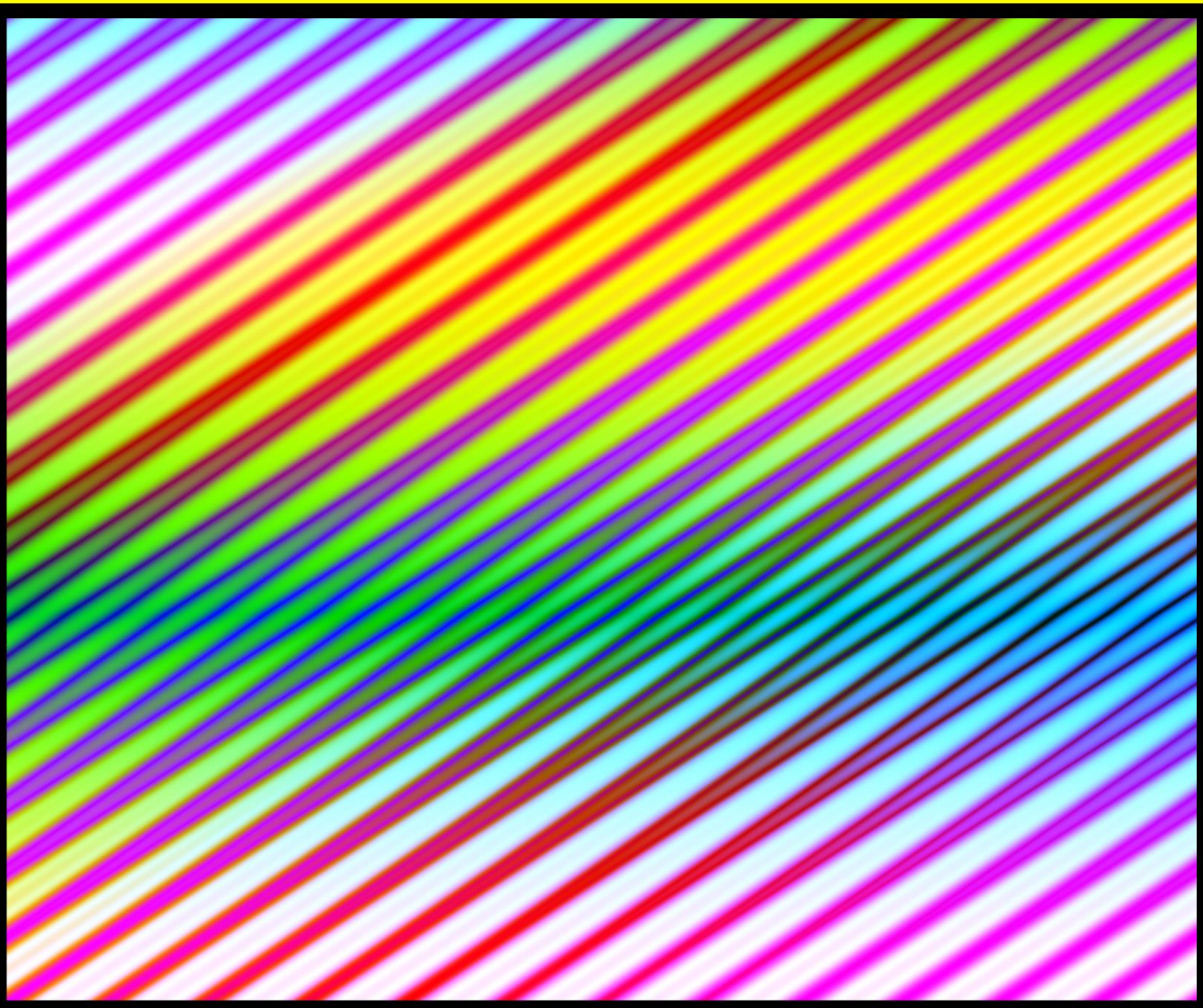
Рубанцев Валерий

Программирование на *паскале*

Графика для компьютерных игр



SFML

PascalABC.NET

Валерий Рубанцев

Графика для компьютерных игр

С примерами на *паскале* и *SFML*

От автора

Невозможно представить современные компьютерные игры без качественной и быстрой графики. В программах на *паскале* доступна графика GDI+. Её вполне достаточно для разработки «статических» игр типа головоломок или шашек. Для динамических игр с визуальными эффектами такая графика слишком медленная.

Для ускорения графики мы будем использовать библиотеку SFML (*Simple and Fast Multimedia Library*; по-русски: *простая и быстрая мультимедийная библиотека*). Её вполне достаточно для написания 2D-игр. Она, действительно, очень простая по сравнению с игровыми движками, но работает быстро и позволяет добавлять в программу звуки и музыку.

Цель книги: изучить основные графические возможности библиотеки SFML, чтобы затем пользоваться ею для разработки компьютерных игр. Более того, эта библиотека может успешно использоваться для программирования компьютерной графики, что также может пригодиться вам и при разработке разнообразных программ.

Книга адресуется: школьникам, изучающим *PascalABC.NET* на уроках или самостоятельно, учителям информатики и математики, любителям программирования.

Валерий Рубанцев

Условные обозначения, принятые в книге:

Дополнение или замечание

Требование или указание

```
Исходный код
```

Задание для самостоятельного решения

Заголовок проекта:

Проект

Исходные коды всех проектов находятся в папке **_PasSFMLProjects**

Компилятор *PascalABC.NET* выдаёт ошибки при компиляции новых программ. Поэтому перед началом работы загрузите файл *SFML shader.pas* и запустите его. Затем загружайте свои программы с диска или пишите новые.

Оглавление

Первая программа

Любое игровое приложение начинается с создания **окна**. В графической библиотеке *SFML* окно – это экземпляр класса **RenderWindow**. Класс имеет несколько **конструкторов**, но чаще всего пользуются этим:

```
RenderWindow(mode: VideoMode, title: String);
```

Здесь:

title – *строка*, которая выводится в заголовке окна приложения
mode – *размеры* окна в пикселях

В начале всех графических программ нужно подключить библиотеки *SFML*:

```
{$reference 'sfmlnet-graphics-2.dll'}
{$reference 'sfmlnet-system-2.dll'}
{$reference 'sfmlnet-window-2.dll'}

uses System;
uses SFML.Graphics;
uses SFML.Window;
uses SFML.System;
```

Теперь можно создать **окно приложения**:

```
// создаём главное окно:
var wind := new RenderWindow(new VideoMode(800, 600), 'SFML Window');
```

Оно имеет **размеры** 800 на 600 пикселей и **надпись** "*SFML Window*".

Запустите приложение и убедитесь, что окно появилось на экране. Но оно тут же закрывается, поэтому нужно добавить ещё несколько обязательных строчек кода.

Включаем **вертикальную синхронизацию**:

```
// вертикальная синхронизация:
wind.SetVerticalSyncEnabled(true);
```

Если параметр функции SetVerticalSyncEnabled имеет значение *true*, то вертикальная синхронизация включается. Если *false* – отключается.

Рано или поздно окно нужно **закрыть**, нажав кнопку с крестиком в его правом верхнем углу. При этом вызывается метод, который нужно добавить к обработчику событий Closed:

```
wind.Closed += OnClosed;
```

Сама процедура OnClosed очень простая:

```
// ЗАКРЫВАЕМ ПРИЛОЖЕНИЕ
procedure OnClosed(sender: object; e: EventArgs);
begin
   var w := Window(sender);
   w.Close();
end;
```

Вот теперь можно начинать бесконечный **игровой цикл**:

```
// игровой цикл:
while (wind.IsOpen) do
begin

end;
```

Он начинается после создания окна приложения и заканчивается, когда вы его закроете.

Свойство IsOpen возвращает *true*, если окно создано.

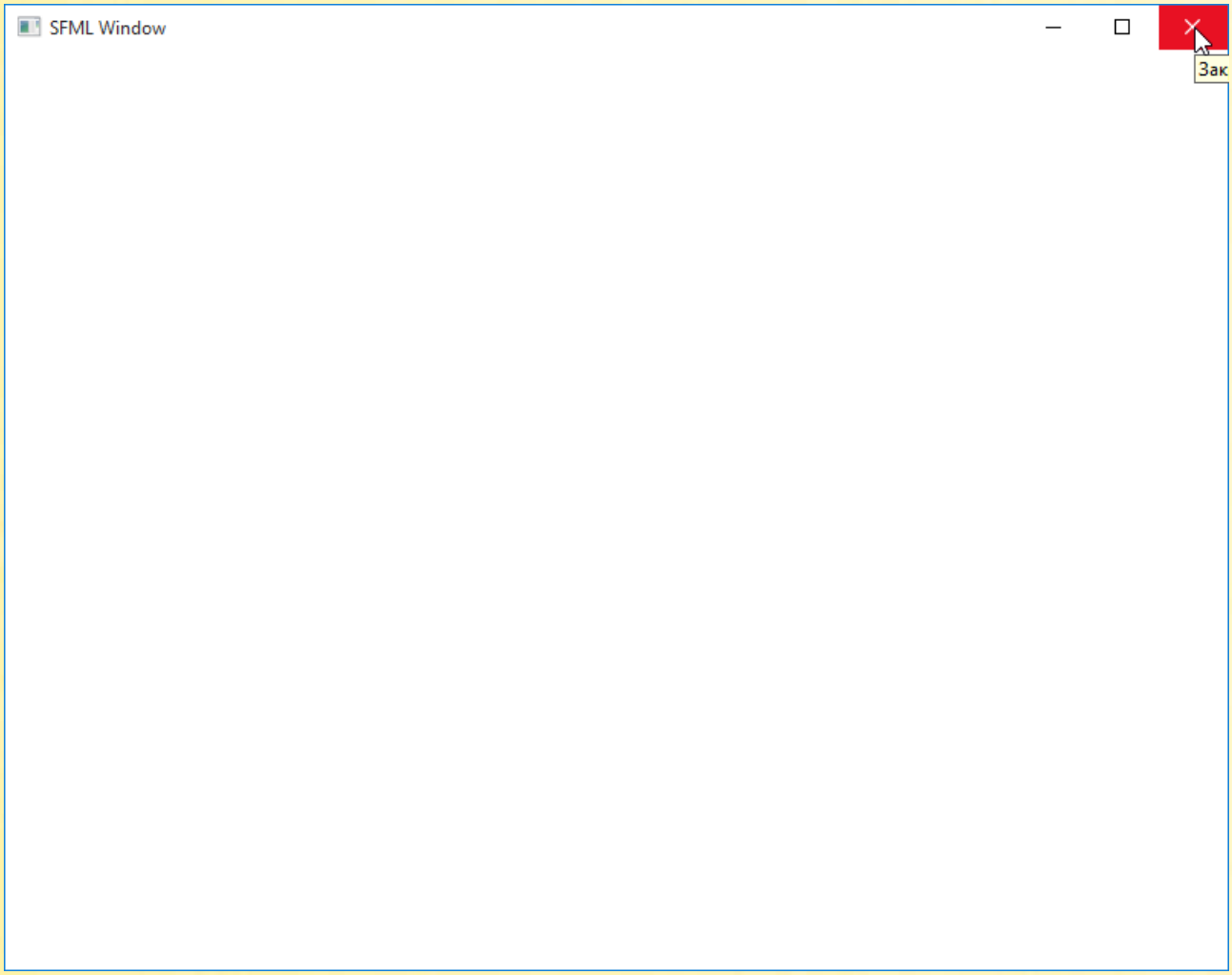
Чтобы окно **реагировало** на наши действия, вызываем **метод** DispatchEvents:

```
// вызываем все обработчики событий:
wind.DispatchEvents();
```

Он, в свою очередь, вызывает все методы (процедуры, функции), которые мы закрепили за обработчиками событий.

Пока у нас только одна процедура *OnClosed*, которая вызывается при нажатии на кнопку *Закрыть*.

Запустите программу. На экране появится белое окно с надписью *SFML Window:*



Окно можно перемещать по экрану, а потом закрыть, нажав кнопку с крестиком.

Минимальное приложение успешно создано:

```
{$reference 'sfmlnet-graphics-2.dll'}
{$reference 'sfmlnet-system-2.dll'}
{$reference 'sfmlnet-window-2.dll'}

uses System;
uses SFML.Graphics;
uses SFML.Window;
uses SFML.System;


// ЗАКРЫВАЕМ ПРИЛОЖЕНИЕ
procedure OnClosed(sender: object; e: EventArgs);
begin
   var w := Window(sender);
   w.Close();
end;



begin
   // создаём главное окно:
   var wind := new RenderWindow(new VideoMode(800, 600), 'SFML Win-
dow');
   // вертикальная синхронизация:
   wind.SetVerticalSyncEnabled(true);

   // обработчики событий:
   wind.Closed += OnClosed;

   // игровой цикл:
   while (wind.IsOpen) do
   begin
      // вызываем все обработчики событий:
      wind.DispatchEvents();
   end;

end.
```

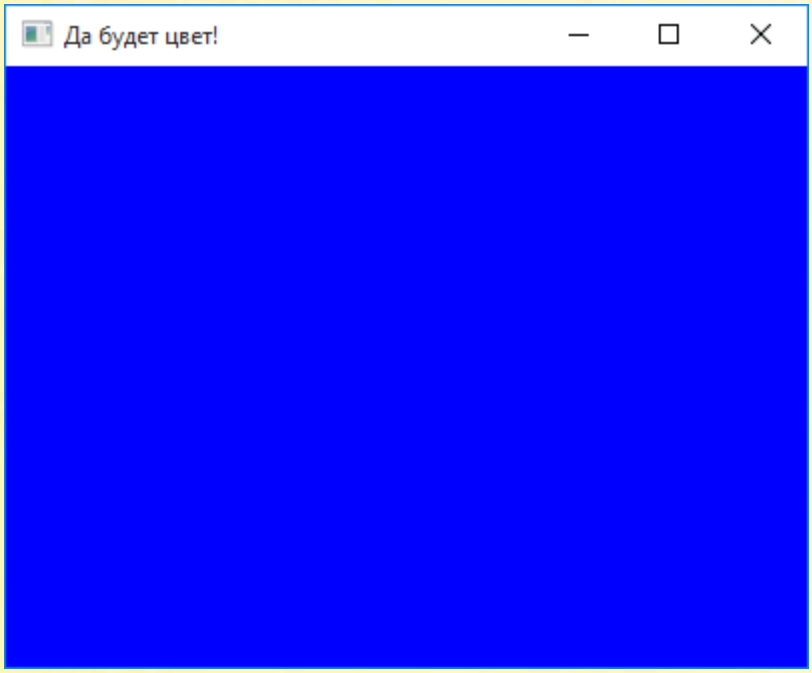
Проект *Да будет цвет!*

По умолчанию окно на экране белое. Но его можно окрасить в цикле *while*, в котором окно обновляется каждую итерацию. Для этого нужно вызвать метод Clear с заданным цветом:

```
// игровой цикл:
while (wind.IsOpen) do
begin
   // вызываем все обработчики событий:
   wind.DispatchEvents();

   // окрашиваем окно в синий цвет:
   wind.Clear(new Color(0, 0, 255));

   // показываем новый кадр на экране:
   wind.Display();
end;
```

Непосредственно окно обновляет метод Display, который должен вызываться последним в цикле *while*.

Окно остаётся синим, пока вы не закроете программу:



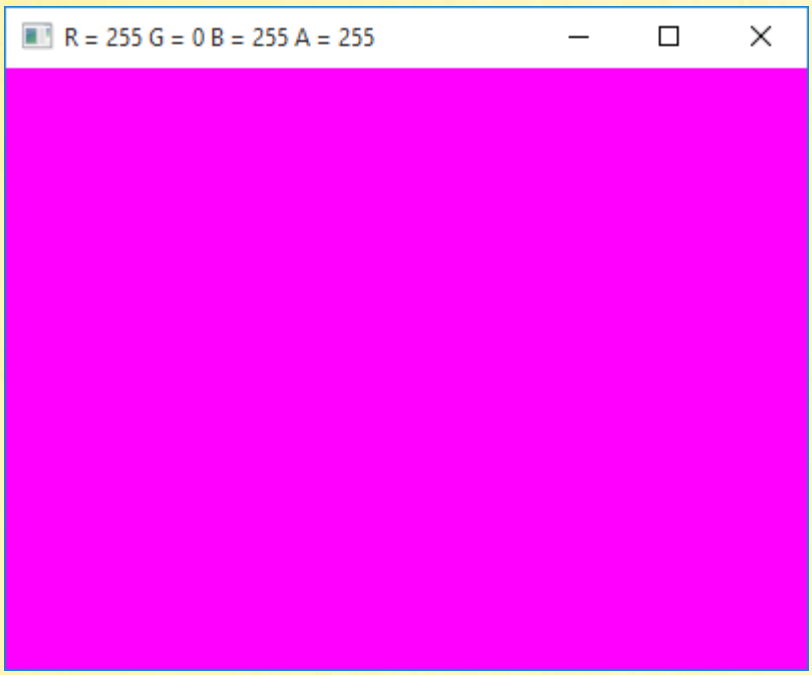
Однако нас в этом проекте больше интересуют **цвета**, в которые можно окрашивать фон окна.

Создаём новый цвет clr, окрашиваем окно и печатаем в заголовке числовые значения составляющих:

```
// окрашиваем окно:
var clr := new Color(1, 2, 255, 111);
wind.Clear(clr);
wind.SetTitle(String.Format('R = {0} G = {1} B = {2} A = {3}' ,
                            clr.R, clr.G, clr.B, clr.A));
```

Структура Color имеет 9 предопределённых цветов, пользоваться которыми удобнее, чем конструкторами, поскольку новый цвет из составляющих создавать не нужно:

```
// окрашиваем окно:
var clr := Color.Magenta;
wind.Clear(clr);
wind.SetTitle(String.Format('R = {0} G = {1} B = {2} A = {3}',
                            clr.R, clr.G, clr.B, clr.A));
```



Проект *Иконка*

По умолчанию в заголовке окна выводится системная иконка. Её можно заменить более подходящей.

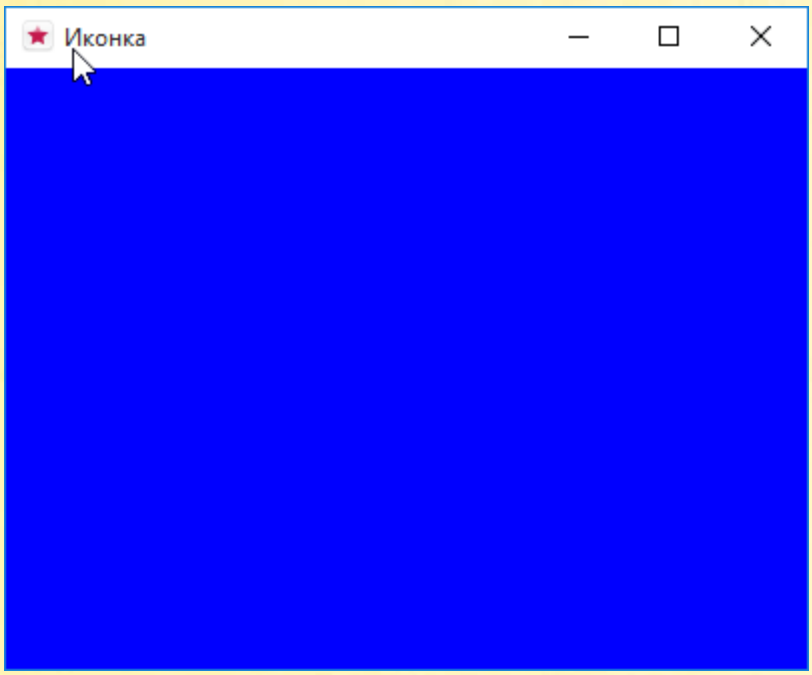
Обычно размер иконок составляет 16 x 16 или 32 x 32 пикселя. Загружаем картинку из файла:

```
// загружаем картинку:
var image := new Image('Media/ico_32x32.png');
```

Передаём методу окна SetIcon размеры картинки и массив её пикселей:

```
// устанавливаем иконку в заголовке окна:
wind.SetIcon(image.Size.X, image.Size.Y, image.Pixels);
```

Запускаем программу – новая иконка на месте:


Иконка

Класс RectangleShape (Прямоугольники)

Класс RectangleShape описывает прямоугольник. Рассмотрим свойства объектов этого класса на примерах.

