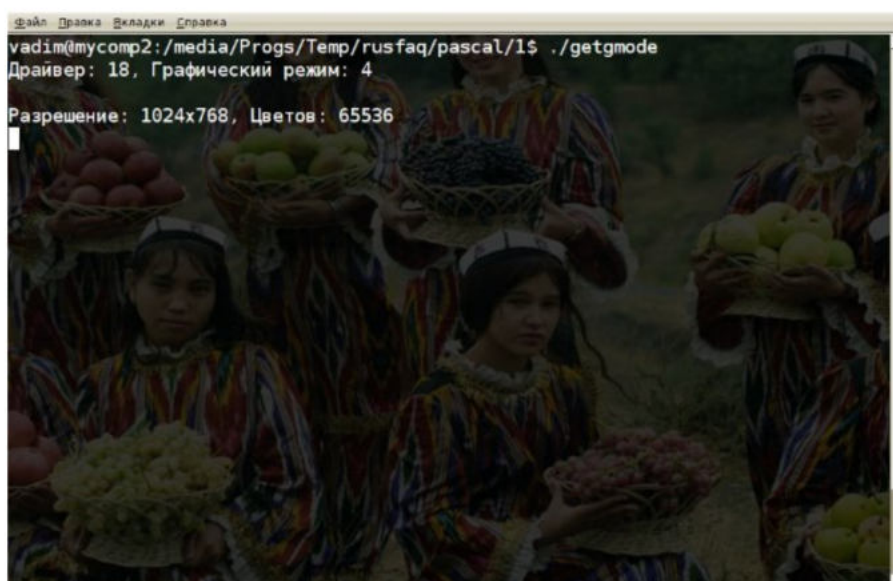


## Слово об использовании модуля Graph в Linux

 [freepascal.ru/article/freepascal/20120215095658](http://freepascal.ru/article/freepascal/20120215095658)



[Публикации FreePascal](#)

15.02.2012

Вадим Исаев

Те, кто изучает Pascal по книгам с описанием TurboPascal, дойдя до примеров работы с графикой и попытавшись перенести их в Linux, обычно сталкиваются с проблемой, когда эти программы отказываются компилироваться, выдавая какие то дурацкие сообщения, вроде этого:

```
/usr/bin/ld: cannot find -lvga
Error: error while linking
```

Скажу сразу - это линковщик (т.е. программа, которая собирает после компиляции все модули в единое целое и приделывает в самом начале правильный заголовок-запускатель) жалуется на отсутствие графической библиотеки. **"Cannot find"** - "Не могу найти", далее **"-l"** - это опция подключения внешней библиотеки, после чего идёт название самой библиотеки. А те же самые программы, при компиляции их в Windows, компилируются без проблем. Почему? Тут без исторического экскурса не обойтись.

### Путешествие в далёкое прошлое

Когда создавалась операционная система Unix, на потомке которой мы с вами сейчас работаем, компьютеры были чрезвычайно малоохотными и падали в обморок от малейшей перегрузки. На первом компьютере, его звали "Эниак", даже не было клавиатуры для ввода данных и монитора для просмотра результатов вычислений. Данные в него вводили с помощью переключателей (либо 0, либо 1), а результаты считывали с ряда лампочек (если лампочка горит - это 1, если не горит - это 0). Печальна и тяжела была работа с таким компьютером. Я это знаю по себе, т.к. у меня в детстве была вычислительная машина, наподобие "Эниак" - звалась она ДВМ-2 и программировалась проволоочными перемычками, а ответ выводила на четыре разноцветные лампочки.

Когда данных немного, такие убогие средства ввода-вывода ещё можно было пережить, но а если данных очень много? Тогда вводить данные удобно теми символами, к которым привык человек, благо образец уже был - печатная машинка. По её образцу и подобию сделали клавиатуру.

