| \le 1{,}5,\ |Im\ z| \le 1{,}5\}$, на которой черным цветом будет изображено множество Жюлиа многочлена $z^2 + c$ для выбранного параметра $c = p + iq$, а остальные точки будут раскрашены в K цветов.

Увеличивая числа a и b, можно повышать разрешение экрана и тем самым улучшать качество изображения. Меняя K и подбирая соответствие между цветами и их номерами, можно добиться довольно красивых картинок.



3.2. Фрактал Ньютона

Бассейны Ньютона, фракталы Ньютона — разновидность алгебраических фракталов. Области с фрактальными границами появляются при приближенном нахождении корней нелинейного уравнения алгоритмом Ньютона на комплексной плоскости (для функции действительной переменной метод Ньютона часто называют методом касательных, который, в данном случае, обобщается для комплексной плоскости).

Формулы для их построения основаны на методе решения нелинейных уравнений, который был придуман великим математиком еще в XVII веке.

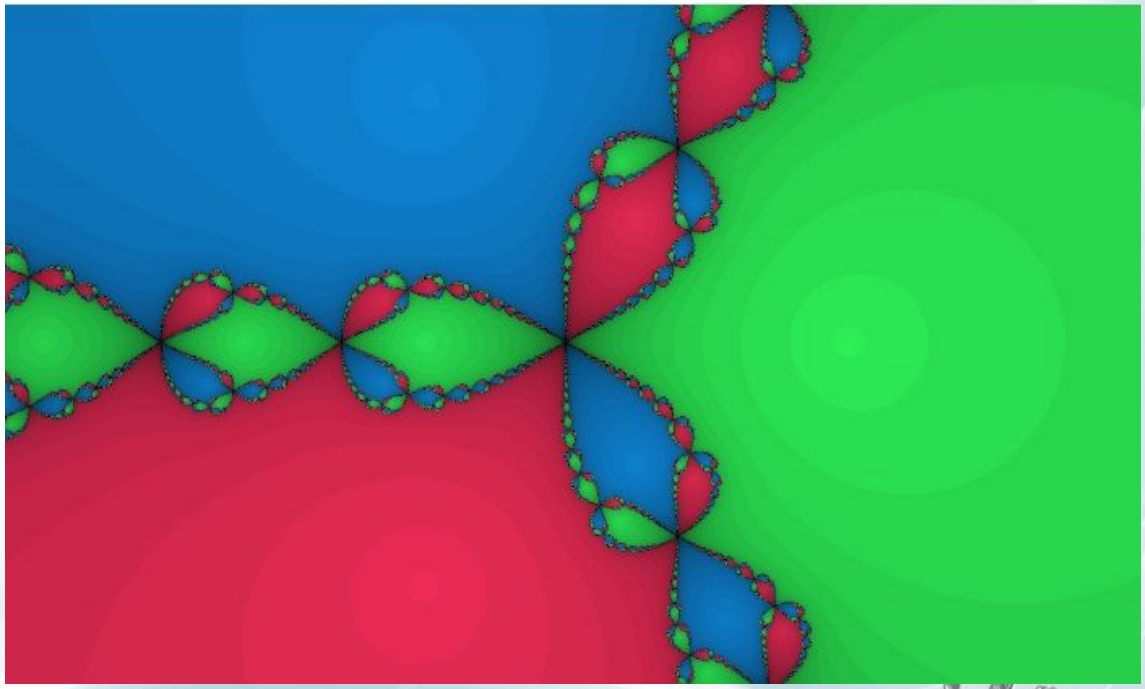
Применяя общую формулу метода Ньютона

$$z_{n+1} = z_n - f(z_n)/f'(z_n),\ n = 0, 1, 2, \ldots$$

для решения уравнения $f(z) = 0$ к многочлену $z^k - a$, получим последовательность точек:

$$z_{n+1} = ((k - 1)zn_k - a)/kzn_k - 1, n = 0, 1, 2, \ldots .$$

Выбирая в качестве начальных приближений различные комплексные числа z_0, будем получать последовательности, которые сходятся к корням этого многочлена. Поскольку корней у него ровно k, то вся плоскость разбивается на k частей — областей притяжения корней. Границы этих частей имеют фрактальную структуру. (Заметим в скобках, что если в последней формуле подставить $k = 2$, а в качестве начального приближения взять $z_0 = a$, то получится формула, которую реально используют для вычисления квадратного корня из a в компьютерах.) Наш фрактал получается из многочлена $f(z) = z^3 - 1$.



§4. Стохастические фракталы

Еще одним известным классом фракталов являются стохастические фракталы, которые получаются в том случае, если в итерационном процессе случайным образом менять какие-либо его параметры. При этом получаются объекты очень похожие на природные - несимметричные деревья, изрезанные береговые линии и т.д. Двумерные стохастические фракталы используются при моделировании рельефа местности и поверхности моря.

Фракталы, при построении которых случайным образом изменяются какие-либо параметры, называются стохастическими. Термин “стохастичность” происходит от греческого слова, обозначающего “предположение”.

Стохастическим природным процессом является броуновское движение. С помощью компьютера такие процессы строить достаточно просто: надо просто задать последовательности случайных чисел и настроить соответствующий алгоритм. При этом получаются объекты, очень похожие на природные, — несимметричные деревья, изрезанные береговые линии и т.д. Двумерные стохастические фракталы используются при моделировании рельефа местности и поверхности моря

С помощью компьютерной программы можно построить какие-нибудь объекты живой природы, например, ветку дерева. Процесс конструирования этого геометрического фрактала задается более сложным правилом, нежели построение вышеописанных кривых.

Рассмотрим его на примере ветки. Всего лишь несколько шагов в компьютерном алгоритме и мы видим, как образуется ветка-фрактал (рис.29).

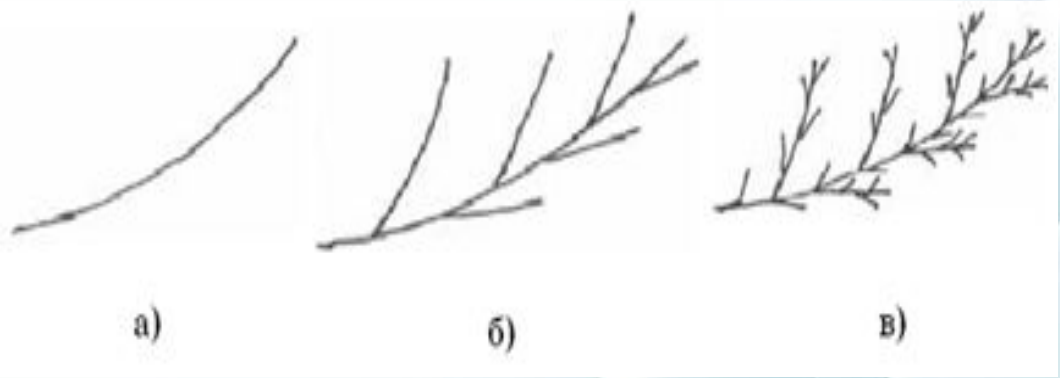


Рис. 29.Фрактал ветка