Проект *Класс RectangleShape*

Закрасим окно серым цветом:

```
// цвет окна:
var wclr := new Color(222,222,222);
```

И нарисуем в центре окна прямоугольник. Используем конструктор по умолчанию:

```
// создаём прямоугольник:
var rect := new RectangleShape();
```

Его размеры можно задать сразу, в конструкторе, но мы сделаем это с помощью свойства Size:

```
// задаём размеры:
rect.Size := new Vector2f(120,80);
```

Поместим прямоугольник в центре окна:

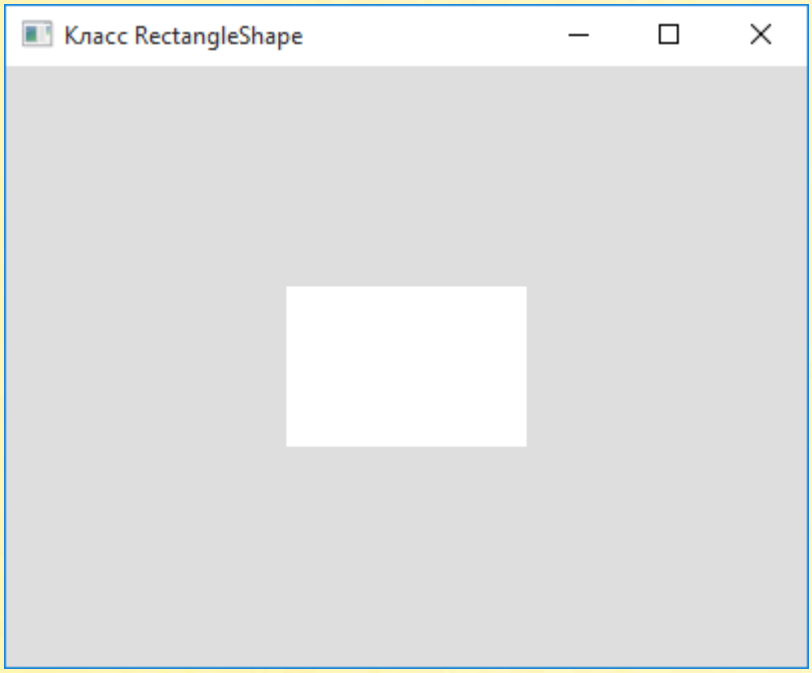
```
// задаём координаты:
rect.Position := new Vector2f(140,110);
```

В игровом цикле заливаем фон окна цветом *wclr*, а затем рисуем прямоугольник:

```
   // игровой цикл:
```

```
  while (wind.IsOpen) do
  begin
     // вызываем все обработчики событий:
     wind.DispatchEvents();

     // очищаем окно:
     wind.Clear(wclr);

     // рисуем прямоугольник:
     wind.Draw(rect);

     // показываем новый кадр на экране:
     wind.Display();
end;
```

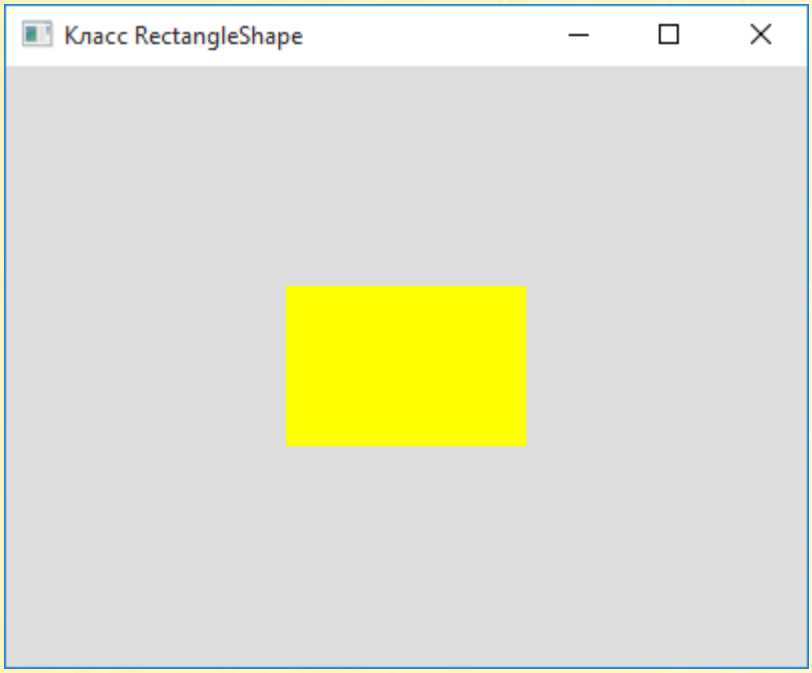
По умолчанию прямоугольник окрашен в белый цвет и не имеет контура:



Прямоугольник получился не очень красивый, поэтому закрашиваем его жёлтым цветом:

```
// окрашиваем в жёлтый цвет:
rect.FillColor := Color.Yellow;
```

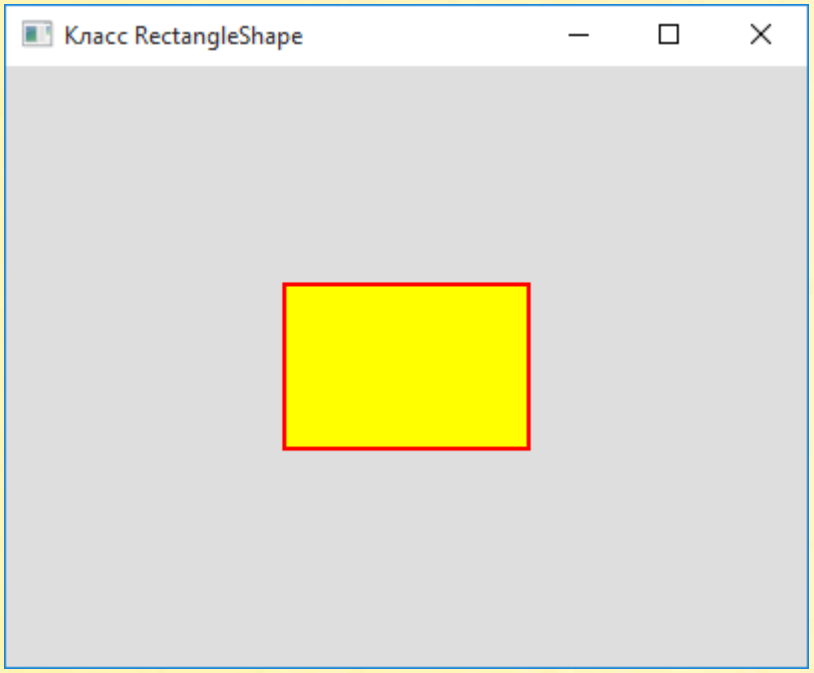
Уже красивее:



Осталось добавить **контур**:

```
// толщина контура:
rect.OutlineThickness
:= 2;
// цвет контура:
rect.OutlineColor :=
Color.Red;
```

Совсем другое дело:



Создаём точную **копию** первого прямоугольника:

```
// создаём копию:
var rect2 := new RectangleShape(rect);
```

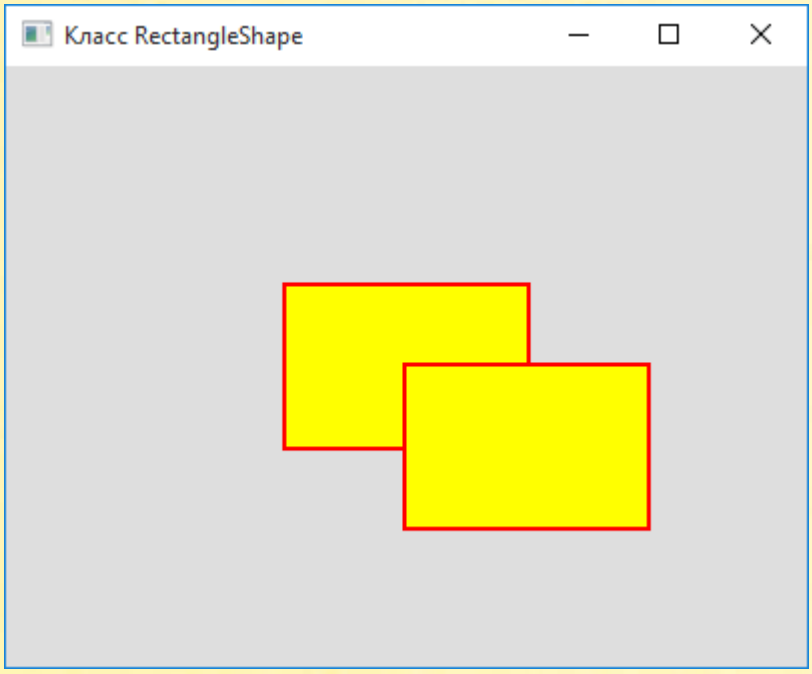
Чтобы увидеть новый прямоугольник на экране, немного сдвигаем его вниз-вправо:

```
// перемещаем:
rect2.Position := new Vector2f(200, 150);
```

В игровом цикле рисуем оба прямоугольника:

```
// рисуем прямоугольники:
wind.Draw(rect);
wind.Draw(rect2);
```

Теперь у нас 2 прямоугольника:



Проект *Трансформации прямоугольника*

Чтобы избежать коллизий с типом *Text паскаля*, объявляем новый **тип текста** *SFML*:

```
type
   Text = SFML.Graphics.Text;
```

Размеры окна и цвет фона зададим **константами**:

```
const
   // размеры окна:
   WIDTH = 600;
   HEIGHT = 300;
   // цвет фона:
   BACKCOLOR = Color.White;
```

Также нам понадобятся **переменные**:

```
var
   // прямоугольник:
   rect := new RectangleShape();
   // окно приложения:
   wind: RenderWindow := nil;
   // шрифт:
   fnt: Font := nil;
   // текст:
   txt: Text := nil;
   // координатры мышки:
   mousePos: Vector2i;
```

Процедура **OnClosed** вам уже известна, но нужна ещё пара процедур-обработчиков события **мышки**. Первая реагирует на **нажатие кнопки** мышки:

```
// НАЖИМАЕМ КНОПКУ МЫШКИ
```

```
procedure Window_MouseButtonPressed(sender: Object; e:
MouseButtonEventArgs);
begin
   // запоминаем координаты мышки:
   var mouseX := e.X;
   var mouseY := e.Y;
   mousePos := new Vector2i(mouseX, mouseY);
end;
```

А вторая - на её перемещение:

```
// ПЕРЕМЕЩАЕМ МЫШКУ
procedure Window_MouseMoved(sender: Object; e: MouseMoveEventArgs);
begin
   if (not Mouse.IsButtonPressed(Mouse.Button.Left)) then
      exit;

   // текущие координаты мышки:
   var x := e.X;
   var y := e.Y;
   // расстояние:
   var dx := x - mousePos.X;
   var dy := y - mousePos.Y;

   // новый центр трансформаций прямоугольника:
   var newX := rect.Origin.X - dx;
   var newY := rect.Origin.Y - dy;
   rect.Origin := new Vector2f(newX, newY);

   // запоминаем новые координаты мышки:
   mousePos := new Vector2i(e.X, y);
end;
```

Создавать окно и другие атрибуты приложения удобнее в отдельной процедуре. Ей нужно передать *размеры*, *заголовок* и *стиль* окна.

По умолчанию в заголовке выводится строка *SFML Window*, а стиль окна - *Close*, то есть размеры окна изменять запрещено:

```
// СОЗДАЁМ ОКНО
```

```
procedure CreateWindow(width, height: uint64;
                       title: string := 'SFML Window';
                       style: Styles := Styles.Close);
begin
   // создаём окно приложения:
   wind := new RenderWindow(new VideoMode(width, height), title,
style);
   wind.SetVerticalSyncEnabled(true);

   // добавляем метод для закрывания окна:
   wind.Closed += OnClosed;
   // обработчик перемещения мышки:
   wind.MouseMoved += Window_MouseMoved;
   // обработчик нажатия кнопки мышки:
   wind.MouseButtonPressed += Window_MouseButtonPressed;

   // загружаем шрифт с диска:
   fnt := new Font('Media/verdana.ttf');
   // создаём текст:
   txt := new Text('МЫШКА: ВВЕРХ-ВНИЗ | ВЛЕВО-ВПРАВО', fnt, 20);

   //цвет текста:
   txt.Color := Color.Red;
   // координаты текста:
   txt.Position := new Vector2f(20, 0);
   // стиль текста:
   txt.Style := Text.Styles.Bold;
end;
```

В главном блоке вызываем процедуру *CreateWindow*:

```
begin
   // создаём главное окно:
   CreateWindow(WIDTH, HEIGHT, 'Трансформации прямоугольника');
```

Затем задаём свойства прямоугольника:

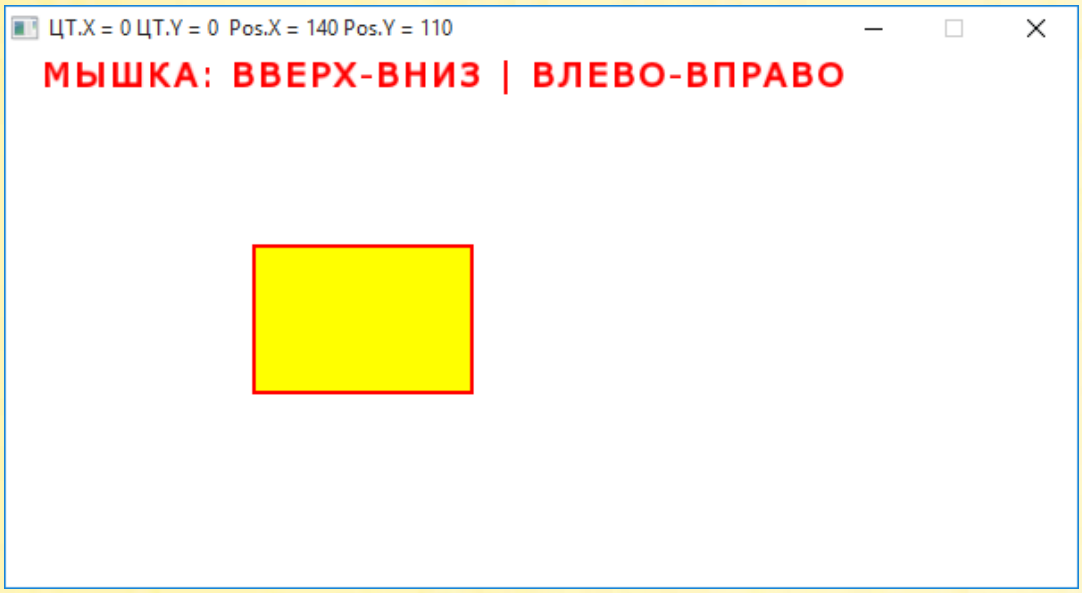
```
// задаём размеры прямоугольника:
rect.Size := new Vector2f(120, 80);
// координаты:
rect.Position := new Vector2f(140, 110);
```

```
// окрашиваем в жёлтый цвет:
rect.FillColor := Color.Yellow;
// толщина контура:
rect.OutlineThickness := 2;
// цвет контура:
rect.OutlineColor := Color.Red;
```

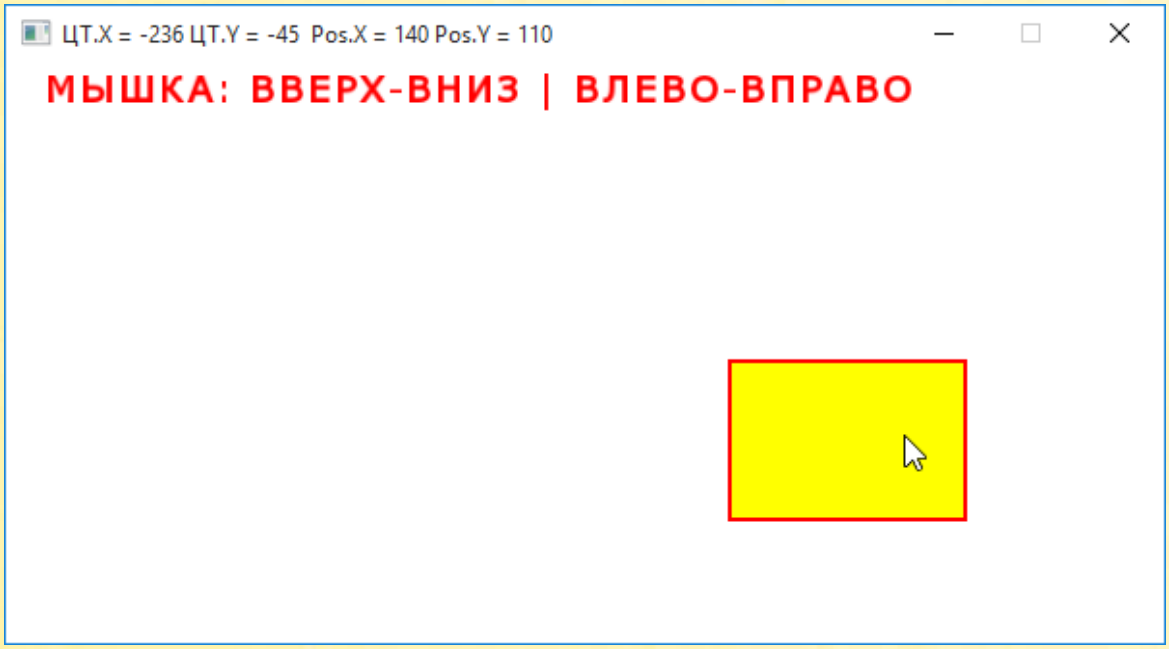
И наконец, в игровом цикле выводим сообщения и рисуем прямоугольник в исходном положении:

```
// игровой цикл:
while (wind.IsOpen) do
begin
   // вызываем все обработчики событий:
   wind.DispatchEvents();

   // очищаем окно:
   wind.Clear(BACKCOLOR);

   // рисуем прямоугольник:
   wind.Draw(rect);

   // печатаем в заголовке окна
   // координаты центра трансформации и
   // координаты прямоугольника:
   wind.SetTitle('ЦТ.X = ' + rect.Origin.X +
                 ' ЦТ.Y = ' + rect.Origin.Y +
                 '  Pos.X = ' + rect.Position.X +
                 ' Pos.Y = ' + rect.Position.Y);

   // цвет фона окна:
   wind.Clear(BACKCOLOR);

   // рисуем прямоугольник:
   wind.Draw(rect);
   // печатаем надпись:
   wind.Draw(txt);

   // показываем новый кадр на экране:
   wind.Display();
end;
```

Запускаем программу – **центр трансформаций прямоугольника** находится в левом верхнем углу. Его *локальные координаты* равны *(0,0).* А координаты самого прямоугольника – это координаты его точки трансформации (сейчас это верхний левый угол) относительно точки начала координат окна, которая находится в его левом верхнем углу:



Передвигаем мышку. Обратите внимание, что изменяются только координаты центра трансформаций окна, а координаты самого прямоугольника остаются без изменений:



Проект *Трансформации прямоугольника 2*

В этом проекте мы рассмотрим ещё 2 вида *трансформаций* – **поворот** и **масштабирование**. Начнём с поворота.

Свойство Rotation устанавливаеи или возвращает угол поворота фигуры в градусах:

```
property Rotation: float read write;
```

Поворот по часовой стрелке *положительный*, а обратном направлении – *отрицательный*.

Как вы помните, все виды трансформаций применяются относительно *центра трансформаций*, который по умолчанию находится в левом верхнем углу прямоугольника.

В процедуре **Window_MouseMoved** при вертикальном перемещении мышки изменяется *угол поворота* прямоугольника:

```
// ПЕРЕМЕЩАЕМ МЫШКУ
procedure Window_MouseMoved(sender: Object; e: MouseMoveEventArgs);
begin
   . . .

   // поворот:
   rect.Rotation += dy;

   . . .
end;
```

Удобнее вращать прямоугольник вокруг его **центра**, в который и нужно перенести центр трансформаций:

```
// переносим центр трансформаций
// в центр прямоугольника:
rect.Origin := new Vector2f(60, 40);
```

Свойство Scale устанавливает или возвращает масштаб (кратность) изменения размеров по горизонтали и по вертикали:

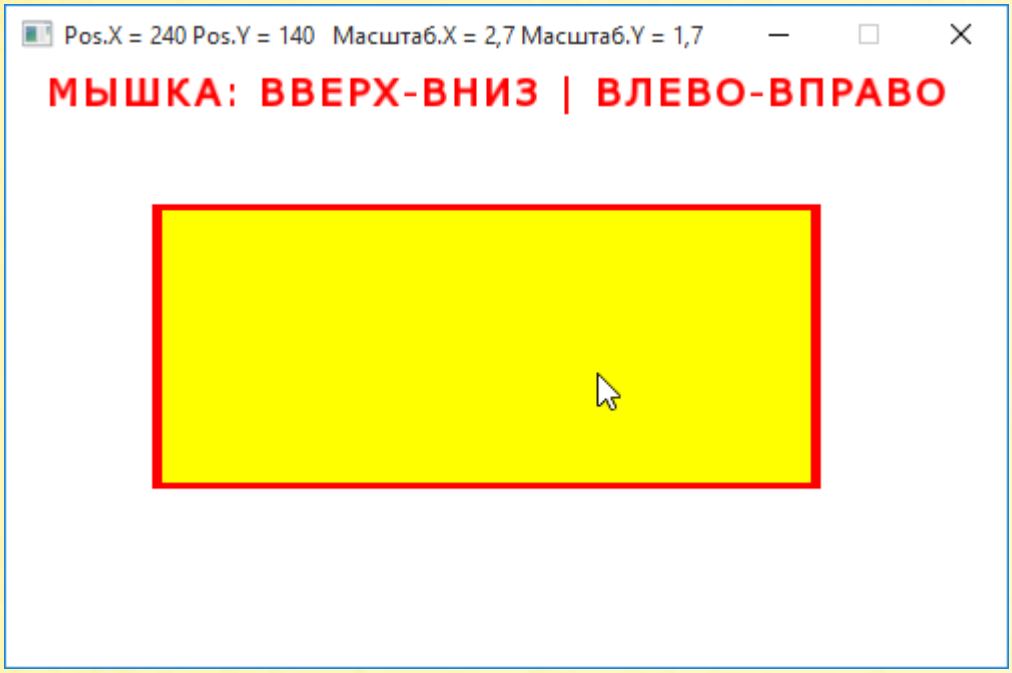
```
property Scale: Vector2f read write;
```

Теперь в методе Window_MouseMoved мы изменяем размеры прямоугольника в двух направлениях:

```
// ПЕРЕМЕЩАЕМ МЫШКУ
procedure Window_MouseMoved(sender: Object; e: MouseMoveEventArgs);
begin
. . .

   // переносим центр трансформаций
   // в центр прямоугольника:
   rect.Origin := new Vector2f(60, 40);

   // поворот:
   //rect.Rotation += dy;

   // масштабирование:
   rect.Scale := new Vector2f(rect.Scale.X + dx/10,
```

```
                                rect.Scale.Y + dy/10);
   . . .
end;
```

При масштабировании пропорционально изменяется и толщина контура:



При **отрицательных** значениях масштаба прямоугольник **переворачивается** на обратную сторону.

Проект *Случайные прямоугольники*

В **главном блоке** программы создаём *окно* с заданными размерами:

```
begin
   // создаём главное окно:
   CreateWindow(WIDTH, HEIGHT, 'Случайные прямоугольники');
```

Затем окрашиваем фон окна в светло-серый цвет:

```
// цвет фона окна:
var clr := new Color(222,222,222);
wind.Clear(clr);
```

Создаём и рисуем на экране 100 случайных прямоугольников:

```
// создаём прямоугольники:
for var i := 1 to 100 do
   wind.Draw(GetRandomRect3(160, 140));

// показываем на экране:
wind.Display();
```

А в игровом цикле только обрабатываем события:

```
// игровой цикл:
while (wind.IsOpen) do
begin
   // вызываем все обработчики событий:
   wind.DispatchEvents();
end;
```

Для окрашивания прямоугольников нам нужна функция GetRandomColor, которая возвращает *случайный цвет*:

```
rand: Random := new Random();


// ПОЛУЧАЕМ СЛУЧАЙНЫЙ ЦВЕТ
function GetRandomColor(alpha: boolean := false): Color;
begin
    // красная составляющая:
    var r := rand.Next(256);
    // зелёная составляющая:
    var g := rand.Next(256);
    // синяя составляющая:
    var b := rand.Next(256);
```

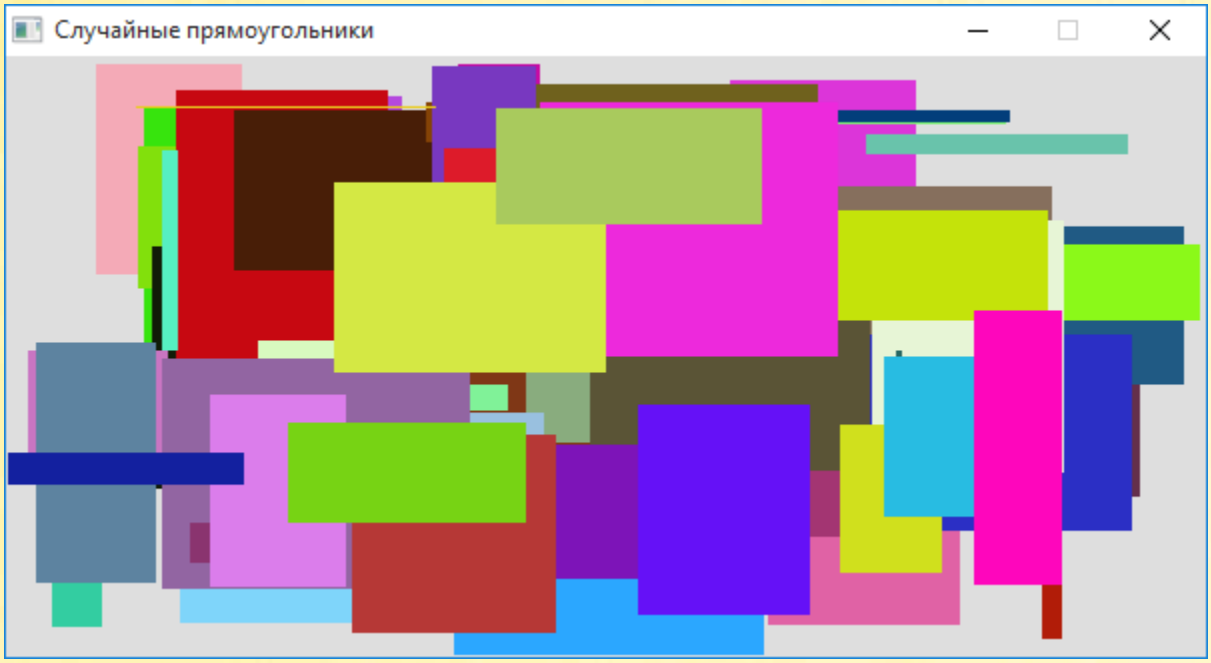
```
    // прозрачность:
    var a := rand.Next(256);

    if (not alpha) then
        Result := new Color(r,g,b)
    else
        Result := new Color(r, g, b, a);
end;
```

Новый случайный прямоугольник мы создаём в функции GetRandomRect, которая получает максимальные размеры прямоугольника:

```
// ВОЗВРАЩАЕМ СЛУЧАЙНЫЙ ПРЯМОУГОЛЬНИК
function GetRandomRect(maxW, maxH: integer): RectangleShape;
begin
   // размеры:
   var w := rand.Next(maxW);
   var h := rand.Next(maxH);
   var rect := new RectangleShape(new Vector2f(w,h));

   // координаты:
   var x := rand.Next(WIDTH - w);
   var y := rand.Next(HEIGHT - h);
   rect.Position := new Vector2f(x, y);

   // случайный цвет:
   rect.FillColor := GetRandomColor();

   Result := rect;
end;
```

Координаты прямоугольника мы выбираем так, чтобы он *целиком* поместился в окне приложения. А случайный цвет заливки получаем от функции *GetRandomColor*.