А вот как выводить? Опять же, глядя на печатную машинку, весь вывод сначала распечатывали на бумаге. Вот только не всегда программы были сразу правильными, поэтому тонны бумаги уходили в мусор. Умные головы довольно быстро обратили внимание на телевизор - а что если весь мусор сначала печатать на телеэкране и только правильные ответы выдавать на бумагу? Сказано - сделано,

припаяли к компьютеру телевизор. Все радуются, что удалось сохранить леса от неправильных вычислений.

Конечно, немедленно стали думать, а как на компьютере показывать картинки? И тут же поняли, чем компьютер отличается от телевизионной студии. Телестудия передаёт в эфир уже готовую картинку, а компьютер должен её сначала сконструировать внутри себя и только потом уже показывать. А ни памяти, ни быстродействия на это не хватает. Если с генерацией букв и цифр компьютер справлялся более-менее успешно, то с картинками, которые надо вырисовывать по точкам, уже нет. Поэтому очень долго компьютеры изображали только текст, либо псевдографику, используя для неё подходящие символы.

Становление и развитие Unix пришлось как раз на этот период, поэтому основной упор в ней делался на отображение и передачу текста. Но всё меняется, в том числе мощность процессоров и объём оперативной памяти. И вот появилась возможность генерации более-менее нормальной графики. Многие учёные-компьютерщики стали думать, как это делать в уже готовой ОС. Конечно придумали. И получилось так, что каждый коллектив предлагал что-то своё. Поэтому в Unix, в отличие от Windows, нет какой-то одной стандартной графической библиотеки. Их много. Как говорить - выбирай на любой вкус.

### Начинаем графититься...

---

И вот однажды я задался вопросом - каким образом можно рисовать в Linux? Нашёл, что модуль **Graph** присутствует. Это меня воодушевило и я немедленно попытался скомпилировать простейшее приложение с рисованием линий, кружочков и т.п. И получил сообщение, которое привёл в начале. Порылся в исходниках **FreePascal** и увидел, что графических модулей, оказывается, много, видимо по числу графических библиотек. Модулей обнаружилось ажно четыре (речь идёт только о совместимых с **TurboPascal**'евским **Graph**):

- Graph,
- ggiGraph,
- ptcGraph,
- sdlGraph.

Посмотрев исходники этих модулей я понял, что все они предназначены для совмещения стандартных графических процедур и функций, которые были в **TurboPascal**'евском модуле **Graph**, с теми функциями, которые присутствуют непосредственно в графических библиотеках. Увы, непосредственно модуль **Graph** для книжных примеров с графикой в современных операционных системах абсолютно не годится. Причин этому несколько:

- Графическая библиотека **svglib**, которая требуется для работы модуля **Graph**, уже давным давно устарела и не развивается больше десяти лет, поэтому сия библиотека по умолчанию не устанавливается.
- Самая главная проблема (если вы, всё же, установили эту библиотеку вручную) заключается в том, что для своей работы эта библиотека требует прав **root**'а и от обычного пользователя, под которым все, обычно, работают в **Linux**, не будет работать.  
Интересно, почему? Оказывается, для своей работы она требует выделения отдельной консоли. Естественно, обычный пользователь этого сделать не может.

Итак, модуль **Graph** мы отмечаем и попробуем найти его работающий и лёгкий в использовании аналог. Как я написал выше, таких аналогов целых три, у аждого из которых своя собственная графическая библиотека. Не буду описывать процесс испытаний. В конце концов, самым бесппроблемным в использовании оказался модуль **ptcGraph**.

### Что нужно сделать для работы ptcGraph?

---

Для работы этого модуль требует две библиотеки:

- **Xxf86vm** (файл **libXxf86vm.so**).
- **Xxf86dga** (файл **libXxf86dga.so**).

Ищем их в каталогах:

- `/usr/lib/`,
- `/lib/`.