Запускаем программу – прямоугольники получились на славу:



Кто-то скажет, что им не хватает **контура**, и это правильно!

Пишем **вторую функцию** для создания случайного прямоугольника. На этот раз **с контуром**:

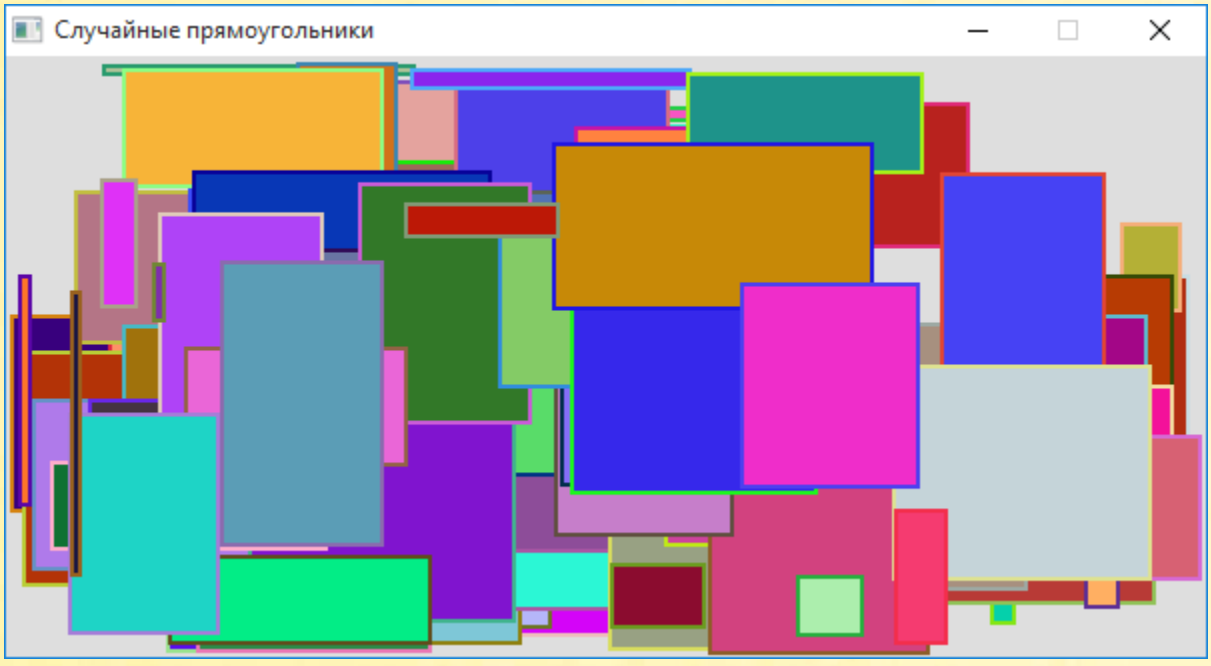
```
// ВОЗВРАЩАЕМ СЛУЧАЙНЫЙ ПРЯМОУГОЛЬНИК С КОНТУРОМ
function GetRandomRect2(maxW, maxH: integer): RectangleShape;
begin
    // размеры:
    var w := rand.Next(maxW);
    var h := rand.Next(maxH);
    var rect := new RectangleShape(new Vector2f(w,h));

    // координаты:
    var x := rand.Next(WIDTH - w);
    var y := rand.Next(HEIGHT - h);
    rect.Position := new Vector2f(x, y);

    // случайный цвет:
    rect.FillColor := GetRandomColor();
    rect.OutlineThickness := 2;
    rect.OutlineColor := GetRandomColor();

    Result := rect;
```

```
end;
```

Исправляем 1 строчку в главном блоке:

```
// создаём прямоугольники:
for var i := 1 to 100 do
   wind.Draw(GetRandomRect2(160, 140));
```

Оконтуренные прямоугольники ещё краше!

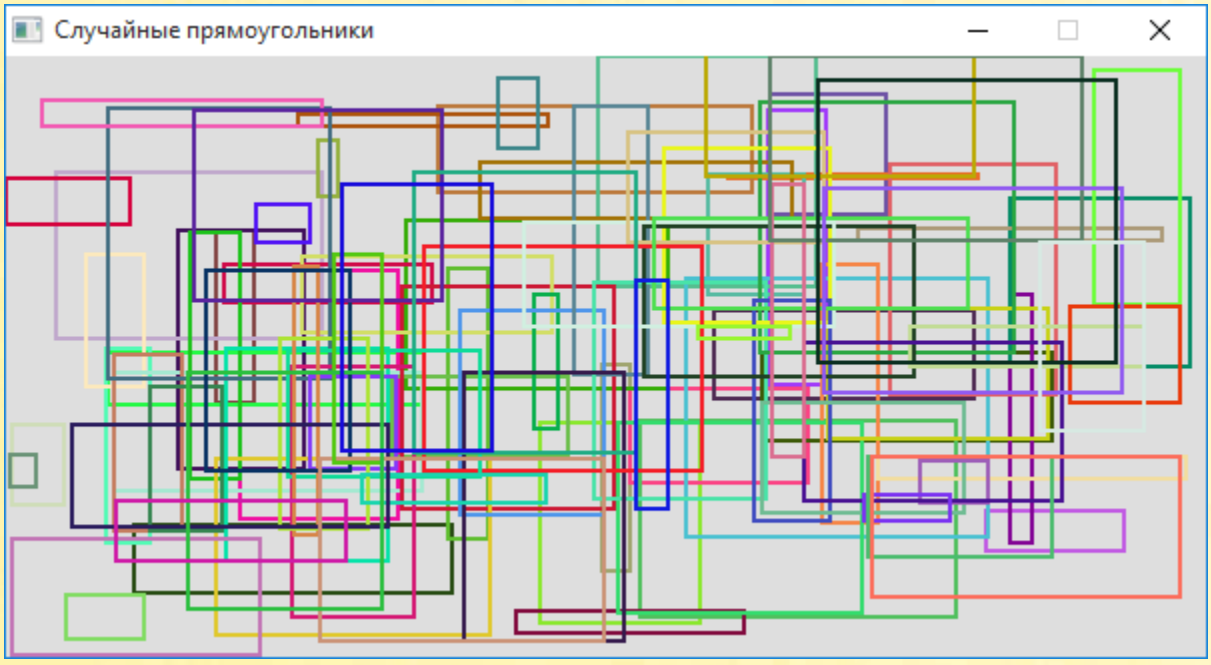


Однако отдадим должное минимализму и вегетарианству и нарисуем бесплотные прямоугольники:

```
// ВОЗВРАЩАЕМ СЛУЧАЙНЫЙ КОНТУРНЫЙ ПРЯМОУГОЛЬНИК
function GetRandomRect3(maxW, maxH: integer): RectangleShape;
begin
    // размеры:
    var w := rand.Next(maxW);
    var h := rand.Next(maxH);
    var rect := new RectangleShape(new Vector2f(w,h));
```

```
    // координаты:
    var x := rand.Next(WIDTH - w);
    var y := rand.Next(HEIGHT - h);
    rect.Position := new Vector2f(x, y);

    // прозрачная заливка:
    rect.FillColor := Color.Transparent;
    // случайный цвет контура:
    rect.OutlineThickness := 2;
    rect.OutlineColor := GetRandomColor();

    Result := rect;
end;


// создаём прямоугольники:
for var i := 1 to 100 do
   wind.Draw(GetRandomRect3(160, 140));
```

Пустые рамки оставляют место для самых смелых фантазий!



Проект *Случайные квадраты*

В предыдущем проекте мы без особой на то нужды создали 100 разных прямоугольных **объектов**, чтобы продемонстрировать некоторые методы класса *RectangleShape*.

И поскольку это так, то в новом проекте мы обойдёмся **единственным** прямоугольником для разузоривания всего окна приложения.

Всё окно (поле) разбито на квадратные клетки – 25 по ширине и столько же по высоте. Длина стороны квадрата – 25 пикселей (случайное совпадение!):

```
const
   // размер клеток в пикселях:
   QSIZE = 25;
   // число клеток по горизонтали:
   NCOL = 25;
   // число клеток по вертикали:
   NROW = 25;
```

Между клетками оставляем **зазор** шириной в PEN_WIDTH пикселей:

```
// толщина линий:
const PEN_WIDTH = 1;
```

Окрашиваем фон окна в чёрный цвет, который определяет цвет линий – границ (зазоров) между клетками:

```
// цвет фона:
BACKCOLOR = Color.Black;
```

Сами клетки окрашиваем в один из **двух** случайных цветов:

```
// цвета клеток:
COLORS: array of Color = (Color.Blue, Color.Yellow);
```

Размеры окна подгоняем под размеры поля:

```
// размеры окна:
WIDTH = (QSIZE + PEN_WIDTH) * NCOL + PEN_WIDTH;
HEIGHT = (QSIZE + PEN_WIDTH) * NROW + PEN_WIDTH;
```

На этом «глобальные» приготовления закончены, и мы переходим в главный блок. Здесь мы создаём окно приложения с соответсвующим заголовком, вызываем метод *Draw* для рисования цветных квадратиков и показываем художественные результаты на экране:

```
begin
   // создаём главное окно:
   CreateWindow(WIDTH, HEIGHT, 'Случайные квадраты');

    // цвет фона окна:
    wind.Clear(BACKCOLOR);

    // рисуем прямоугольники:
    Draw();

    // показываем на экране:
    wind.Display();

   // игровой цикл:
   while (wind.IsOpen) do
   begin
      // вызываем все обработчики событий:
      wind.DispatchEvents();
   end;

end.
```

Процедуру Draw мы начинаем с создания единственного квадрата *rect* по размерам клетки поля, или таблицы:

```
// ЧЕРТИМ ТАБЛИЦУ
procedure Draw();
begin
```

```
   // квадрат:
   var rect := new RectangleShape(new Vector2f(QSIZE, QSIZE));
```

Самый первый квадрат рисуем в левом верхнем углу окна с отступом на величину зазора:

```
//координаты верхнего левого угла первой клетки:
var x1 := PEN_WIDTH;
var y1 := PEN_WIDTH;
// чертим таблицу:
for var r := 0 to NROW-1 do
begin
   for var c := 0 to NCOL-1 do
   begin
      rect.Position := new Vector2f(x1, y1);
```

Цвет квадратика выбираем случайным образом из массива COLORS:

```
var clr := rand.Next(2);
var col := COLORS[clr];
// закрашиваем клетку:
rect.FillColor := col;
```

Рисуем квадратик в окне:

```
wind.Draw(rect);
```

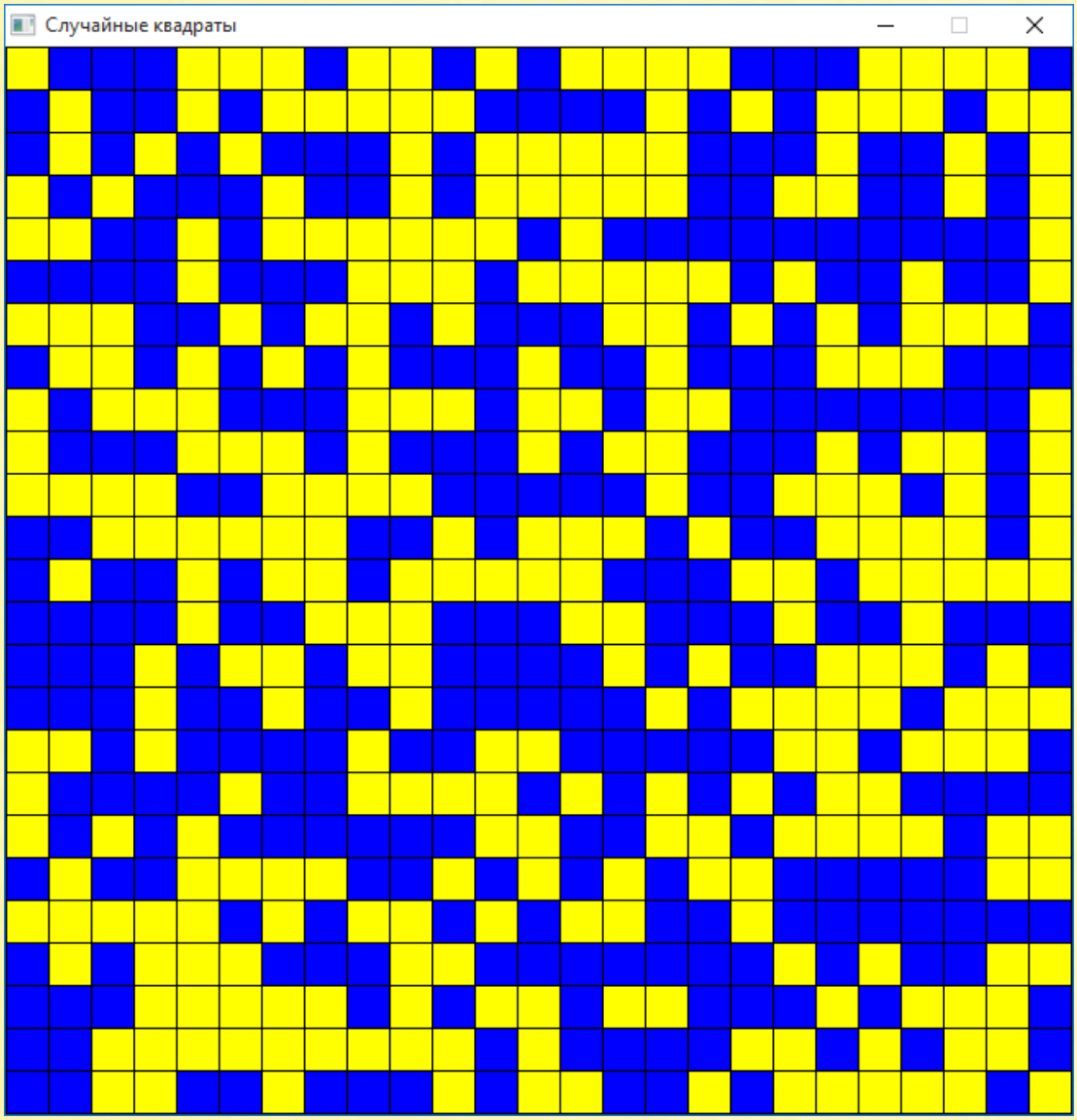
И переходим к следующей клетке по горизонтали:

```
   // координаты верхнего левого угла следующей клетки:
   x1 += QSIZE + PEN_WIDTH;
end;
```

Когда очередная строка квадратиков закончится, опускаемся на 1 клетку ниже:

```
        // переходим на следующую горизонталь:
        x1 := PEN_WIDTH;
        y1 += QSIZE + PEN_WIDTH;
    end;
end;
```

Все квадратики нарисованы, и у нас получилась вот такая геометрическая абстрактная картинка:



Поскольку цвет квадратиков выбирается случайно, то каждый раз будет получаться новая картинка, веселя и радуя вас!

Проект *Считаем квадратики*

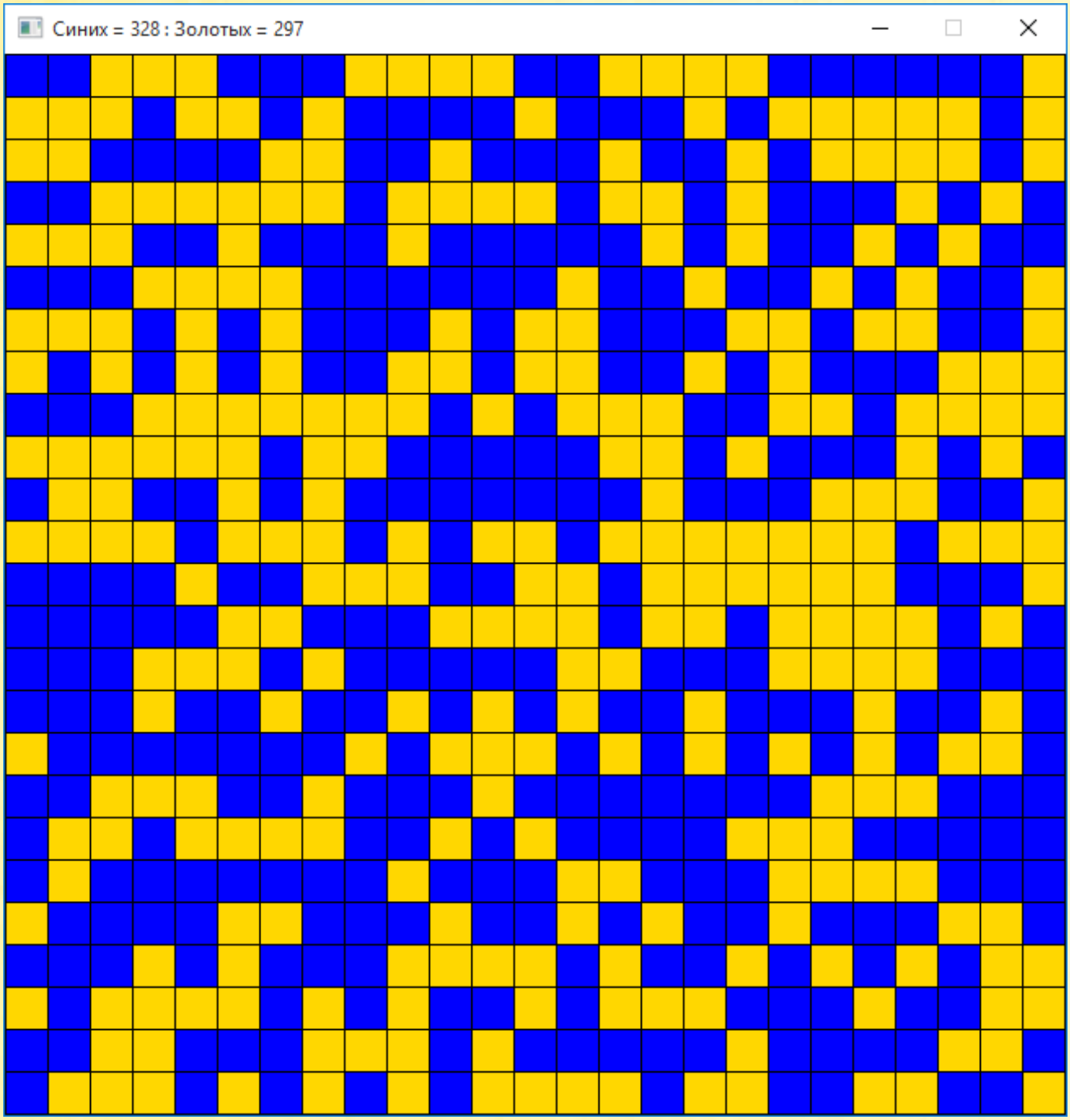
Чтобы рисование квадратиков не стало пустою забавою, их следует **пересчитать**! Конечно, считать квадратики, водя пальцем по экрану, это совсем неинтересно, поэтому мы будем оценивать число квадратиков **на глазок** и по результатам оценки делать вывод о том, квадратиков какого цвета на картинке **больше**. Вот так просто у нас получится психологический тест на объём восприятия.

За основу возьмём предыдущий проект и дополним **процедуру Draw** несколькими строчками:

```
// ЧЕРТИМ ТАБЛИЦУ
procedure Draw();
begin
   // число клеток с индексом 0:
   var n0 := 0;
   // квадрат:
   var rect := new RectangleShape(new Vector2f(QSIZE, QSIZE));
         . . .
         // цвет клетки:
         var clr := rand.Next(2);
         n0 += clr = 0 ? 1 : 0;
         . . .
   var vsego := NROW * NCOL;
   // число клеток с индексом 1:
   var n1 := vsego - n0;
   wind.SetTitle('Синих = ' + n0 + ' : Золотых = ' + n1);
end;
```

Запускаем программу – тест готов.

В заголовке мы предусмотрительно напечатали число квадратиков каждого цвета. В готовом задании такой подсказки быть не должно.

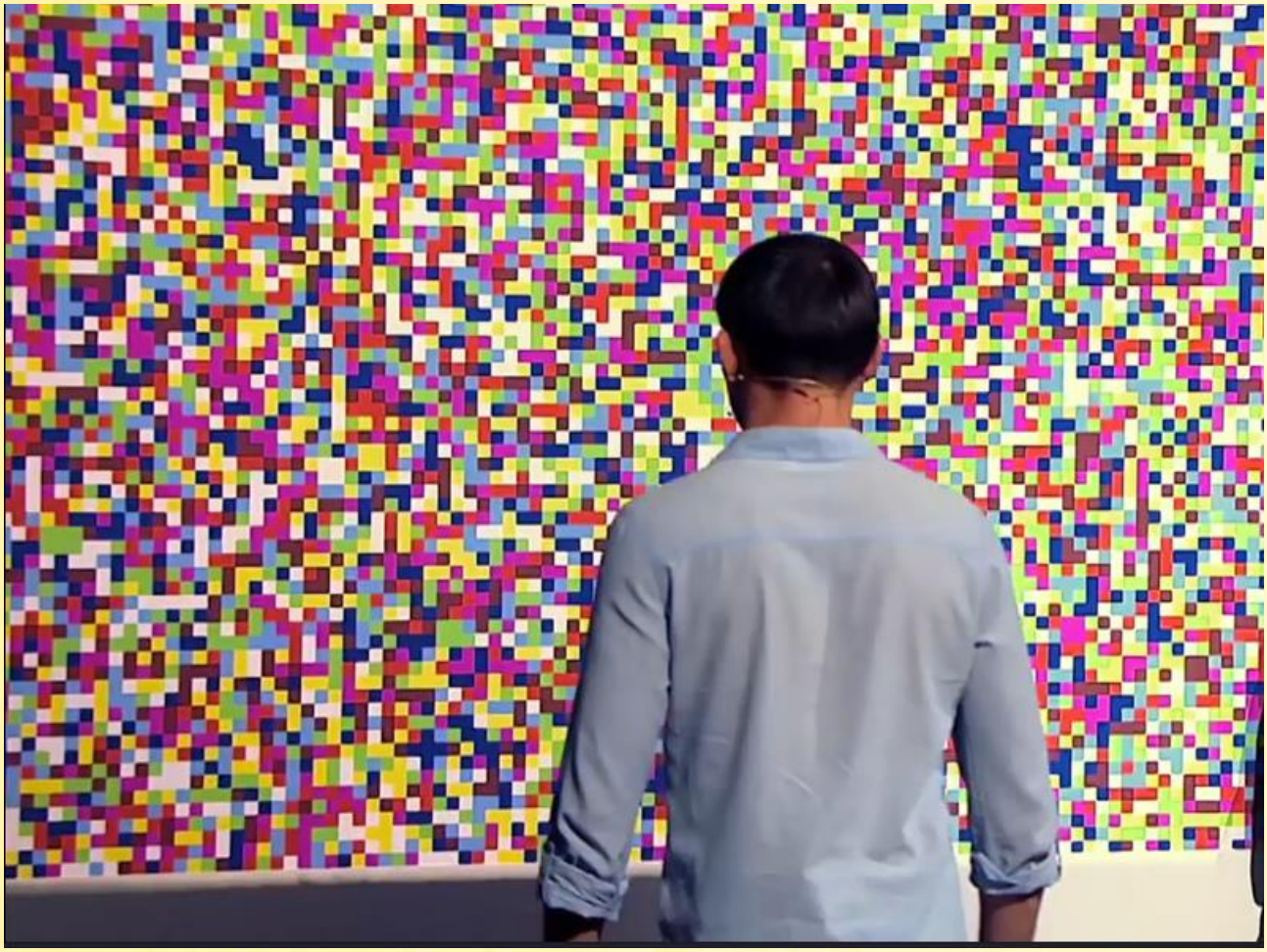

Синих = 328 : Золотых = 297

Проект *Пиксели*

В одном из выпусков программы *Удивительные люди* Евгений Дубин из Омска демонстрировал свои способности в номере *Стереоскопическое зрение*:

Он должен был найти на двух картинках 3 квадратика-пикселя, имеющие разный цвет.

Мы облегчим себе задачу: у нас пикселей гораздо меньше, и окрашены они только в 2 цвета. Здесь мы опять воспользуемся предыдущим проектом, и внесём в него необходимые изменения.



Так как теперь у нас 2 картинки, то ширину окна нужно **удвоить** и добавить ещё зазор между картинками

```
begin
   // создаём главное окно:
   CreateWindow(WIDTH + 10 + WIDTH, HEIGHT, 'Пиксели');
```

Мы заранее создали 2 **массива** для хранения цветов клеток:

```
var
   // окно приложения:
   wind: RenderWindow := nil;
   rand: Random := new Random();

   // таблицы:
```

```
   tableLeft: array [,] of integer = new integer[NCOL,NROW];
   tableRight: array [,] of integer = new integer[NCOL,NROW];
```

Сначала мы **заполняем** обе таблицы одинаковыми квадратиками:

```
// СОЗДАЁМ ТАБЛИЦЫ
procedure CreateTable();
begin
   for var r := 0 to NROW-1 do
      for var c := 0 to NCOL-1 do
      begin
         // цвет клетки:
         var clr := rand.Next(2);
         tableLeft[c, r] := clr;
         tableRight[c, r] := clr;
     end;
```

Но несколько квадратиков должны отличаться по цвету:

```
// число отличных пикселей:
OTL = 3;
```

Мы можем случайно выбирать квадратики по номеру, но их нужно **запоминать**, чтобы не перекрасить один и тот же квадратик несколько раз. Для хранения номеров перекрашенных квадратиков заведём **список**. Сначала он пуст:

```
// изменяем цвет OTL квадратиков:
var lstRCell := new List<integer>();
```

Также нам понадобятся **переменные** - для подсчёта перекрашенных квадратиков и для номера случайной клетки:

```
// перекрасили квадратиков:
var n := 0;
// номер случайной клетки:
var rcell := 0;
```

В цикле *while* мы выбираем случайную клетку, которой нет в списке, и записываем её туда:

```
while (n < OTL) do
begin
   repeat
      // выбираем случайную клетку:
      rcell := rand.Next(0, NCOL * NROW);
   until (not lstRCell.Contains(rcell));
   lstRCell.Add(rcell);
   n += 1;
end;
```

Все клетки выбраны. Для «красоты» ответа сортируем список:

```
lstRCell.Sort();
otvet := '(Строка, Колонка) --> ';
```

Ответ записываем в строковую переменную otvet:

```
// ответ:
otvet: string;
```

По номерам клеток в списке lstRCell определяем их индексы и перекрашиваем в другой цвет. Так как у нас всего два цвета, то сделать это нетрудно:

```
foreach var cell in lstRCell do
begin
   // по номерам клеток определяем их координаты:
   var r := cell div NCOL;
   var c := cell - r*NCOL;

   // цвет клетки:
   var clr := tableRight[c,r];
   // новый цвет:
   clr := 1 - clr;
   tableRight[c, r] := clr;
```

```
    otvet += '(' + (r + 1) + ',' +  (c + 1) + ') ';
end;
```

Процедуру Draw начинаем с создания таблиц:

```
// ЧЕРТИМ ТАБЛИЦУ
procedure Draw();
begin
   CreateTable();
```

Одновременно рисуем квадратики в обеих таблицах на экране. Цвета квадратиков для левой таблицы берём из массива tableLeft, а для правой - из массива tableRight:

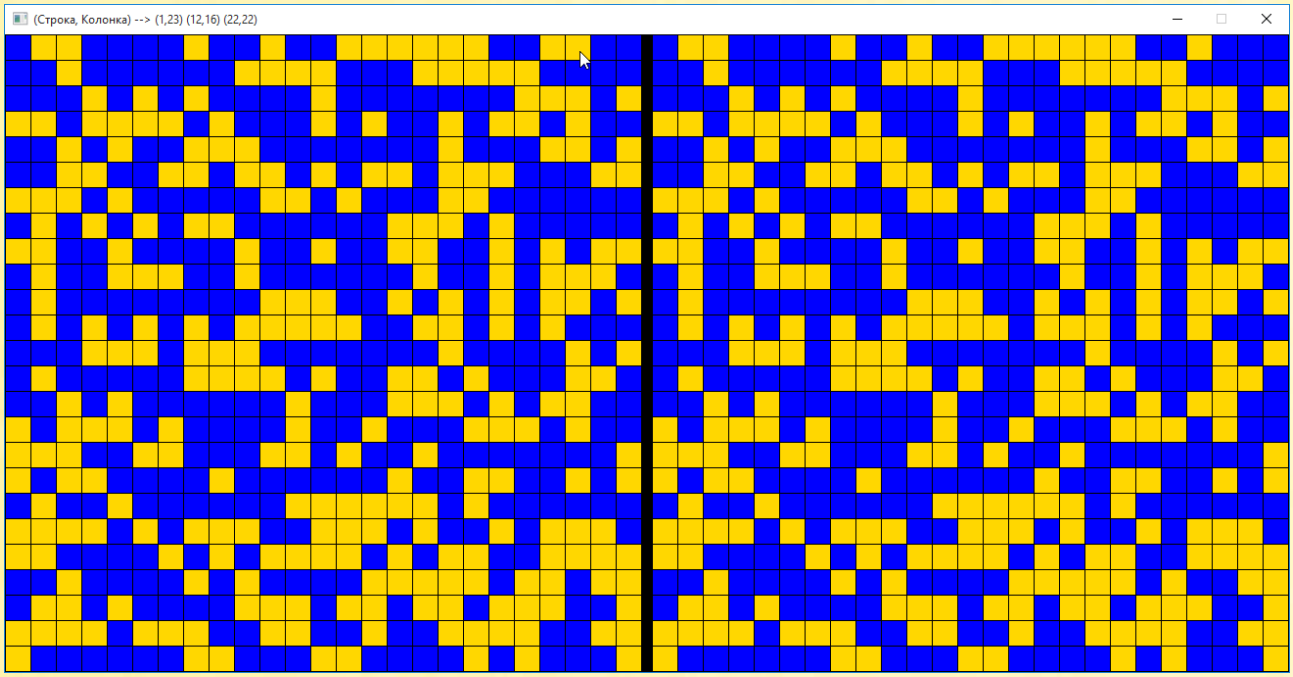
```
// квадрат:
var rect := new RectangleShape(new Vector2f(QSIZE, QSIZE));

// координаты верхнего левого угла первой клетки:
var x1 := PEN_WIDTH;
var y1 := PEN_WIDTH;
// чертим таблицу:
for var r := 0 to NROW-1 do
begin
   for var c := 0 to NCOL-1 do
   begin
      rect.Position := new Vector2f(x1, y1);
      // цвет клетки:
      var clr := tableLeft[c,r];
      var col := COLORS[clr];

      // закрашиваем клетку:
      rect.FillColor := col;
      wind.Draw(rect);

       // квадратик на второй таблице:
       clr := tableRight[c, r];
       col := COLORS[clr];
       rect.FillColor := col;
       rect.Position := new Vector2f(x1 + 10 + WIDTH, y1);
       wind.Draw(rect);
```

```
      // координаты верхнего левого угла следующей клетки:
      x1 += QSIZE + PEN_WIDTH;
   end;
   // переходим на следующую горизонталь:
   x1 := PEN_WIDTH;
   y1 += QSIZE + PEN_WIDTH;
end;
```

Закончив рисование квадратиков, печатаем **ответ** в заголовке окна:

```
  wind.SetTitle(otvet);
end;
```

Теперь вы можете развивать стереоскопическое зрение на простых примерах:

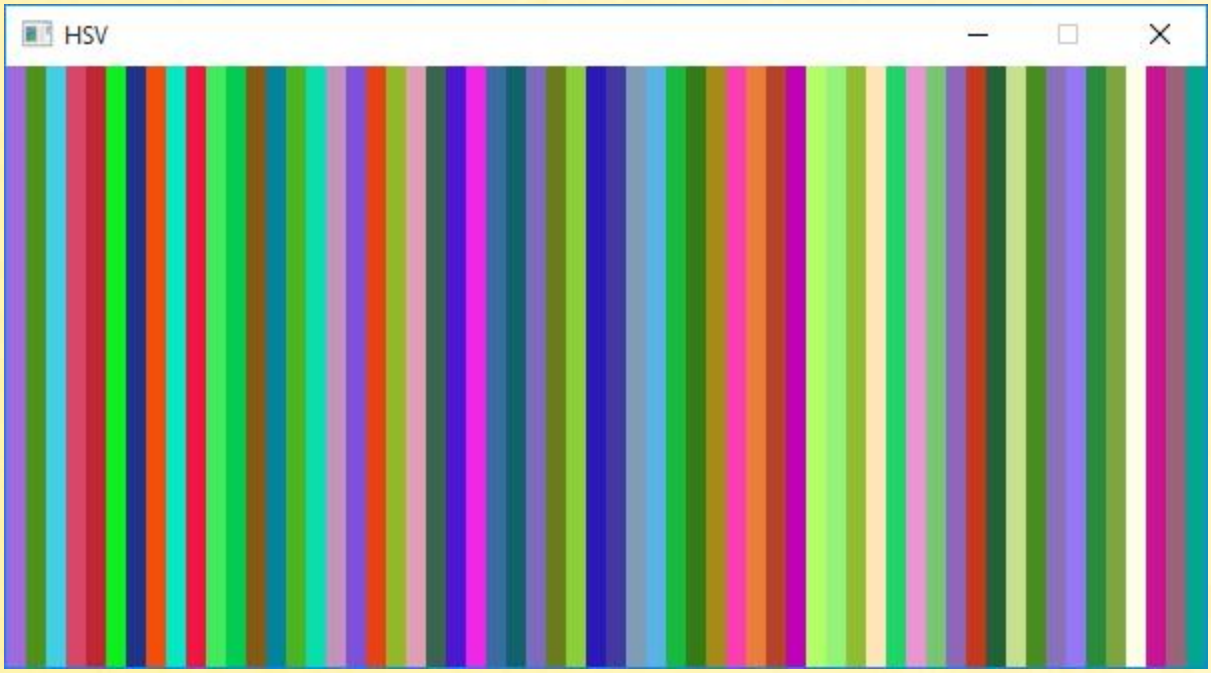


А когда вам это занятие покажется лёгким, то добавьте пикселей и цветов к нашим таблицам. Конечно, такими стереоскопическими способностями вы уже никого не удивите, поэтому постарайтесь придумать что-нибудь своё!

Проект *HSV*

В компьютерной графике чаще всего используется **цветовая модель RGB**, в которой все цвета получаются смешиванием трёх **цветовых составляющих** – **красной**, **зелёной** и **синей**. Из них легко получить любой случайный цвет. Мы написали для этого специальную функцию GetRandomColor:

```
procedure DrawRandRect();
begin
   var rect := new RectangleShape(new Vector2f(10, 300));
   for var i := 0 to 60-1 do
   begin
      rect.Position := new Vector2f(i * 10, 0);
      rect.FillColor := GetRandomColor();
      wind.Draw(rect);
   end;
end;
```



Но иногда нужно выполнить **плавный переход** от одного цвета у другому, как в радуге. В цветовой модели *RGB* это сделать сложно, поэтому для таких случаев разработали другую цветовую модель. В ней цвета также составляются из трёх

компонентов, но это не отдельные цвета, а **тон, насыщенность, значение** (или **тон, насыщенность, яркость**). Эта цветовая модель называется **HSV** (или **HSB**).

Но для вывода изображения на экран используется цветовая модель **RGB**, поэтому мы воспользуемся новой функцией **HSV2RGB** из библиотеки **RBLib**.

> Об этой библиотеке читайте в конце книги.

Она получает составляющие цвета в модели *HSV*, а возвращает тот же цвет в модели *RGB*.

Чтобы прямоугольники смотрелись убедительнее, **окно** сделаем шире:

```
const
   // размеры окна:
   WIDTH = 600;
   HEIGHT = 300;
```

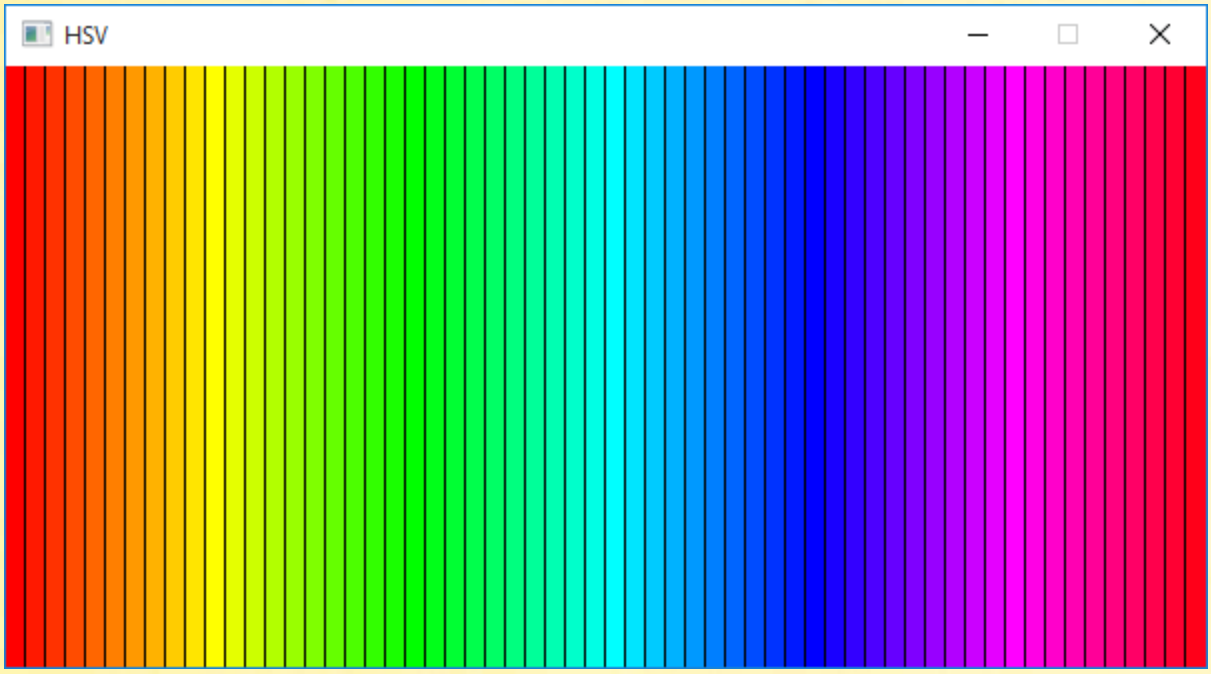
В процедуре **DrawRect** рисуем полоску из цветных прямоугольников:

```
// РИСУЕМ ПРЯМОУГОЛЬНИКИ
procedure DrawRect();
begin
   // прямоугольник -->
   // ширина:
   var w := WIDTH div N_RECT;
   var rect := new RectangleShape(new Vector2f(w, HEIGHT));
   // контур:
   rect.OutlineThickness := 1;
   rect.OutlineColor := Color.Black;
   for var i := 0 to N_RECT-1 do
      begin
         rect.Position := new Vector2f(i * w, 0);
```

Цвета заливки прямоугольников выбираем так, чтобы они полностью покрыли диапазон **0..360**:

```
            rect.FillColor :=  HSV2RGB(i*360.0/N_RECT);
            wind.Draw(rect);
        end;
end;
```

На рисунке хорошо видно, что цвета плавно переходят от красного к оранжевому, потом жёлтому – и так пока снова не вернутся к красному:

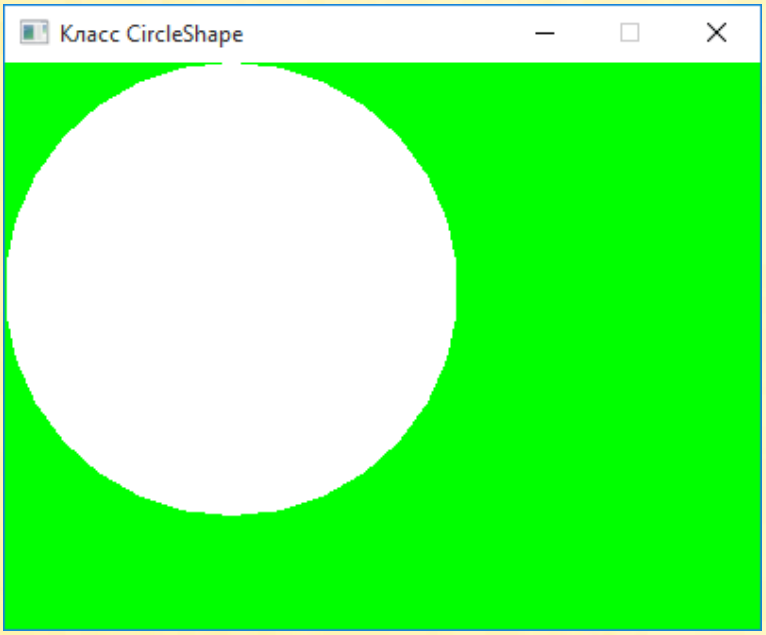


Класс CircleShape (Круги)

Класс CirleShape описывает круг.

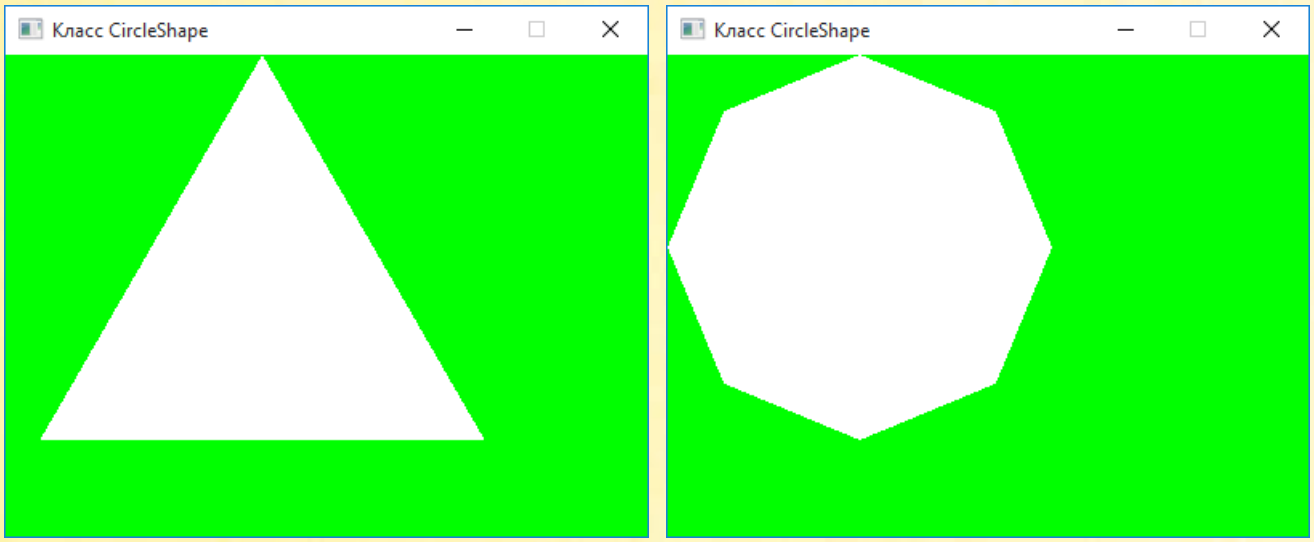
По умолчанию круг окрашен в белый цвет и не имеет контура:

```
begin
   // создаём главное окно:
   CreateWindow(WIDTH, HEIGHT, 'Класс CircleShape');

   wind.Clear(BACKCOLOR);

   // чертим круг:
   var circle := new CircleShape(120);
   wind.Draw(circle);
   wind.Display();

   // игровой цикл:
   while (wind.IsOpen) do
   begin
      // вызываем все обработчики событий:
      wind.DispatchEvents();
   end;
end.
```



Можно создавать не только круги, но и **правильные многоугольники**. Число точек должно быть не меньше трёх:

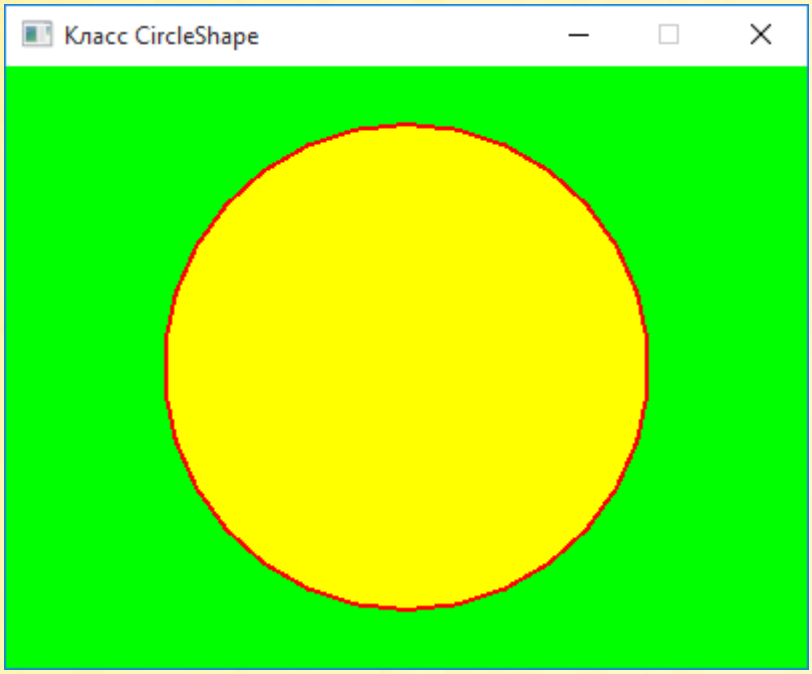
```
var circle := new CircleShape(120,3);

var circle := new CircleShape(120,8);
wind.Draw(circle);
```



По умолчанию все фигуры имеют координаты **(0,0)**, то есть левый верхний угол описывающего прямоугольника находится точно в начале координат.