У меня (ОС **Runtu 10.04**) Обе обнаружили уже установленными, однако содержали в своём названии ещё всякие цифры (по всей видимости версии библиотеки). Если у вас их нет, то установите из репозитория вашей операционной системы. Далее нужно сделать симлинки с правильным названием:

```
ln -s /usr/lib/libXxf86vm.so.1.0.0 /usr/lib/libXxf86vm.so
ln -s /usr/lib/libXxf86dga.so.1.0.0 /usr/lib/libXxf86dga.so
```

Эти действия нужно проводить с правами root.

### Разные мелочи, без которых никак не обойтись

Самая наипервейшая мелочь касается инициализации графического режима с помощью процедуры **InitGraph(Driver, Mode, ...)**. Драйвер **Driver** во всех книжках по **TurboPascal** рекомендуют назначать с помощью макроса **DETECT**, заодно и автоматом назначается графический режим - разрешение и цветность:

```
Driver:=DETECT;
...
```

**!!!Крайне не рекомендую так делать!!!**. Дело в том, что этот макрос выберет максимально возможное разрешение для вашей видеокарточки, но отнюдь не для вашего монитора. Сейчас ситуация такая, что любая, даже самая дешёвая видеокарта, может без труда выдать разрешение хоть 2048x1280. А вот поддерживает ли ваш монитор такое? Скорее всего нет и вы, вместо картинки, увидите пустой чёрный экран. Графический режим надо обязательно назначать вручную и в тех пределах, которые поддерживает ваш монитор. В этом главный недостаток досовских графических программ - невозможность автоматом принять тот режим, который уже есть в системе.

Однако, не всё так печально. Процедура:

```
DetectGraph(var Driver: SmallInt; var Mode: SmallInt);
```

даёт нам более правильные данные о драйвере и графическом режиме. Графический режим можно включать так:

```
Var
    Driver: SmallInt; // Номер драйвера
    Mode:   SmallInt; // Номер графического режима
    ...
Begin
    DetectGraph(Driver, Mode);
    InitGraph(Driver, Mode, '');
    ...
End.
```

Получается довольно симпатичное окошко на всю высоту экрана. Однако было бы интересно узнать, какое разрешение и какое количество цветов имеет этот полученный по-умолчанию графический режим. Напишем для этого небольшую программку:

```

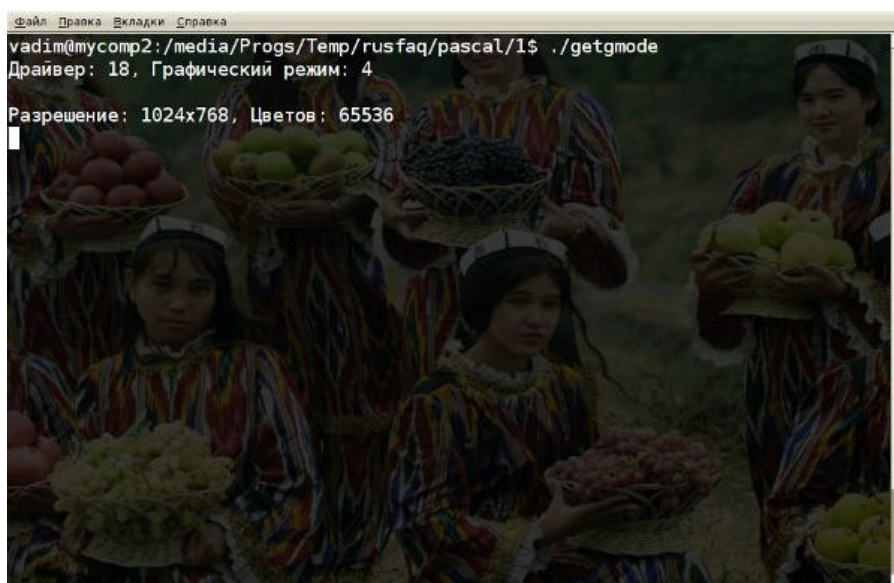
Program getgmode;
Uses ptcGraph;

Var
    Driver, Mode: SmallInt;

Begin
    DetectGraph(Driver, Mode);
    WriteLn('Драйвер: ', Driver, ', Графический режим: ', Mode);
    ReadLn;
    InitGraph(Driver, Mode, '');
    WriteLn('Разрешение: ', GetMaxX+1, 'x', GetMaxY+1, ', Цветов: ', GetMaxColor+1);
    ReadLn;
    CloseGraph;
End.