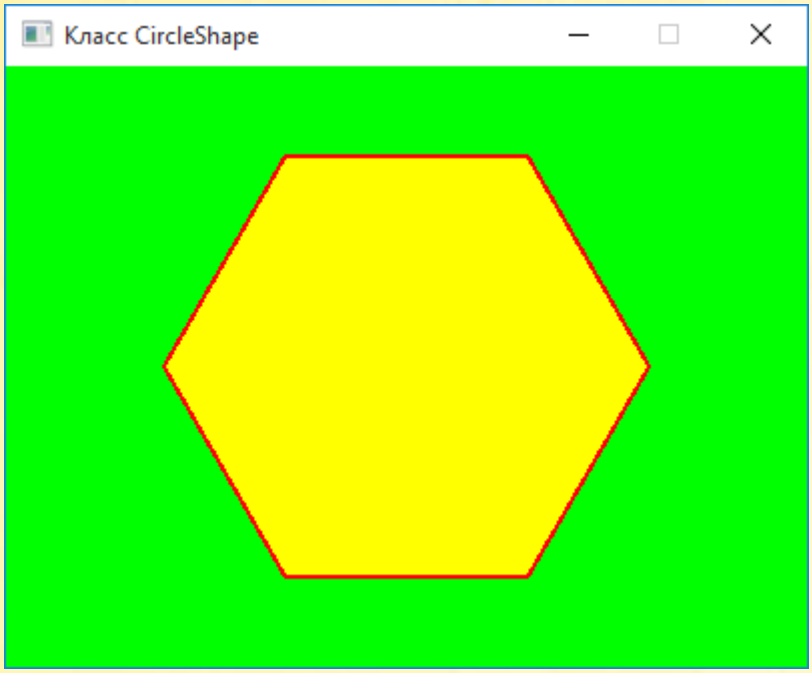
Экземпляры класса **CircleShape** имеют те же самые свойства, что и прямоугольники:

```
// ЧЕРТИМ КРУГ
procedure DrawCircle();
begin
   var circle := new CircleShape(120);
   circle.FillColor := Color.Yellow;
   circle.OutlineColor := Color.Red;
   circle.OutlineThickness := 2;
   circle.Origin := new Vector2f(120, 120);
   circle.Position := new Vector2f(WIDTH / 2, HEIGHT / 2);
   wind.Draw(circle);
end;
```



Так как мы можем создать не только круг, но и правильный многоугольник, то класс *CircleShape* имеет метод SetPointCount, который позволяет задавать число вершин многоугольника:

```
// ЧЕРТИМ МНОГОУГОЛЬНИК
procedure DrawPoly();
begin
   var circle := new CircleShape(120);
   circle.SetPointCount(6);
   circle.Rotation := 30;

   circle.FillColor := Color.Yellow;
   circle.OutlineColor := Color.Red;
   circle.OutlineThickness := 2;
   circle.Origin := new Vector2f(120, 120);
   circle.Position := new Vector2f(WIDTH / 2, HEIGHT / 2);
   wind.Draw(circle);
end;
```



По умолчанию круг – это многоугольник с 30-ю вершинами, поэтому он не очень ровный. Но вы можете задать и большее число вершин. Например, 300.

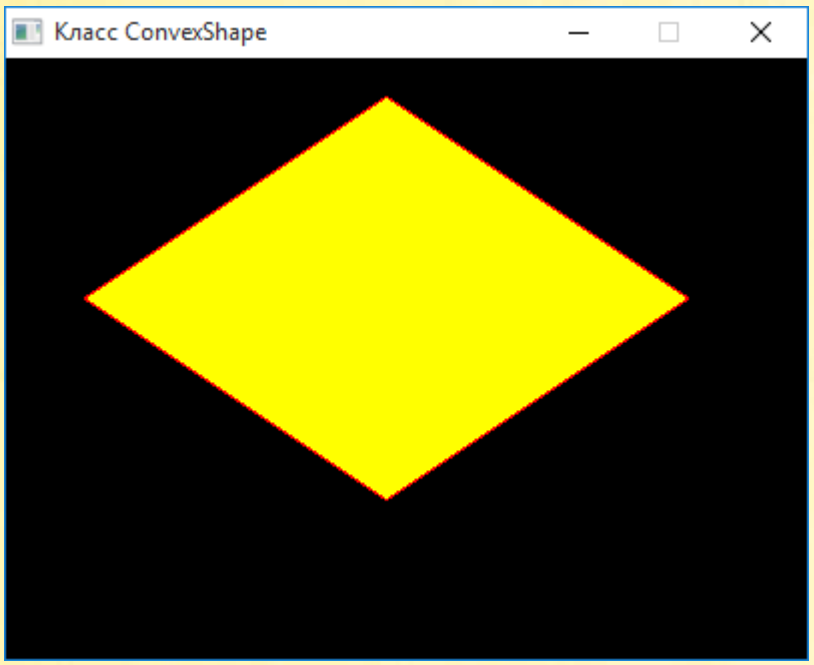
Класс ConvexShape (Многоугольники)

Класс ConvexShape описывает произвольный многоугольник.

Все свойства у экземпляров класса *ConvexShape* такие же, как у прямоугольников.

Создаём прямоугольник с 4 вершинами:

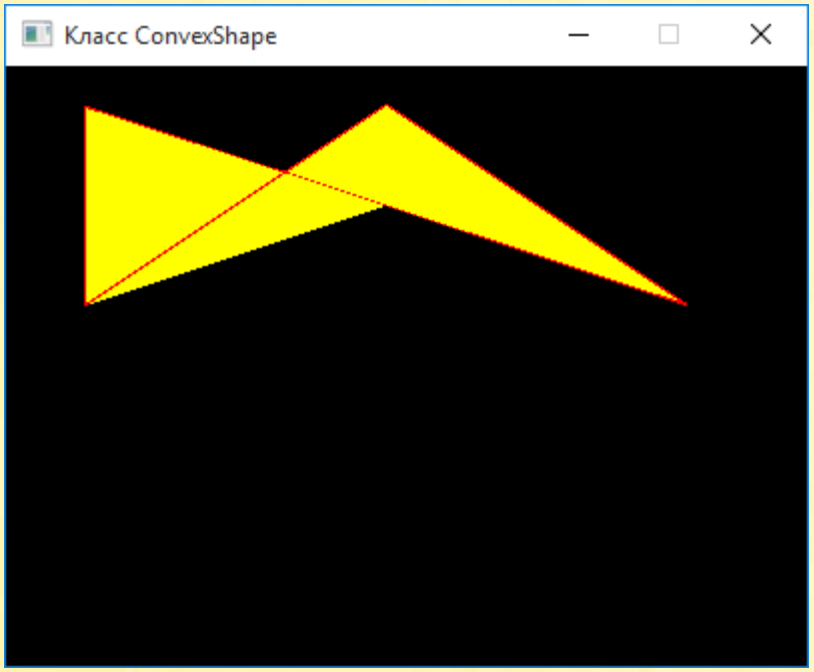
```
// СОЗДАЁМ МНОГОУГОЛЬНИК
procedure CreateConvex();
begin
   var h := 200;
   var w := 300;
   var convex := new ConvexShape(4);
   convex.Position := new Vector2f(40, 20);
   convex.FillColor := Color.Yellow;
   convex.OutlineColor := Color.Red;
   convex.OutlineThickness := 1;
   convex.SetPoint(0, new Vector2f(0, h/2));
   convex.SetPoint(1, new Vector2f(w/2, 0));
   convex.SetPoint(2, new Vector2f(w, h / 2));
   convex.SetPoint(3, new Vector2f(w / 2, h));
   wind.Draw(convex);
end;

begin
   // создаём главное окно:
   CreateWindow(WIDTH, HEIGHT, 'Класс ConvexShape');

   // создаём многоугольник:
   CreateConvex();
   wind.Display();

   // игровой цикл:
   while (wind.IsOpen) do
   begin
      // вызываем все обработчики событий:
      wind.DispatchEvents();
   end;

end.
```



Закомментируйте одну строчку:

```
//convex.SetPoint(3, new Vector2f(w / 2, h));
```

Так как координаты одной вершины мы не задали, то она получила нулевые значения, и наш ромб превратился в причудливую загогулину:

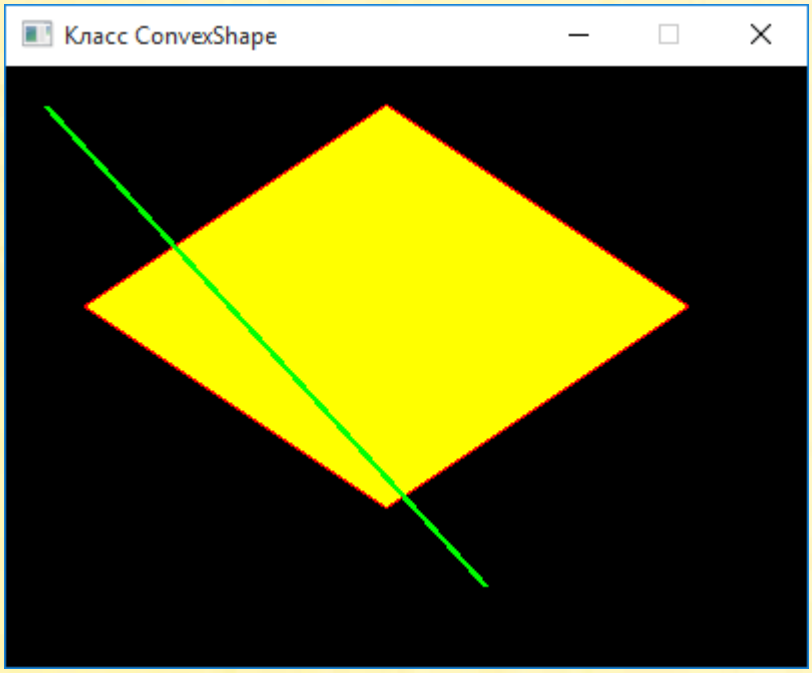


Если вы закомментируете 3 строчки, то от ромба останется отрезок прямой. И это хорошо, ведь в графическом арсенале *SFML* прямые линии отстутствуют.

Пишем процедуру DrawLine, которая по заданным координатам двух точек чертит отрезок прямой линии:

```
// ЧЕРТИМ ОТРЕЗОК ПРЯМОЙ
procedure DrawLine(x1, y1, x2, y2: single);
begin
   var line := new ConvexShape(4);
   line.OutlineColor := Color.Green;
   line.OutlineThickness := 1.2;
   line.SetPoint(0, new Vector2f(x1, y1));
   line.SetPoint(1, new Vector2f(x1, y1));
   line.SetPoint(2, new Vector2f(x2, y2));
   line.SetPoint(3, new Vector2f(x2, y2));
   wind.Draw(line);
end;

   . . .
   // создаём многоугольник:
   CreateConvex();
   // чертим линию:
   DrawLine(20, 20, 240, 260);
   wind.Display();
   . . .
```

Эта процедура очень простая; цвет и толщину линии изменить нельзя:



Пишем более сложную процедуру:

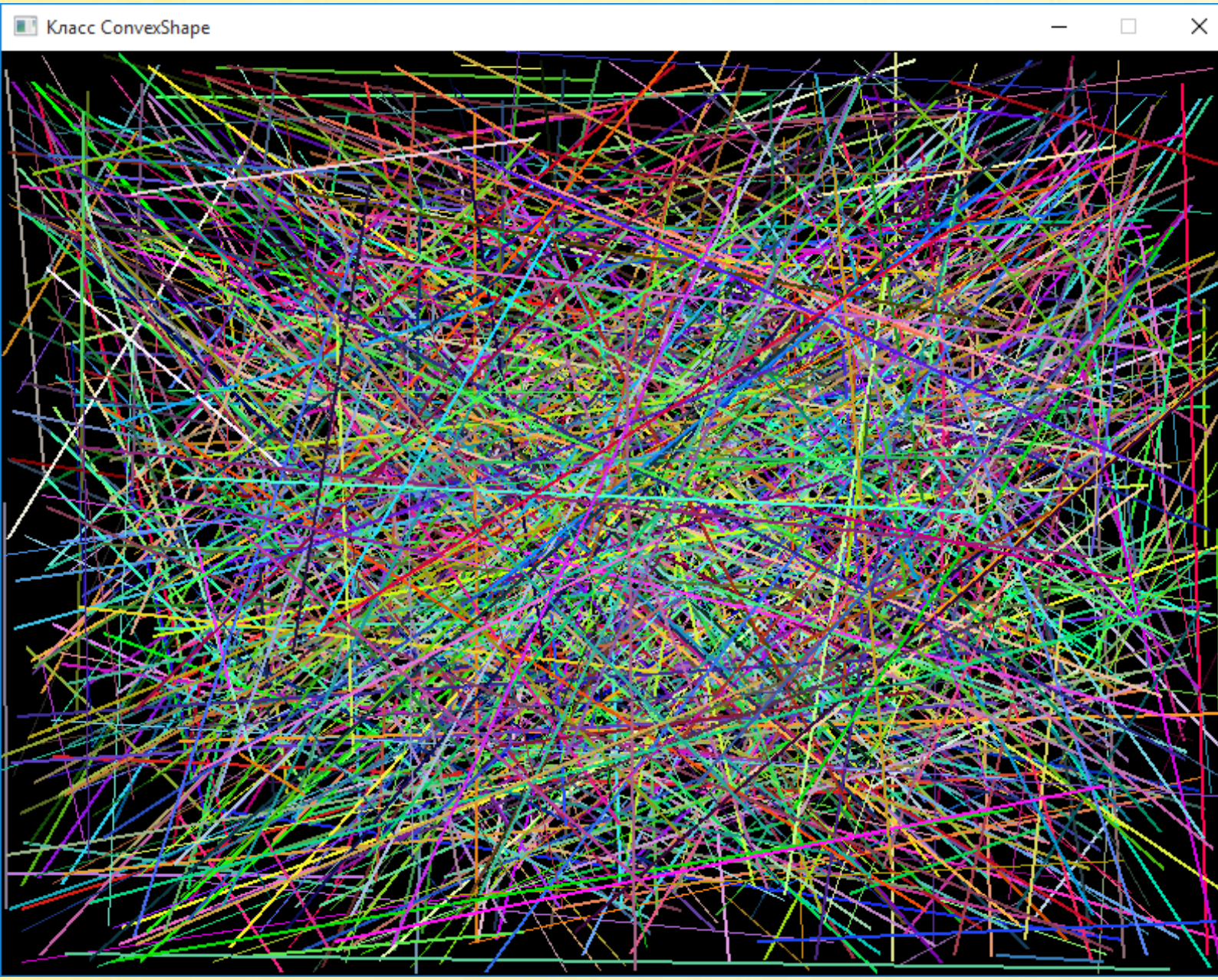
```
// ЧЕРТИМ ОТРЕЗОК ПРЯМОЙ
procedure DrawLine(x1, y1, x2, y2: single; clr: Color);
begin
   var line := new ConvexShape(4);
   line.OutlineColor := clr;
   line.OutlineThickness := 1.2;
   line.SetPoint(0, new Vector2f(x1, y1));
   line.SetPoint(1, new Vector2f(x1, y1));
   line.SetPoint(2, new Vector2f(x2, y2));
   line.SetPoint(3, new Vector2f(x2, y2));
   wind.Draw(line);
end;
```

Здесь уже можно задать **цвет** линии.

Чертим сразу 1000 цветных линий:

```
for var i := 1 to 1000 do
begin
   var x1 := rand.Next(WIDTH);
   var y1 := rand.Next(HEIGHT);
   var x2 := rand.Next(WIDTH);
   var y2 := rand.Next(HEIGHT);
   var clr := GetRandomColor();
   DrawLine(x1, y1, x2, y2, clr);
end;
```

Получилось пёстро и весело:


Класс ConvexShape

Класс VertexArray (Массив вершин)

Класс VertexArray описывает набор 2D-примитивов: точек (вершин), ломаных линий, треугольников и четырёхугольников.

Так точки на экране – это пиксели, то массив вершин вполне можно использовать для рисования пикселями.

Проект Синусоидные полоски

В этом проекте рисунок будут создавать горизонтальные строки одинаково окрашенных пикселей (проще говоря, *линии*, которые мы рисуем отдельными точками). Цвет пикселей изменяется по высоте окна сверху вниз по следующей формуле:

```
var iclr := Ceil(255 * (1 + Math.Sin(y / wl)) / 2);          (1)
```

Поскольку в формуле присутствует *синус* угла, то и проект называется Синусоидные полоски. Частота горизонтальных волн зависит от длины волны wl, так что вы легко получите разные картинки, если измените значение этого параметра. Выражение в скобках изменяется в диапазоне 0..2, а делённое на два – в диапазоне 0..1. Цветные составляющие пикселя должны иметь значение от 0 до 255, поэтому мы умножаем результат в скобках на 255.

Высоту окна можно выбирать произвольно, а ширина должна быть кратна 128, иначе на некоторых компьютерах появляются чёрные вертикальные полосы:

```
const
   // размеры окна:
   WIDTH = 512;
   HEIGHT = 400;
```

В главном блоке создаём окно заданных размеров:

```
begin
   // создаём главное окно:
   CreateWindow(WIDTH, HEIGHT, 'Синусоидные полоски');


// СОЗДАЁМ ОКНО
procedure CreateWindow(width, height: uint64;
                       title: string := 'SFML Window';
                       style: Styles := Styles.Close);
begin
   // создаём окно приложения:
   wind := new RenderWindow(new VideoMode(width, height), title,
style);
   wind.SetVerticalSyncEnabled(true);
```

В программе нам понадобятся обработчики событий для мышки и клавиатуры:

```
// добавляем метод для закрывания окна:
wind.Closed += OnClosed;
// обработчик перемещения мышки:
wind.MouseMoved += Window_MouseMoved;
// обработчик нажатия кнопки мышки:
wind.MouseButtonPressed += Window_MouseButtonPressed;
// обработчик нажатия клавиши:
wind.KeyPressed += OnKeyPressed;
```

А также шрифт для поясняющей надписи:

```
// шрифт:
fnt: Font := nil;
// текст:
txt: Text := nil;

// загружаем шрифт с диска:
fnt := new Font('Media/segmono.ttf');
// создаём текст:
txt := new Text('1 - серый цвет 2 - красный 3 - синий', fnt, 20);
//цвет текста:
txt.Color := Color.Black;
```

```
// координаты текста:
txt.Position := new Vector2f(20, 0);
// стиль текста:
txt.Style := Text.Styles.Bold;
```

Окно создано. Начинается **игровой цикл**.

Прежде всего, нужно нарисовать пиксельную картинку, а затем поверх неё напечать надпись:

```
// игровой цикл:
while (wind.IsOpen) do
begin
   // вызываем все обработчики событий:
   wind.DispatchEvents();

   // рисуем пиксели:
   Draw();

   // печатаем надпись:
   wind.Draw(txt);

   // показываем на экране:
   wind.Display();
end;
```

Так как длину синусоиды мы будем изменять динамически, то её текущее значение сохраняем в переменной **wl**:

```
// длина волны синусоиды:
wl: single := 10;
```

Для вершин сразу создаём **массив** нужного размера:

```
va := new VertexArray(PrimitiveType.Points, WIDTH * HEIGHT);
```

В процедуре Draw заполняем массив вершинами, цвет которых вычисляем по формуле (1). Все полоски – цветные, но монохромные. Номер текущего цвета хранится в переменной nColor:

```
// номер цвета волн:
nColor := 1;
```

Сначала номер цвета равен 1, поэтому полосы получатся серыми.

Чтобы изменить цвет полосок, нужно нажать клавишу 1, 2 или 3, о чём сообщает подсказка на экране:

```
// ЗАДАЁМ ЦВЕТ ПОЛОСОК
procedure OnKeyPressed(sender: object; e: KeyEventArgs);
begin
   if (e.Code = Keyboard.Key.Num1) then
       nColor := 1
   else if (e.Code = Keyboard.Key.Num2) then
       nColor := 2
   else if (e.Code = Keyboard.Key.Num3) then
       nColor := 3
   else nColor := 1;
end;


// РИСУЕМ ПИКСЕЛИ
procedure Draw();
begin
    var clr: Color;
    //рисуем волны на картинке:
    for var y := 0 to HEIGHT-1 do
    begin
        //цвет очередного пикселя:
        var iclr := Ceil(255 * (1 + Math.Sin(y / wl)) / 2);
        //серные полоски:
        if (nColor = 1) then
           clr := new Color(iclr, iclr, iclr)
        //красные полоски:
        else if (nColor = 2) then
            clr := new Color(iclr, 0, 0)
```

```
        else
            //синие полоски:
            clr := new Color(0, 0, iclr);
```

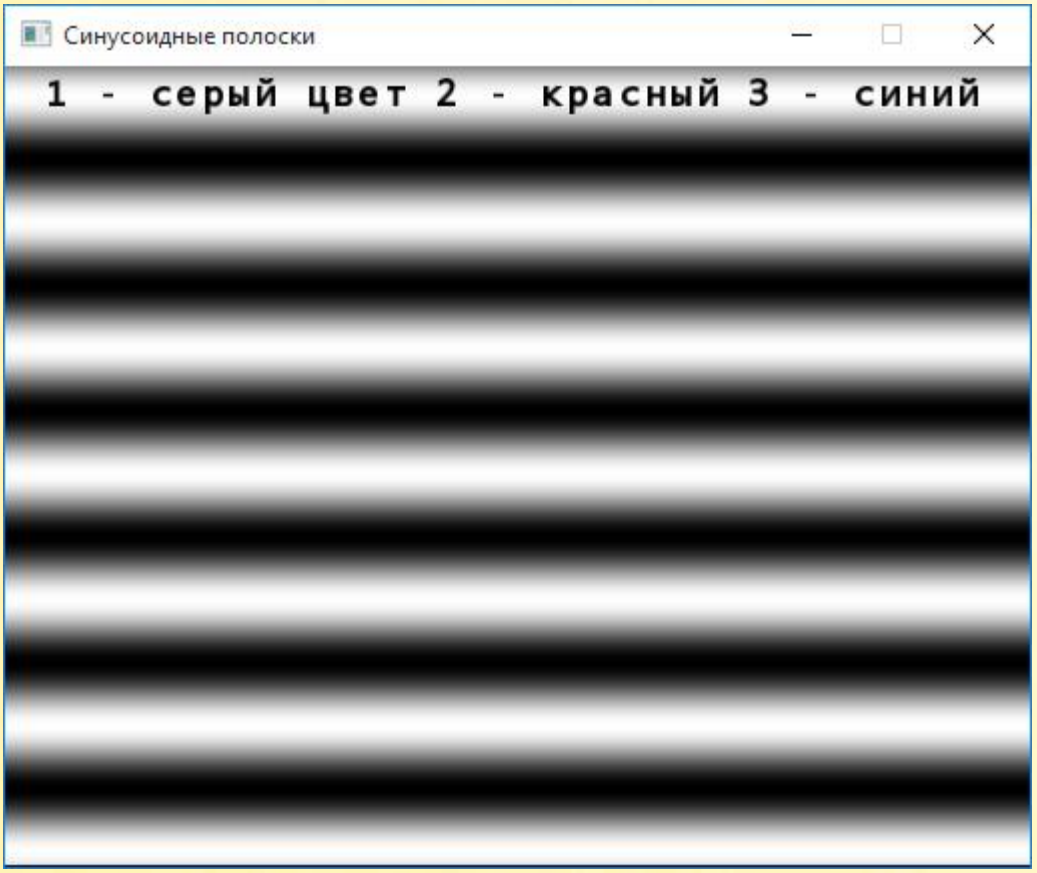
Создаём новую вершину цвета *clr*. Её координаты равны *(x,y)*, а индекс - *y * WIDTH + x*:

```
    for var x := 0 to WIDTH-1 do
     begin
        var v := new Vertex(new Vector2f(x, y), clr);
        va[(y * WIDTH + x)] := v;
     end;
end;
```

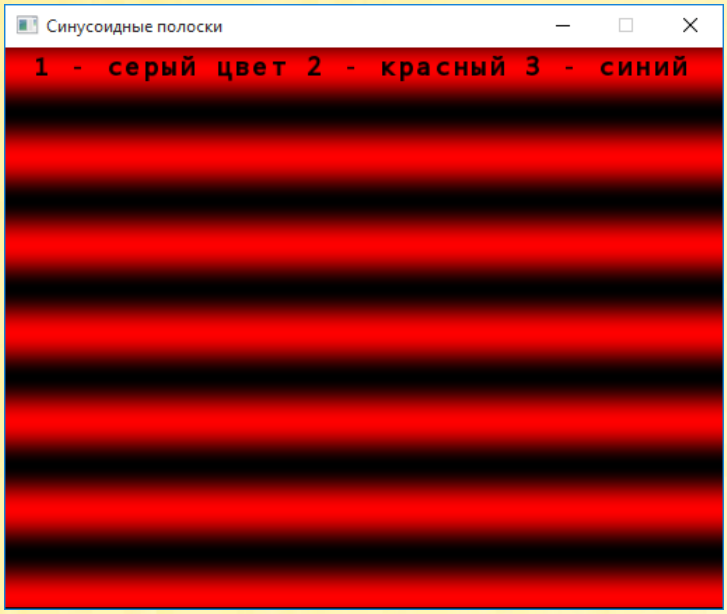
Массив вершин заполнен, и можно его показать на экране:

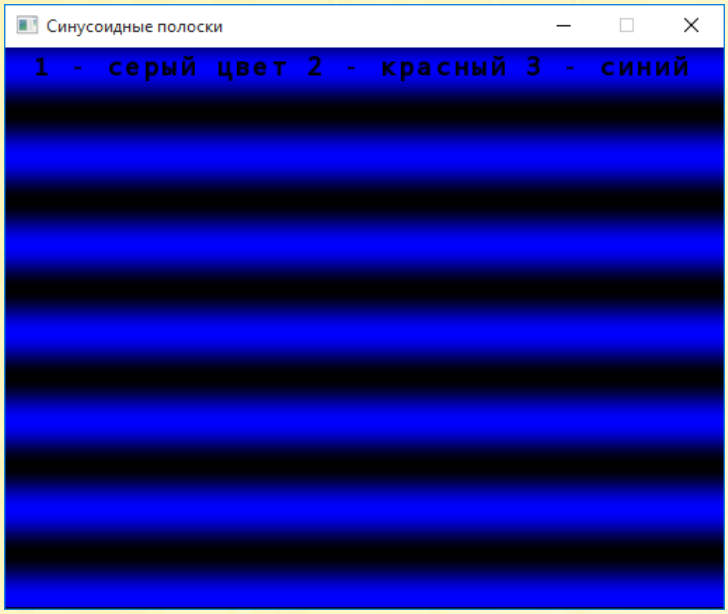
```
    wind.Draw(va);
end;
```

Вот такая получилась волнистая картинка:



Нажимаем цифровые клавиши – и волны становятся цветными!





Но длину волны так просто не изменить. Для этого нужно нажать кнопку **мышки**.

В процедуре **Window_MouseButtonPressed** мы запоминаем в переменной **mousePos** координаты мышиного курсора:

```
// координаты мышки:
mousePos: Vector2i;

// НАЖИМАЕМ КНОПКУ МЫШКИ
procedure Window_MouseButtonPressed(sender: Object; e: MouseButtonE-
ventArgs);
begin
   // запоминаем координаты мышки:
   var mouseX := e.X;
   var mouseY := e.Y;
   mousePos := new Vector2i(mouseX, mouseY);
end;
```

Теперь **перемещаем мышку** вверх-вниз, удерживая левую кнопку:

```
// ПЕРЕМЕЩАЕМ МЫШКУ
procedure Window_MouseMoved(sender: Object; e: MouseMoveEventArgs);
begin
   if (not Mouse.IsButtonPressed(Mouse.Button.Left)) then
```

```
        exit;
```

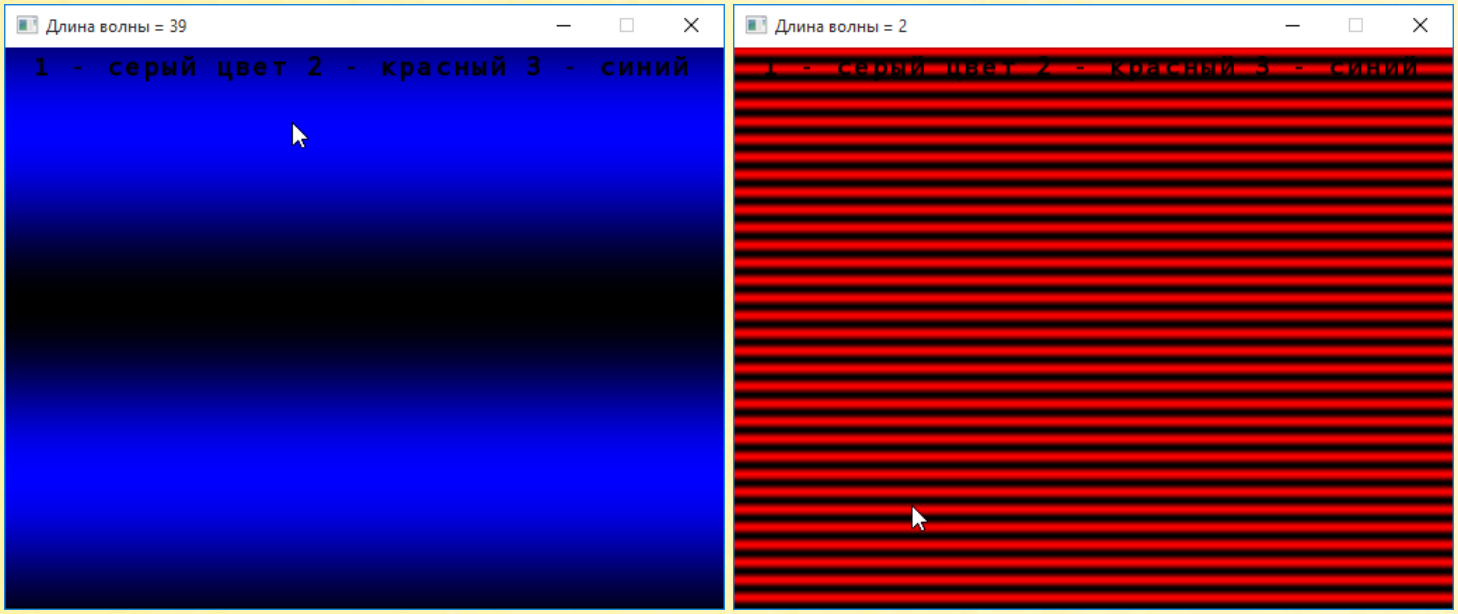
Находим расстояние по вертикали, на которое переместилась мышка:

```
// текущие координаты мышки:
var x := e.X;
var y := e.Y;
var dy := y - mousePos.Y;
```

Если оно отрицательное (мышка побежала вверх), то длина волны увеличивается, а если положительное – уменьшается:

```
  if (dy < 0) then
       wl += 1
   else wl -= 1;
   if(wl < 1) then
        wl := 1;
   wind.SetTitle('Длина волны = ' + wl);
   // запоминаем новые координаты мышки:
   mousePos := new Vector2i(x, y);
end;
```

Мышкой (или даже пальцем на сенсорном экране) вы можете пускать волны любой длины:



Проект *Синусоидные полоски 2*

Неподвижные волны очень скучны, даже цветные. Но их легко привести в движение, то есть **анимировать**.

Добавим в процедуру **Draw** параметр **t** – *условное время*:

```
// РИСУЕМ ПИКСЕЛИ
procedure Draw(t : double := 0);
begin
    var clr: Color;
    //рисуем волны на картинке:
    for var y := 0 to HEIGHT-1 do
    begin
        //цвет очередного пикселя:
        var iclr := Ceil(255 * (1 + Math.Sin(y / wl + t)) / 2);
```

Теперь волны будут двигаться **вниз**, если время отрицательное, или **вверх** – если **положительное**:

```
// время:
var t := 0.0;
```

```
// игровой цикл:
while (wind.IsOpen) do
begin
   // вызываем все обработчики событий:
   wind.DispatchEvents();

   // рисуем пиксели:
   Draw(t);
   t -= 0.9;
```

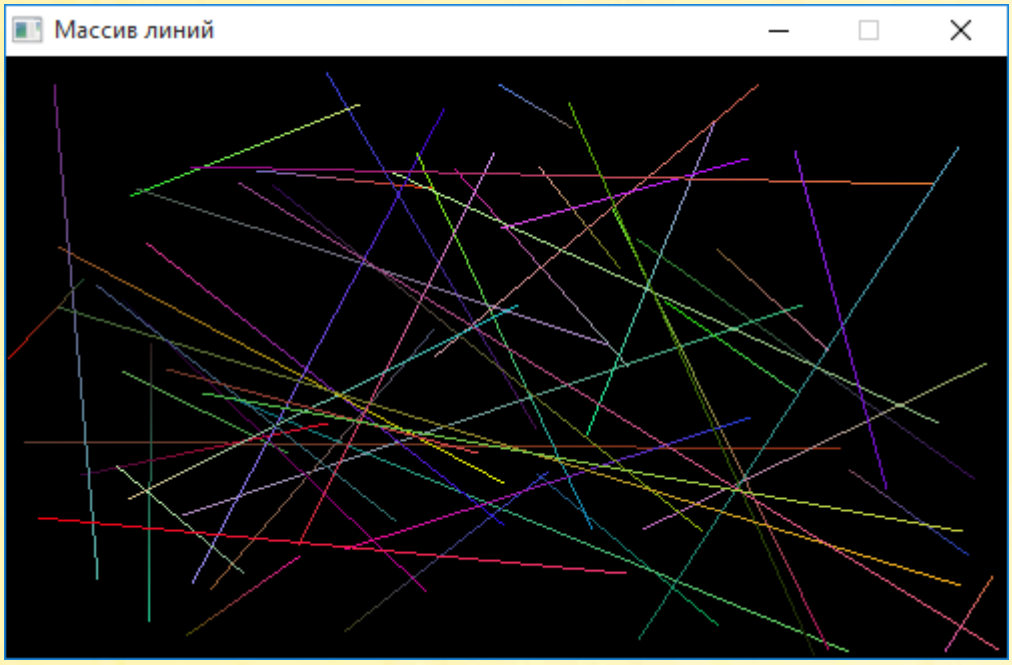
Проект *Массив линий*

Вершины в массиве VertexArray можно использовать по-разному. Если тип вершин - PrimitiveType.Points, то каждая вершина будет нарисована как пиксель на экране. Но если тип элементов массива - PrimitiveType.Lines, то каждая пара точек будет задавать координаты начала или конца отрезка прямой. Так как каждая вершина имеет собственный цвет, то можно получить градиентно раскрашенные прямые, что очень красиво. Но все линии имеют толщину в 1 пиксель, поэтому очень тонкие.

Давайте начертим 50 разноцветных линий, а для этого нам понадобится массив из 100 точек:

```
// РИСУЕМ ЛИНИИ
procedure Draw();
begin
   var va := new VertexArray(PrimitiveType.Lines);
   for var i := 1 to 100 do
   begin
       var pos := new Vector2f(rand.Next(WIDTH), rand.Next(HEIGHT));
       var v := new Vertex(pos, GetRandomColor());
       va.Append(v);
   end;
   wind.Draw(va);
end;

begin
   // создаём главное окно:
   CreateWindow(WIDTH, HEIGHT, 'Массив линий');

   // рисуем линии:
   Draw();

   // показываем на экране:
   wind.Display();

   // игровой цикл:
   while (wind.IsOpen) do
   begin
      // вызываем все обработчики событий:
      wind.DispatchEvents();
```

```
    end;
end.
```



Проект *Градиент*

Если чертить горизонтальные линии вплотную друг к другу, то можно получить **градиентную заливку**, в которой один цвет *плавно* переходит в другой. Этот приём часто применяется в заставках, которые появляются на экране при установке программ, а также в компьютерной графике для раскрашивания кнопок, баннеров и других объектов.

Хорошим примером натуральной «градиентной заливки» может служить безоблачное полуденное или закатное небо!

Для сокращения имён и надписей обозначим цвета первыми русскими и латинскими буквами:

Ж – жёлтый - Y

К – красный - R

Ц – циан - C

С – синий - B

Л – лиловый - M

При переходе от циана к синему цвету *синяя* составляющая текущего цвета всегда равна 255 (максимальное значение), *красная* – нулю (отсутствует вообще), а *зелёная* составляющая постепенно уменьшает своё значение при перемещении линий сверху вниз от 255 до 0. Именно благодаря этому изменению и возникает плавный переход цвета по вертикали. Так как по горизонтали цвет остается без изменений, то достаточно провести горизонтальную линию текущего цвета:

```
// ПЕРЕХОД ОТ ЦИАНА  К СИНЕМУ - CYAN TO BLUE
procedure CB();
begin
   va := new VertexArray(PrimitiveType.Lines, HEIGHT);
   var b := 255;
   var r := 0;
   for var y := 0 to HEIGHT-1 do
   begin
      // вычисляем интенсивность зелёной составляющей цвета:
      var g := Ceil(255 * (1 - y / HEIGHT));
      // задаём цвет линии:
      var clr := new Color(r, g, b);
      // координаты начала и конца линии:
      var v := new Vertex(new Vector2f(0, y), clr);
      va.Append(v);
      v := new Vertex(new Vector2f(WIDTH, y), clr);
      va.Append(v);
   end;
end;
```

Всего в программе 6 цветных процедур, которые мало отличаются друг от друга, поэтому остальные процедуры вы легко напишете их сами или посмотрите в исходном коде.

Градиент выбирается нажатием цифровой клавиши 1..6:

```
// ЗАДАЁМ ЦВЕТ ПОЛОСОК
procedure OnKeyPressed(sender: object; e: KeyEventArgs);
begin

   case (e.Code) of
       Keyboard.Key.Num1: CB();
       Keyboard.Key.Num2: YG();
       Keyboard.Key.Num3: CG();
       Keyboard.Key.Num4: YR();
       Keyboard.Key.Num5: MB();
       Keyboard.Key.Num6: MR();
       else CB();
   end;
end;
```

В процедуре CreateWindow мы загружаем шрифт с диска и формируем надпись-подсказку:

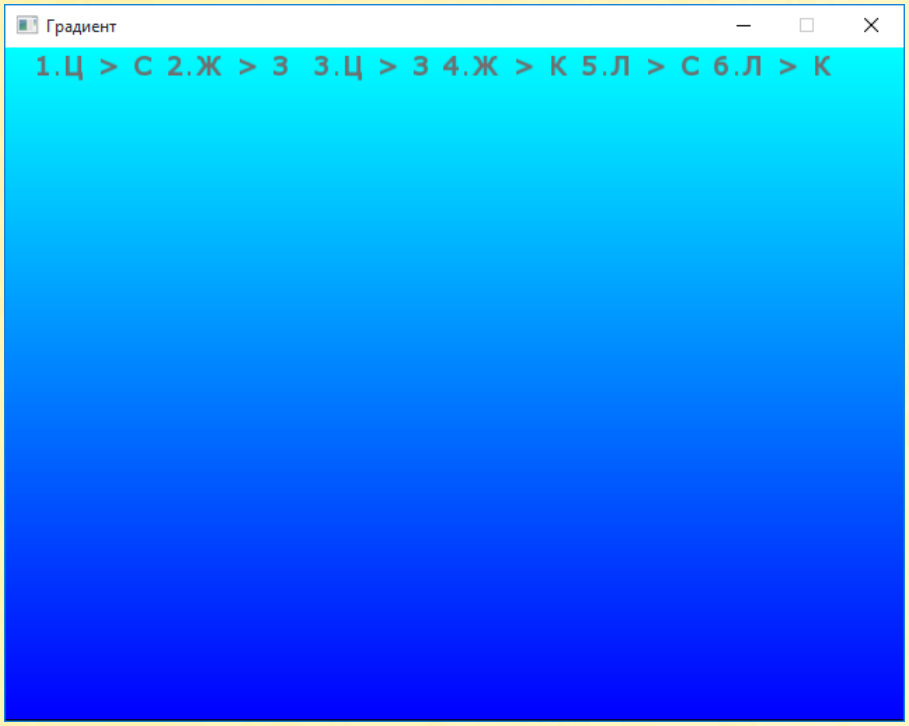
```
// СОЗДАЁМ ОКНО
procedure CreateWindow(width, height: uint64;
                       title: string := 'SFML Window';
                       style: Styles := Styles.Close);
begin
   // создаём окно приложения:
   wind := new RenderWindow(new VideoMode(width, height), title,
style);
   wind.SetVerticalSyncEnabled(true);

   // добавляем метод для закрывания окна:
   wind.Closed += OnClosed;
   // обработчик нажатия клавиши:
   wind.KeyPressed += OnKeyPressed;

   // загружаем шрифт с диска:
   fnt := new Font('Media/verdana.ttf');
```

```
    // создаём текст:
    txt := new Text('1.Ц > С 2.Ж > З  3.Ц > З 4.Ж > К 5.Л > С 6.Л > К',
                    fnt, 20);
    //цвет текста:
    txt.Color := new Color(111,111,111);
    // координаты текста:
    txt.Position := new Vector2f(20, 0);
    // стиль текста:
    txt.Style := Text.Styles.Bold;
end;
```

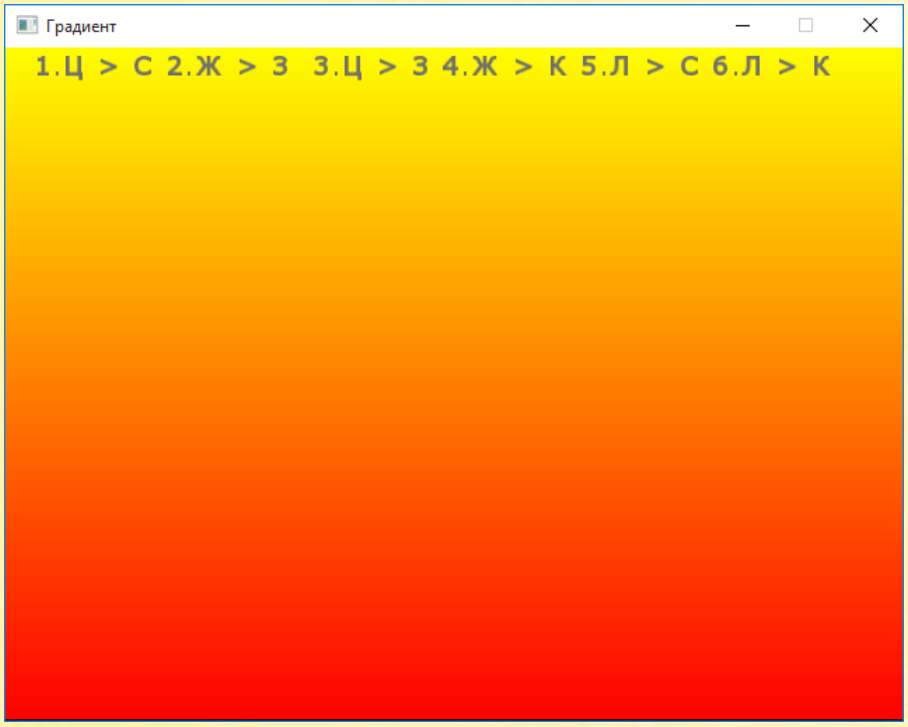
В главном блоке мы сразу создаём градиент СВ, а затем рисуем его на экране:

```
begin
    // создаём главное окно:
    CreateWindow(WIDTH, HEIGHT, 'Градиент');

    // создаём градиент:
    CB();

    // игровой цикл:
    while (wind.IsOpen) do
    begin
        // вызываем все обработчики событий:
        wind.DispatchEvents();

        // рисуем градиент на экране:
        wind.Draw(va);

        // печатаем надпись:
        wind.Draw(txt);

        // показываем на экране:
        wind.Display();
    end;

end.
```

Получился вполне милый градиент:



Одно нажатие на клавишу – и синий градиент превращается в красный:



Точно так же можно было бы нарисовать и синусоидные полоски, но они были нужны нам как упражнение при изучении вершин-пикселей.

Проект *Градиент 2*

Элементы в массиве вершин могут иметь тип `PrimitiveType.Quads`, то есть **четырёхугольник**, и тогда мы сможем сразу создать градиент во всё окно приложения.