```

Вот, что показывает программа:



Однако для примеров из книжек по **TurboPascal** это окошко будет иметь чрезмерно высокое разрешение, а главное, сильно большое количество цветов. Вы, скорее всего, в этом окошке ничего не увидите из того, что хотели нарисовать в программе. Для всех книжных примеров необходимо подставлять драйвер **"VGA"**, а графический режим **"VGAHi"**:

```

Var
    Driver: SmallInt; // Номер драйвера
    Mode: SmallInt; // Номер графического режима
    ...

Begin
    Driver:=VGA;
    Mode:=VGAHi;
    InitGraph(Driver, Mode, '');
    ...

End.

```

и, таким образом, в графическом окошке будете иметь тот режим, на который рассчитывали авторы книжных примеров - разрешение 640x480 и 16 стандартных цветов. Если же хотите иметь разрешение и цветность больше тех, что имел когда-то **TurboPascal**, то номера цветов вам придётся подбирать вручную, руководствуясь максимальным количеством цветов того графического режима, который вы выбрали.

Исходя из вышеизложенного неплохо было бы узнать, а какие графические режимы поддерживает видеоадаптер. Для этого существует функция:

```
QueryAdapterInfo: PModeInfo;
```

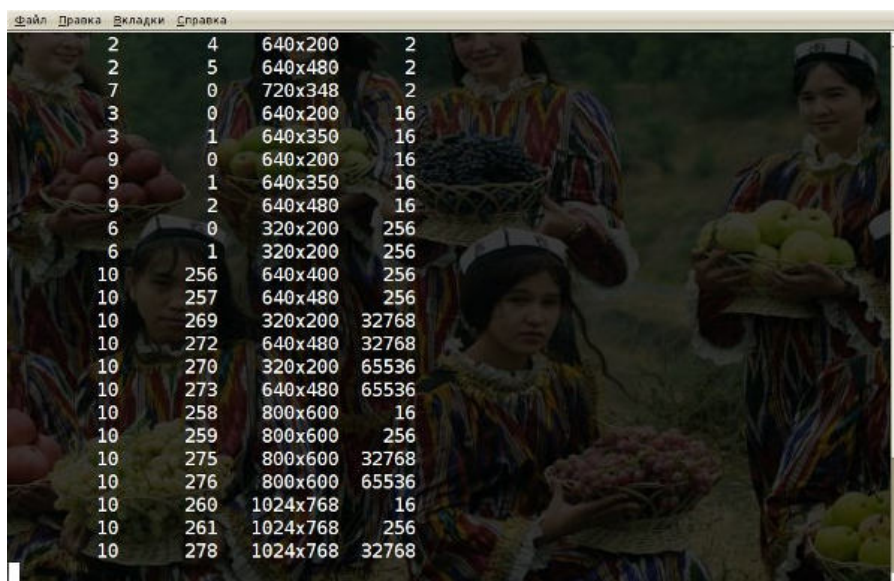
которая выдаёт связанный список всех поддерживаемых видеоадаптером графических режимов с их подробным описанием. Давайте составим программку, которая нам расскажет, чем богат видеоадаптер:

```
Program gmodeinfo;
Uses ptcGraph, SysUtils;

Var
  ModeInfo: PModeInfo; // Сюда будет заносится информация о видеорежимах
  Rez: String;

Begin
  ModeInfo:=QueryAdapterInfo;
  If ModeInfo=NIL Then
    WriteLn('Не удалось получить информацию у видеоадаптера...')
  Else
    Begin
      WriteLn('№ драйвера ', '№ режима ', 'Разрешение ', 'Цветов ');
      WriteLn('-----');
      While ModeInfo^.Next<>NIL Do
        Begin
          Write(ModeInfo^.DriverNumber:10);
          Write(ModeInfo^.ModeNumber:9);
          // Write(' '+ModeInfo^.ModeName+' ':22);
          Rez:=IntToStr(ModeInfo^.MaxX+1)+'x'+IntToStr(ModeInfo^.MaxY+1);
          Write(Rez:11);
          WriteLn(ModeInfo^.MaxColor:7);
          ModeInfo:=ModeInfo^.Next;
        End;
      ReadLn;
    End;
End.
```

Вот, что показывает программа:



|    |     |          |       |
|----|-----|----------|-------|
| 2  | 4   | 640x200  | 2     |
| 2  | 5   | 640x480  | 2     |
| 7  | 0   | 720x348  | 2     |
| 3  | 0   | 640x200  | 16    |
| 3  | 1   | 640x350  | 16    |
| 9  | 0   | 640x200  | 16    |
| 9  | 1   | 640x350  | 16    |
| 9  | 2   | 640x480  | 16    |
| 6  | 0   | 320x200  | 256   |
| 6  | 1   | 320x200  | 256   |
| 10 | 256 | 640x400  | 256   |
| 10 | 257 | 640x480  | 256   |
| 10 | 269 | 320x200  | 32768 |
| 10 | 272 | 640x480  | 32768 |
| 10 | 270 | 320x200  | 65536 |
| 10 | 273 | 640x480  | 65536 |
| 10 | 258 | 800x600  | 16    |
| 10 | 259 | 800x600  | 256   |
| 10 | 275 | 800x600  | 32768 |
| 10 | 276 | 800x600  | 65536 |
| 10 | 260 | 1024x768 | 16    |
| 10 | 261 | 1024x768 | 256   |
| 10 | 278 | 1024x768 | 32768 |

В первом столбце номер драйвера, во втором - номер графического режима, в третьем - разрешение, в четвёртом - количество цветов. Как видите, кроме старых номеров графического режима, которые были ещё в **TurboPascal** и которые имеют номер меньше 10, есть ещё группа режимов, номер у которых больше 100 и которые, судя по разрешению и цветности, более подходят для современных программ. Это так называемые **VESA**-режимы.

## Историческая справка

Аббревиатура **VESA** расшифровывается как Ассоциация Стандартизации ВидеоЭлектроники (**Video Electronics Standards Association**). В конце прошлого века, когда выпускаемые видеокарточки по своим возможностям стали далеко обгонять принятые на тот момент графические стандарты **CGA**, **EGA** и **VGA**, среди разработчиков компьютерных игр возник, мягко говоря, бардак. Чтобы использовать в игре полностью возможности той или иной видеокарточки, необходимо было приобрести программу-драйвер для этой видеокарты. Однако чуть ли не с каждой неделей всяких разных названий видеокарт становилось всё больше и больше, т.к. спрос на компьютеры был чрезвычайно велик и в результате возникла такая ситуация, когда вы запускаете игрушку, вам предлагается в меню выбрать установленную видеокарту, а в 99% случаев в выданном списке вашей видеокарты даже рядом не лежало. И вам приходится выбирать давно устаревший режим **VGA**, в котором вид картинок вызывал, мягко говоря, одно сплошное уныние...

И вот почтенные джентельмены из организации **VESA** решили разработать и внедрить следующий стандарт - **SVGA** или **SuperVGA**, т.е. для режимов, которые имеют более высокие характеристики, чем **VGA**. Скажу сразу - им это удалось. Были разработаны некие стандартные видеофункции с одними и теми же названиями и параметрами, задавая которые, можно было получить набор графических режимов SVGA. С некоторым скрипом, но эти функции постепенно стали поддерживать и все разработчики видеокарт, закладывая их в **VideoBIOS**, т.е. программу управления видеоадаптером.