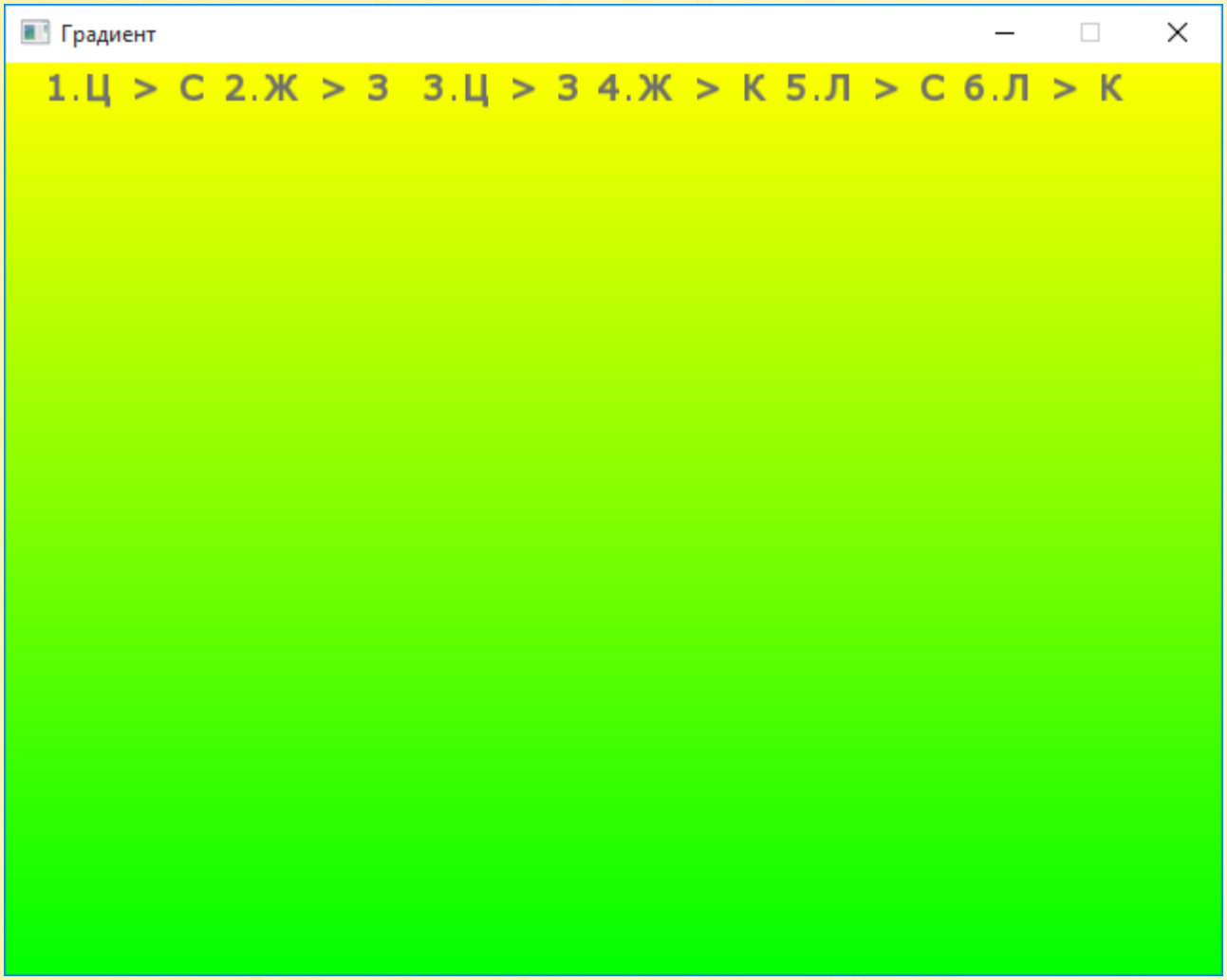
Четырёхугольник задаётся **четырьмя вершинами**. Пара верхних вершин имеет начальный цвет градиента, а пара нижних – конечный.

> При задании координат нужно обходить все вершины в одном направлении!

Опять для образца возьмём первый градиент:

```
// ПЕРЕХОД ОТ ЦИАНА  К СИНЕМУ - CYAN TO BLUE
procedure CB();
begin
   va := new VertexArray(PrimitiveType.Quads, HEIGHT);
   var b := 255;
   var r := 0;
   var g := 255;
   var clr := new Color(r, g, b);

   // координаты верхних вершин:
   var v := new Vertex(new Vector2f(0, 0), clr);
   va.Append(v);
   v := new Vertex(new Vector2f(WIDTH, 0), clr);
   va.Append(v);

   g := 0;
   clr := new Color(r, g, b);
   // координаты нижних вершин:
   v := new Vertex(new Vector2f(WIDTH, HEIGHT), clr);
   va.Append(v);
   v := new Vertex(new Vector2f(0, HEIGHT), clr);
   va.Append(v);
end;
```

Все остальные градиенты конструируются аналогично. Сами градиенты остаются прежними:



Проект *Градиент 3*

Очень легко получить **случайный градиент**, если для верхних и нижних вершин выбирать случайный цвет.

Программа упрощается, но нужно внести в неё несколько добавлений и исправлений.

Подсказка должна быть такая:

```
// создаём текст:
txt := new Text('Нажмите ПРОБЕЛ', fnt, 20);
```

Можно нажимать любую клавишу, но удобнее пользоваться клавишей ПРОБЕЛ, потому что она всегда под рукой.

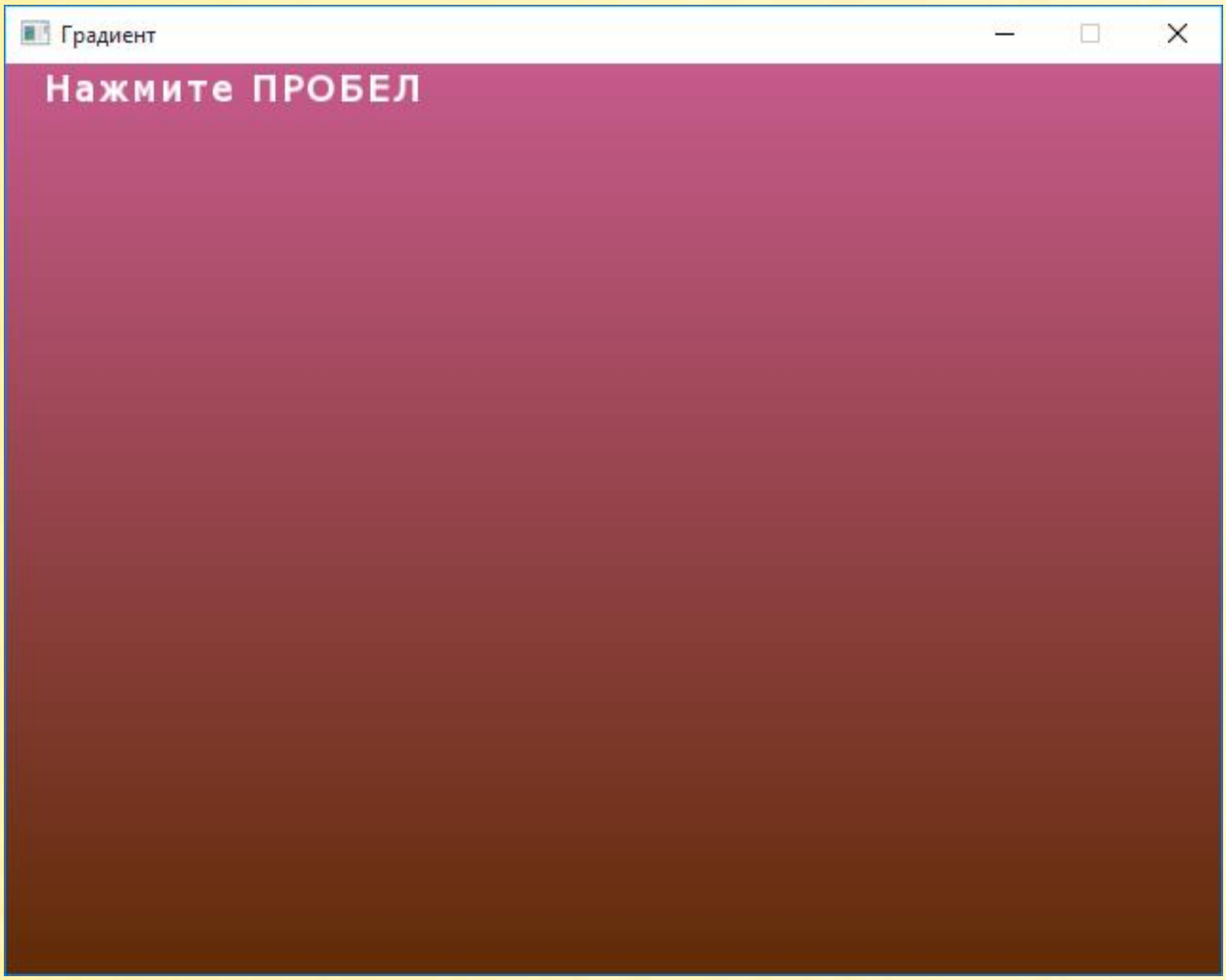
Нажали ПРОБЕЛ – вызвали процедуру CreateRandomGradient:

```
// НАЖИМАЕМ КЛАВИШУ ПРОБЕЛ
procedure OnKeyPressed(sender: object; e: KeyEventArgs);
begin
   if (e.Code = Keyboard.Key.Space) then
       CreateRandomGradient();
end;
```

А создать случайный градиент проще простого:

```
// СОЗДАЁМ СЛУЧАЙНЫЙ ГРАДИЕНТ
procedure CreateRandomGradient();
begin
    va := new VertexArray(PrimitiveType.Quads, HEIGHT);
    // случайный цвет:
    var clr := GetRandomColor();

    // координаты верхних вершин:
    var v := new Vertex(new Vector2f(0, 0), clr);
    va.Append(v);
    v := new Vertex(new Vector2f(WIDTH, 0), clr);
    va.Append(v);

    clr := GetRandomColor();
    // координаты нижних вершин:
    v := new Vertex(new Vector2f(WIDTH, HEIGHT), clr);
    va.Append(v);
    v := new Vertex(new Vector2f(0, HEIGHT), clr);
    va.Append(v);
end;
```

Не всегда случайные градиенты получаются красивыми, но, нажимая клавишу ПРОБЕЛ, вы дождётесь и совсем неплохих картинок:



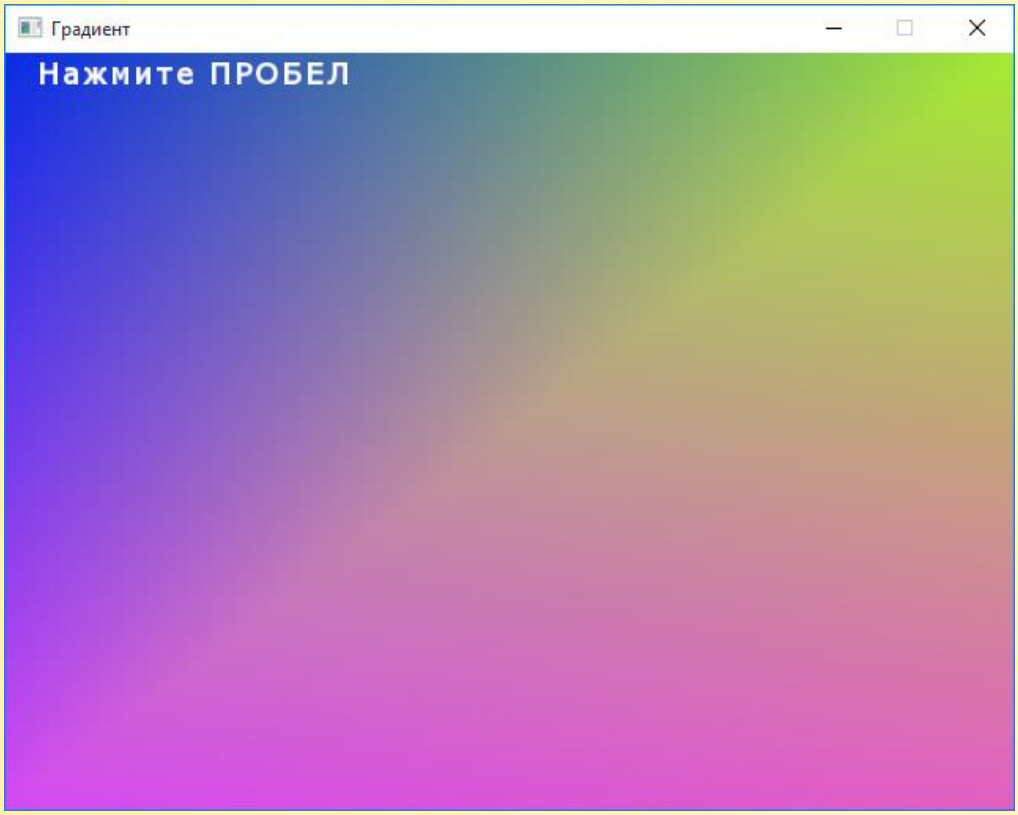
Проект *Градиент 4*

Достаточно добавить в процедуру CreateRandomGradient несколько строк, чтобы получить *сложный градиент*:

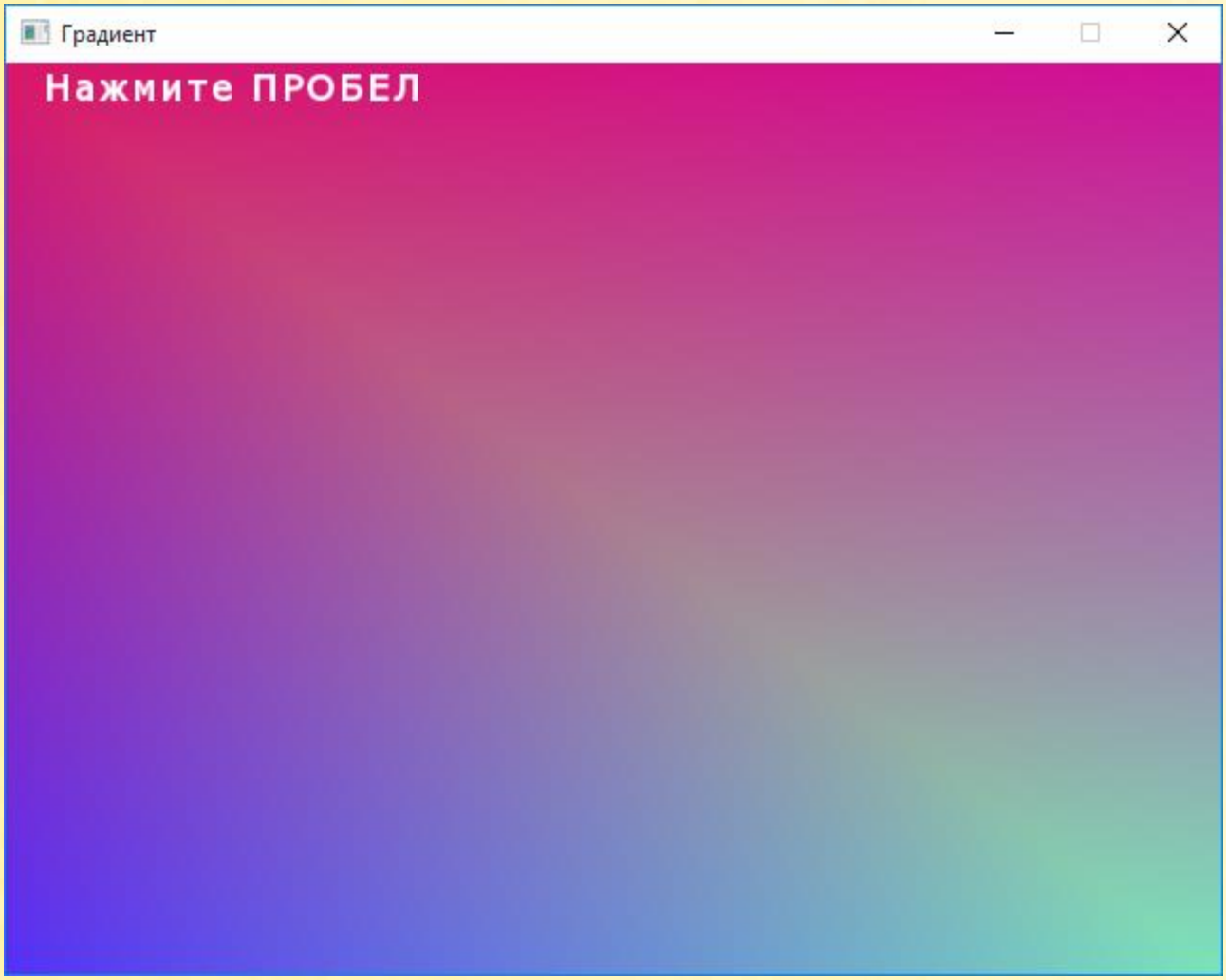
```
// СОЗДАЁМ СЛУЧАЙНЫЙ ГРАДИЕНТ
procedure CreateRandomGradient();
begin
```

```
    va := new VertexArray(PrimitiveType.Quads, HEIGHT);
    // случайный цвет:
    var clr := GetRandomColor();
    // координаты верхних вершин:
    var v := new Vertex(new Vector2f(0, 0), clr);
    va.Append(v);
    clr := GetRandomColor();
    v := new Vertex(new Vector2f(WIDTH, 0), clr);
    va.Append(v);

    clr := GetRandomColor();
    // координаты нижних вершин:
    v := new Vertex(new Vector2f(WIDTH, HEIGHT), clr);
    va.Append(v);
    clr := GetRandomColor();
    v := new Vertex(new Vector2f(0, HEIGHT), clr);
    va.Append(v);
end;
```

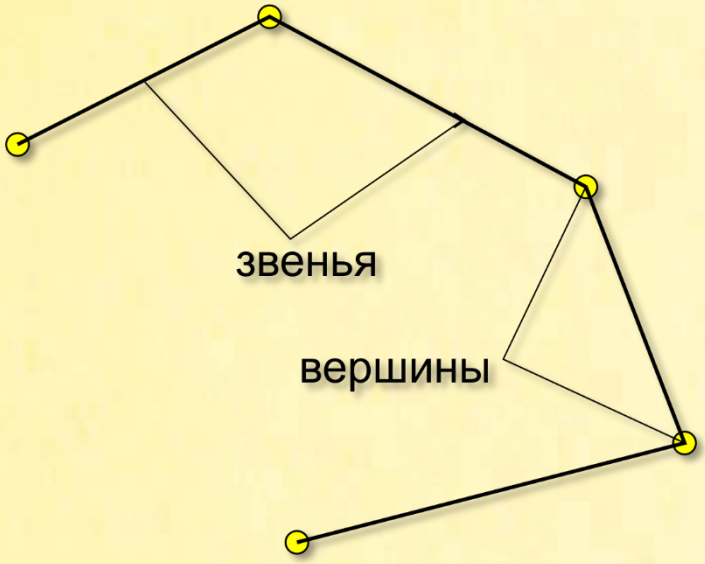
Действительно, каждая вершина прямоугольника может иметь собственный цвет, и тогда все 4 цвета будут перемешиваться по всему окну приложения:



Если обходить вершины против часовой стрелки, то цветная диагональ изменит направление:



Если задать элементам массива VertexArray тип `PrimitiveType.LinesStrip`, то можно вычертить не отдельные отрезки прямых, а **ломаные**. Первые 2 вершины в массиве определяют координаты первого отрезка прямой. Второй отрезок начинается из конца первого, а третья вершина в массиве задаёт координаты его конца. Третий отрезок начинается из конца второго, а четвёртая вершина в массиве хранит координаты его конца. И так далее, пока в массиве не закончатся все вершины:



Если звенья ломаной очень короткие, то она превращается в гладкую кривую. В следующем проекте мы будем вычерчивать кривые, которые представляют собой ломаные с очень короткими звеньями.

Проект *Интерактивный градиент*

Давайте избавимся от случайности в проекте *Градиент 3*!

Верхний и нижний цвет градиента сохраним в глобальных переменных:

```
// цвета градиента:
top := 0;
bottom := 0;
```

В процедуре CreateWindow создаём обработчики событий для мышки и поясняющую надпись:

```
// СОЗДАЁМ ОКНО
procedure CreateWindow(width, height: uint64;
                       title: string := 'SFML Window';
                       style: Styles := Styles.Close);
```

```
begin
   // создаём окно приложения:
   wind := new RenderWindow(new VideoMode(width, height), title,
                            style);
   wind.SetVerticalSyncEnabled(true);

   // добавляем метод для закрывания окна:
   wind.Closed += OnClosed;
   // обработчик перемещения мышки:
   wind.MouseMoved += Window_MouseMoved;
   // обработчик нажатия кнопки мышки:
   wind.MouseButtonPressed += Window_MouseButtonPressed;

   // загружаем шрифт с диска:
   fnt := new Font('Media/verdana.ttf');
   // создаём текст:
   txt := new Text('МЫШКА: ВВЕРХ-ВНИЗ | ВЛЕВО-ВПРАВО', fnt, 24);
   //цвет текста:
   txt.Color := Color.Black;
   // координаты текста:
   txt.Position := new Vector2f(20, 0);
   // стиль текста:
   txt.Style := Text.Styles.Bold;
end;
```

В процедуре Window_MouseMoved изменяем значения цветов градиента:

```
// ПЕРЕМЕЩАЕМ МЫШКУ
procedure Window_MouseMoved(sender: Object; e: MouseMoveEventArgs);
begin
   if (not Mouse.IsButtonPressed(Mouse.Button.Left)) then
       exit;

   // текущие координаты мышки:
   var x := e.X;
   var y := e.Y;

   // расстояние:
   var dx := x - mousePos.X;
   var dy := y - mousePos.Y;

   // новые цвета:
   top += dy;
```

```
    bottom += dx;

    // запоминаем новые координаты мышки:
    mousePos := new Vector2i(x, y);

    // создаём градиент:
    CreateGradient();
end;
```

В процедуре CreateGradient создаём новый градиент по текущим значениям переменных *top* и *bottom*:

```
// СОЗДАЁМ ГРАДИЕНТ
procedure CreateGradient();
begin
    va := new VertexArray(PrimitiveType.Quads, HEIGHT);
    // цвет сверху:
    var clr := HsvToRgb(top);

    // координаты верхних вершин:
    var v := new Vertex(new Vector2f(0, 0), clr);
    va.Append(v);
    v := new Vertex(new Vector2f(WIDTH, 0), clr);
    va.Append(v);

    // цвет снизу:
    clr := HsvToRgb(bottom);
    // координаты нижних вершин:
    v := new Vertex(new Vector2f(WIDTH, HEIGHT), clr);
    va.Append(v);
    v := new Vertex(new Vector2f(0, HEIGHT), clr);
    va.Append(v);
end;
```

И наконец, в главном блоке рисуем градиент на экране и печатаем надпись:

```
begin
    // создаём главное окно:
    CreateWindow(WIDTH, HEIGHT, 'Интерактивный градиент');
```

```
    // создаём градиент:
    CreateGradient();

    // игровой цикл:
    while (wind.IsOpen) do
    begin
        // вызываем все обработчики событий:
        wind.DispatchEvents();
        // рисуем градиент на экране:
        wind.Draw(va);
        // печатаем надпись:
        wind.Draw(txt);
        // показываем на экране:
        wind.Display();
    end;
end.
```

Теперь любой пользователь может создавать собственные градиенты:

Проект *Горизонтальный градиент*

Из вертикального градиента легко получить горизонтальный.