Таким образом удалось подстрелить сразу двух вальдшнепов - разработчики игр пользовались только стандартными функциями **VESA** и их игры работали на любой видеокарте без исключения. Продажи игр резко поползли в гору, одновременно вырос и спрос на новейшие видеоадаптеры, которыми стали заменять морально устаревшие.

В общем, бабла на этом деле нарубили просто немеряно...

## Поговорим о цветах...

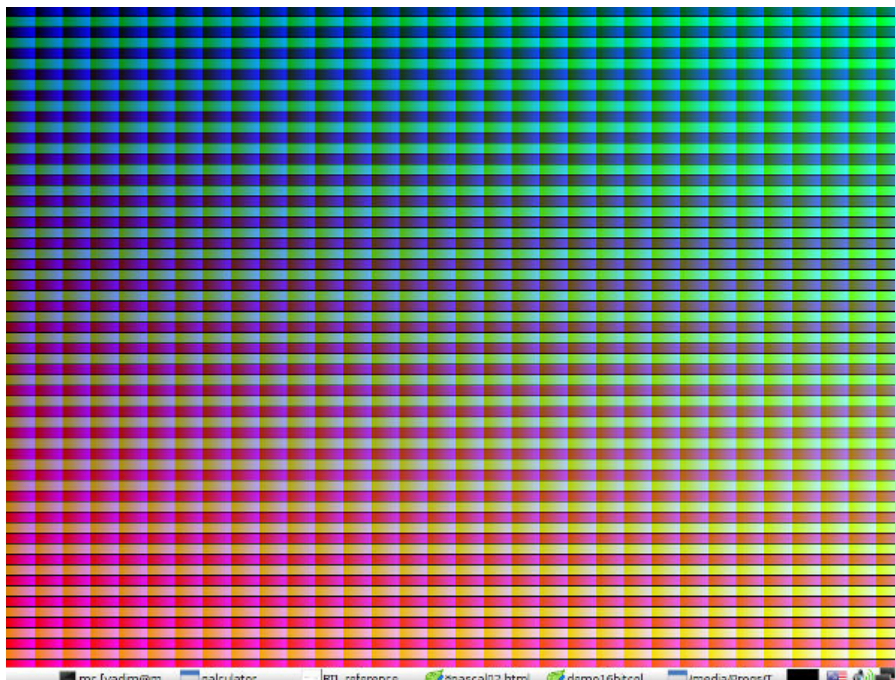
В информации, выданной моим видеоадаптером, написано, что он может отображать 65536 цветов. Было бы крайне любопытно взглянуть, как выглядит цвет сидящий под тем или иным номером. Для этого была составлена следующая программка, в которой последовательно рисуются 65536 квадратиков каждый из которых имеет номер цвета от 0 до 65535:

```
Program demo16bitcolor;
Uses ptcGraph;

Var
  Driver: SmallInt; // Номер драйвера
  Mode : SmallInt; // Номер графического режима
  x, y, i, Color : Integer; // Координаты

Begin
  // Установка графического режима
  DetectGraph(Driver, Mode);
  InitGraph(Driver, Mode, '');
  Color:=0;
  y:=0;
  For i:=0 To 63 Do
    Begin
      For x:=0 To 1023 Do
        Begin
          SetColor(Color);
          Line(x, y, x, y+10);
          Inc(Color);
        End;
        y:=y+12;
      End;
      ReadLn;
      CloseGraph;
    End.
  End.
```

Получается вот такая штука:



Не сказал бы, что зависимость цвета от его номера получилась интуитивно понятной, но это один из недостатков старой "досовской" графики - используется не полный диапазон RGB-значений (**R** - красный, **G** - зелёный, **B** - синий), а только усечённый вариант, т.е. то, что влезло в 16 бит. Тем не менее, это хорошая иллюстрация цветогенерации.

Наверняка вы заметили, что в представленной палитре отсутствуют кое-какие цвета. Это дело можно поправить назначением определённому номеру цвета того цвета, который вам нравится:

```
SetRGBPalette(NumColor, Red, Green, Blue: SmallInt);
```

здесь:

- **NumColor** - номер цвета, начиная от 0 и заканчивая максимальным, для выбранного вами режима,
- **Red, Green, Blue** - соответственно, красная, зелёная и синяя составляющая цвета. Значения от 0 до 255.

### Тоже мелочи, но уже неприятные

Вы наверняка уже успели заметить, что для отображения картинок создаётся второе, графическое окно. Если по ходу программы вам нужно всего лишь показать рисунок или какую-нибудь анимацию, то это было бы и ничего, но вот как только вы захотите интерактивно поуправлять вашим рисунком, то сразу выясняется большая проблема - рисунок отображается в одном окне, а программа ждёт ваши нажатия на клавиши в другом, которое осталось позади. Неудобно.

Как это исправить? Для этого есть модуль **ptcCrt**, который перенаправляет нажатия клавиш в графическое окно. Таким образом, если в своей программе с графикой вы используете управление клавишами, то в строку **Uses** нужно включить модуль **ptcCrt**.

### Изображаем художества на экране. Программа, не особо полезная, но что-то рисующая...

Давайте и мы с вами нарисуем что-нибудь не особо сложное, но красивое. В книге И.А. Бабушкина+соавторы "Практикум по Турбо Паскалю" я нашёл идею этой простой, но весьма интересной программки:

```
Program TriaFractal;
Uses ptcGraph, ptcCrt;

Type
  TTriaPoints = array[0..3] of PointType; // Координаты для рисования треугольника

// Рекурсивная процедура для рисования треугольников
// внутри заданного треугольника
// Параметры:
//   Points - координаты внешнего треугольника
//   N - уровень вложенности процедуры
Procedure Triangle1(Points: TTriaPoints; N: Integer);
Var
  Points1: TTriaPoints;
Begin
  If N>0 Then
    Begin
      Delay(200);
      SetColor(Random(14)+1); // Случайный цвет рисования. Чёрный нам, естественно, не нужен

      // Вычисление новых координат треугольника
      Points1[0].X:=(Points[1].X+Points[0].X) div 2;
      Points1[0].Y:=(Points[1].Y+Points[0].Y) div 2;
      Points1[1].X:=(Points[2].X+Points[0].X) div 2;
      Points1[1].Y:=(Points[2].Y+Points[0].Y) div 2;
      Points1[2].X:=(Points[1].X+Points[2].X) div 2;
      Points1[2].Y:=(Points[1].Y+Points[2].Y) div 2;
      Points1[3].X:=Points1[0].X;
      Points1[3].Y:=Points1[0].Y;

      DrawPoly(4, Points1); // Рисование треугольника

      Triangle1(Points1, N-1); // Рекурсивный вызов для рисования внутри
    End;
  End;

Var
  Driver: SmallInt; // Номер драйвера
  Mode : SmallInt; // Номер графического режима
  Points: TTriaPoints;

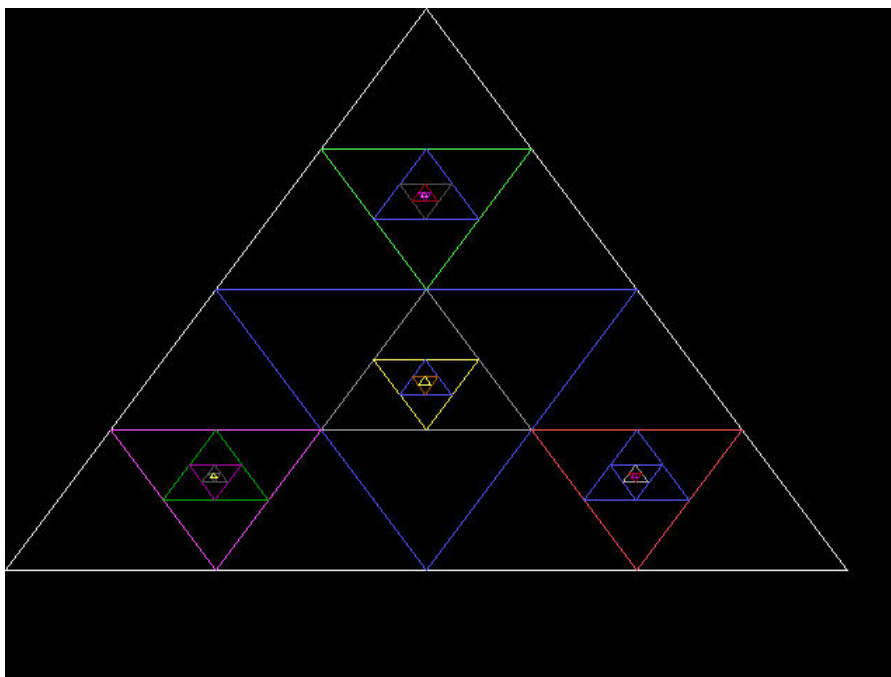
Begin
  Randomize;
  // Установка графического режима
  Driver:=VGA;
  Mode:=VGAHi;
  InitGraph(Driver, Mode, '');

  // Большой треугольник
  Points[0].X:=0;
  Points[0].Y:=400;
  Points[1].X:=600;
  Points[1].Y:=400;
  Points[2].X:=300;
  Points[2].Y:=0;
  Points[3].X:=Points[0].X;
  Points[3].Y:=Points[0].Y;
  DrawPoly(4, Points);
  Triangle1(Points, 6);

  // Верхний треугольник
  Points[0].X:=150;
  Points[0].Y:=200;
  Points[1].X:=450;
  Points[1].Y:=200;
  Points[2].X:=300;
  Points[2].Y:=0;
```

```
Points[3].X:=Points[0].X;  
Points[3].Y:=Points[0].Y;  
Triangle1(Points, 6);  
  
// Левый треугольник  
Points[0].X:=0;  
Points[0].Y:=400;  
Points[1].X:=300;  
Points[1].Y:=400;  
Points[2].X:=150;  
Points[2].Y:=200;  
Points[3].X:=Points[0].X;  
Points[3].Y:=Points[0].Y;  
Triangle1(Points, 6);  
  
// Правый треугольник  
Points[0].X:=300;  
Points[0].Y:=400;  
Points[1].X:=600;  
Points[1].Y:=400;  
Points[2].X:=450;  
Points[2].Y:=200;  
Points[3].X:=Points[0].X;  
Points[3].Y:=Points[0].Y;  
Triangle1(Points, 6);  
  
ReadLn;  
CloseGraph;  
End.
```

Вот такой у меня получился космический корабль пришельцев:



## Post Scriptum

Этот способ рисования не сильно пригоден для повседневного использования, тем более, если вы собираетесь писать полезные для других людей программы. Он годится только для тех случаев, когда вы захотите проверить в работе какие-либо программки из книжек по **TurboPascal**. Графические оболочки операционных систем предоставляют собственные средства рисования графических примитивов, навряде точек, линий, прямоугольников, эллипсов и т.п. Плюс, к этому, предлагаются более удобные функции по работе с цветом, основанные на стандарте RGB. Профессиональные графические библиотеки, такие как OpenGL, предлагают ещё более крутые средства по работе с графикой и тоже на основе той графики, которая уже есть в графической оболочке вашей ОС. Наиболее лёгкий способ

использовать графику - перейти к использованию **Lazarus**.