Переименуем названия переменных для цветов градиента:

```
// цвета градиента:
left := 0;
right := 0;
```

И перепишем процедуру CreateGradient так, чтобы левые вершины прямоугольника имели цвет clrLeft, а правые – цвет clrRight:

```
// СОЗДАЁМ ГРАДИЕНТ
procedure CreateGradient();
begin
    va := new VertexArray(PrimitiveType.Quads, HEIGHT);

    // цвет слева:
    var clrLeft := HsvToRgb(left);
    // цвет справа:
    var clrRight := HsvToRgb(right);

    // левая верхняя вершина:
    var v := new Vertex(new Vector2f(0, 0), clrLeft);
    va.Append(v);
    // правая верхняя вершина:
    v := new Vertex(new Vector2f(WIDTH, 0), clrRight);
    va.Append(v);
    // правая нижняя вершина:
    v := new Vertex(new Vector2f(WIDTH, HEIGHT), clrRight);
    va.Append(v);
    // левая нижняя вершина:
    v := new Vertex(new Vector2f(0, HEIGHT), clrLeft);
    va.Append(v);
end;
```

Если вы не запутались в вершинах, то теперь сможете в интерактивном режиме создавать и горизонтальные градиенты:

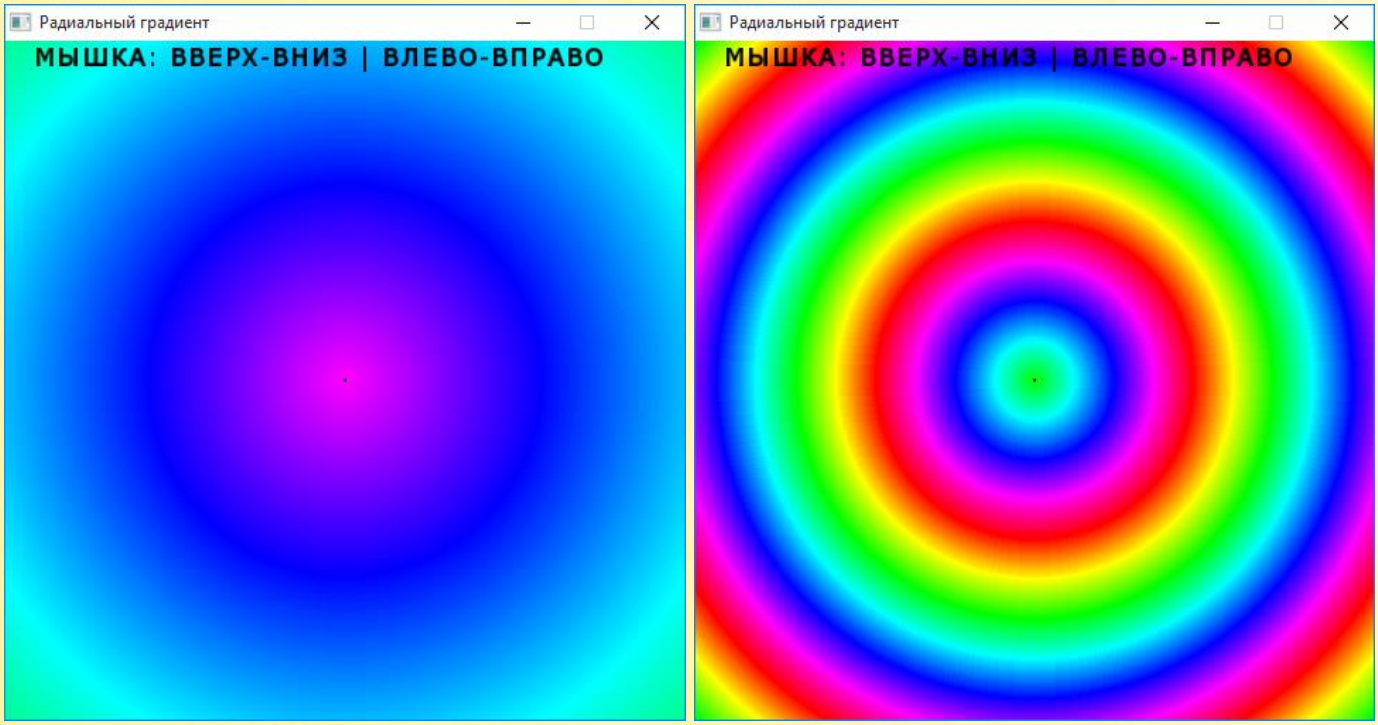
Проект *Радиальный градиент*

Проще всего создать радиальный градиент, вычерчивая окружности, начиная от центра:

```
// СОЗДАЁМ ГРАДИЕНТ
procedure CreateGradient();
begin
   wind.Clear(HsvToRgb(center));
```

```
    var circle := new CircleShape(1,200);
    circle.OutlineThickness := 2;
    circle.Position := new Vector2f(WIDTH div 2, HEIGHT div 2);
    circle.FillColor := Color.Transparent;
    var maxR := Ceil(HEIGHT / 1.4);
    for var r := 0 to maxR do
    begin
       var clr := center + (perif-center) / maxR * r;
       var col := HsvToRgb(clr);
       circle.OutlineColor := col;
       circle.Origin := new Vector2f(r, r);
       circle.Radius := r;
       wind.Draw(circle);
    end;
    // печатаем надпись:
    wind.Draw(txt);
    // показываем на экране:
    wind.Display();
end;
```

Градиент получается многоцветным, что красиво:



Но если вам нужен «двухцветный» градиент, то измените формулу для расчёта промежуточных цветов.

Проект *Параметрические кривые*

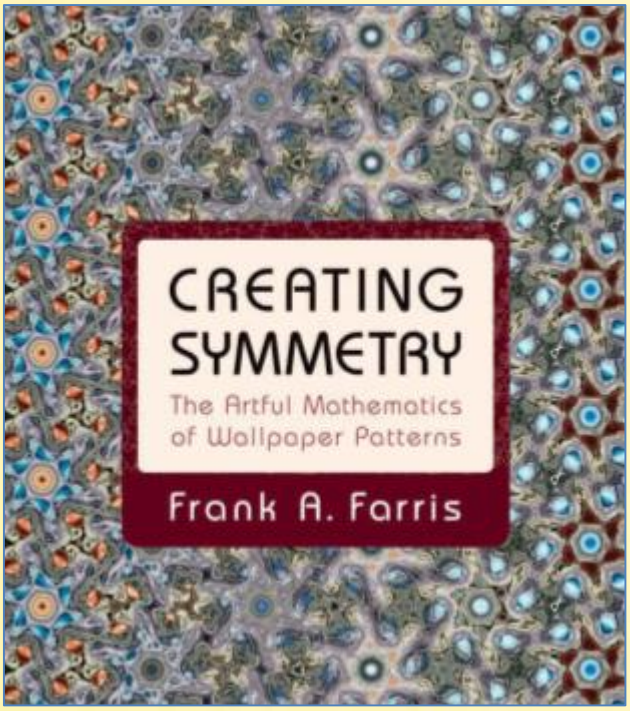
Кривая линия может быть задана **параметрическими уравнениями**. Для плоской кривой понадобятся 2 уравнения – для каждой из координат. В этих уравнениях координаты – это **функции** от некоторой переменной *t*.

В этом проекте мы рассмотрим параметрические кривые, которые задаются такими **уравнениями**:

```
var x = Cos(A * t) + Cos(B * t) / 2 + Sin(C * t) / 3;
var y = Sin(A * t) + Sin(B * t) / 2 + Cos(C * t) / 3;
```

Параметр t изменяется в диапазоне **0..2π**, а коэффициенты **A**, **B** и **C** – это произвольные целые числа.

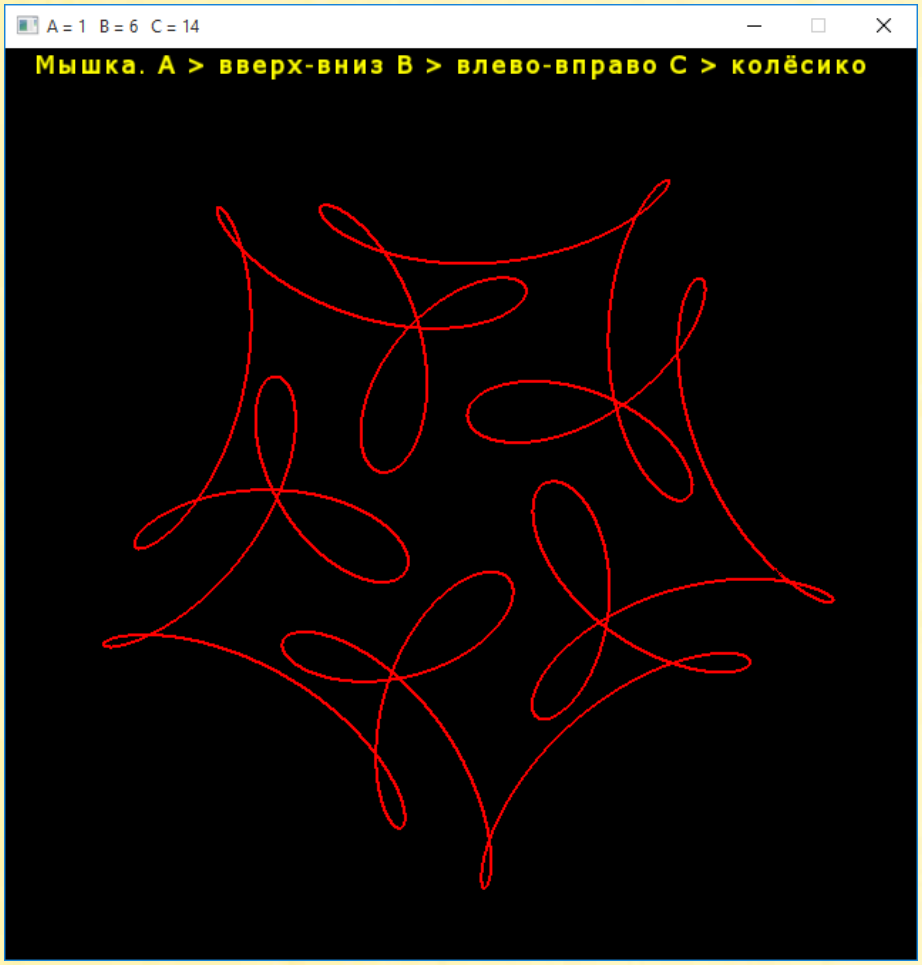
Эти уравнения вы можете найти в книге Фрэнка Фарриса (Frank A. Farris) *Creating Symmetry: The Artful Mathematics of Wallpaper Patterns:*



В ней **параметры** равны:

A = 1
B = 6
C = 14

Им соответствует кривая на рисунке:



Но нам, конечно, хотелось бы посмотреть и на другие кривые – может быть, среди них найдутся ещё более интересные!

Параметрические уравнения имеют 3 параметра:

```
// коэффициенты:
A := 1;
B := 6;
```

```
C := 14;
```

Их нужно изменять в интерактивном режиме. Обычно для этого используют ползунки, но для трёх параметров вполне достаточно и одной мышки! Перемещения мышки **вверх-вниз** изменяют параметр A, **влево-вправо** – параметр B, а **верчение колёсика** – параметр C. Перемещения мышки учитываем, только если нажата кнопка.

В процедуре CreateWindow добавляем процедуры для обработки мышиных событий, а также формируем строковую подсказку:

```
// СОЗДАЁМ ОКНО
procedure CreateWindow(width, height: uint64;
                       title: string := 'SFML Window';
                       style: Styles := Styles.Close);
begin
   // создаём окно приложения:
   wind := new RenderWindow(new VideoMode(width, height), title,
                            style);
   wind.SetVerticalSyncEnabled(true);

   // добавляем метод для закрывания окна:
   wind.Closed += OnClosed;
   // обработчик перемещения мышки:
   wind.MouseMoved += Window_MouseMoved;
   // обработчик нажатия кнопки мышки:
   wind.MouseButtonPressed += Window_MouseButtonPressed;
   // обработчик колёсика мышки:
   wind.MouseWheelMoved += Window_MouseWheelMoved;


   // загружаем шрифт с диска:
   fnt := new Font('Media/verdana.ttf');
   // создаём текст:
   txt := new Text('Мышка. A > вверх-вниз B > влево-вправо C >
                    колёсико', fnt, 18);
   //цвет текста:
   txt.Color := Color.Yellow;
   // координаты текста:
   txt.Position := new Vector2f(20, 0);
   // стиль текста:
```

```
    txt.Style := Text.Styles.Bold;
end;
```

По заданным параметрам A, B и C создаём в процедуре CreateCurve массив вершин типа PrimitiveType.LinesStrip:

```
// СОЗДАЁМ КРИВУЮ
procedure CreateCurve();
begin
    va := new VertexArray(PrimitiveType.LinesStrip);
```

Для окна я выбрал чёрный фон:

```
const
    // размеры окна:
    WIDTH = 640;
    HEIGHT = 640;
    BACKGROUND = Color.Black;
```

А для линий – красный цвет:

```
//задаём цвет линии:
var clr := new Color(255, 0, 0);
```

Но это дело вкуса, и вы можете выбрать свои цвета.

При расчёте по формулам значения координат получаются очень маленькими, поэтому их следует умножить на **коэффициент** *k*, значение которого нужно подобрать опытным путём:

```
// коэффициент увеличения:
var k := 150;
```

Чтобы кривая получилась **гладкой**, нужно задать параметру *t* небольшое приращение **step**, которое зависит от значения параметров кривой:

```
// шаг:
var maxk := Max(Max(A, B), C);
var step := Min(1.0/maxk/8, 0.01);
```

Если коэффициенты небольшие, то мы чертим утолщённую кривую, иначе толщина кривой равна 1 пикселю:

```
// толстая линия:
var thick := true;
if (maxk > 40) then
    thick := false;
```

В цикле while вычисляем значения координат точек кривой и добавляем их в массив va:

```
// добавляем вершины в массив:
var t := 0.0;
while t < PI*2 do
begin
   var x := Cos(A * t) + Cos(B * t) / 2 + Sin(C * t) / 3;
   var y := Sin(A * t) + Sin(B * t) / 2 + Cos(C * t) / 3;
   var v := new Vertex(new Vector2f(x*k + WIDTH / 2,
                                    y * k + HEIGHT / 2), clr);
   va.Append(v);

   t += step;
end;
```

Если линия утолщённая, то чертим её ещё раз - со смещением вправо:

```
   // толстая линия:
   if (thick) then
   begin
      t := 0.0;
      while t < PI*2 do
      begin
         var x := Cos(A * t) + Cos(B * t) / 2 + Sin(C * t) / 3;
         var y := Sin(A * t) + Sin(B * t) / 2 + Cos(C * t) / 3;
         var v := new Vertex(new Vector2f(x * k + WIDTH / 2 + 1,
```

```
                                                    y * k + HEIGHT / 2 + 1),
                                                    clr);
            va.Append(v);
            t += step;
        end;
    end;
end;
```

В главном блоке создаём окно приложения, а затем кривую из книги:

```
begin
    // создаём главное окно:
    CreateWindow(WIDTH, HEIGHT, 'Параметрические кривые');

    // создаём кривую:
    A := 1;
    B := 6;
    C := 14;
    CreateCurve();
```

В игровом цикле чертим исходную кривую и печатаем пояснительную надпись:

```
    // игровой цикл:
    while (wind.IsOpen) do
    begin
        // вызываем все обработчики событий:
        wind.DispatchEvents();

        // очищаем экран:
        wind.Clear(BACKGROUND);

        // чертим кривую на экране:
        wind.Draw(va);

        // печатаем надпись:
        wind.Draw(txt);

        // показываем на экране:
        wind.Display();
    end;
```

```
end.
```

Результат работы программы вы уже видели в начале проекта. Но давайте поиграем параметрами!

Колёсико мышки можно крутить без нажатой кнопки. При этом изменяется значение параметра C:

```
// КРУТИМ КОЛЁСИКО МЫШКИ
procedure Window_MouseWheelMoved(sender: Object; e: MouseWheelEven-
tArgs);
begin
   if (e.Delta > 0) then
         C -= 1
     else
         C += 1;

   wind.SetTitle('A = ' + A + '   B = ' + B + '   C = ' + C);
   CreateCurve();
end;
```

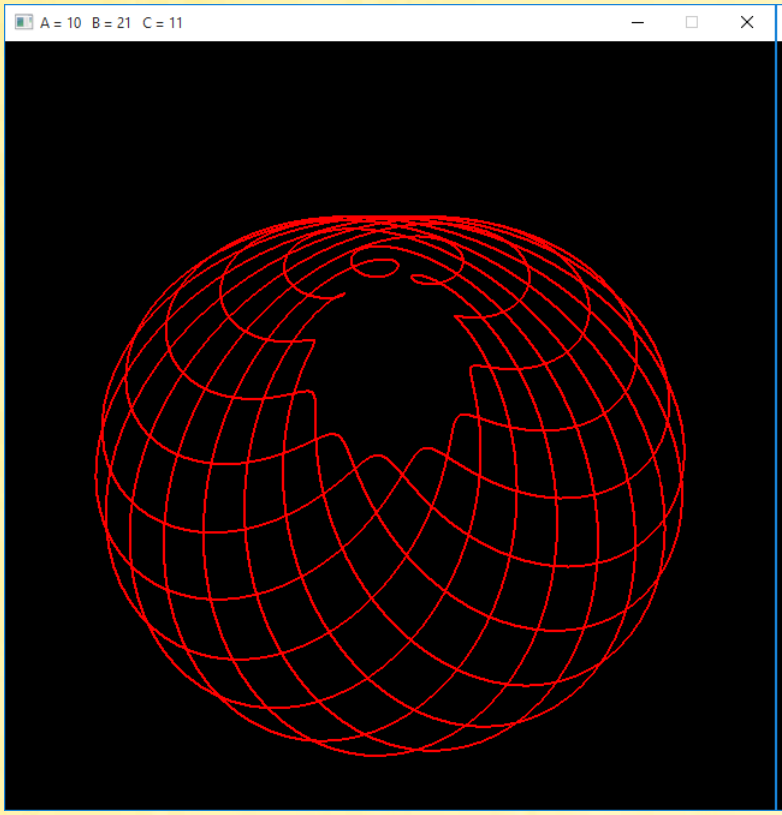
А вот передвигать мышку нужно с нажатой кнопкой:

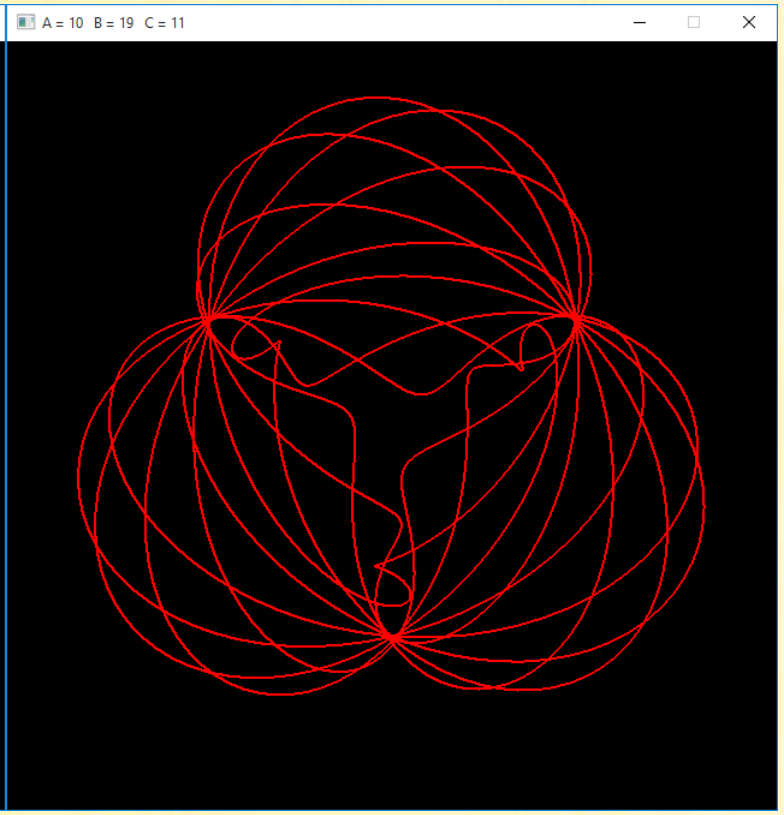
```
// НАЖИМАЕМ КНОПКУ МЫШКИ
procedure Window_MouseButtonPressed(sender: Object; e: MouseButtonE-
ventArgs);
begin
   // запоминаем координаты мышки:
   var mouseX := e.X;
   var mouseY := e.Y;
   mousePos := new Vector2i(mouseX, mouseY);
end;
```

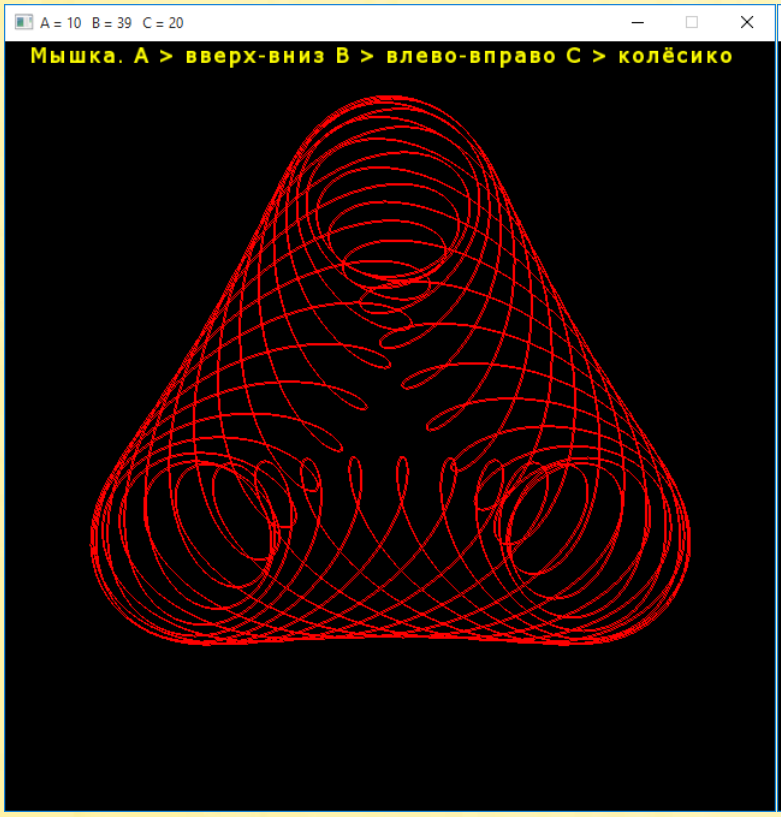
При движении мышки по вертикали изменяется значение параметра A, а при движении по горизонтали – параметра B:

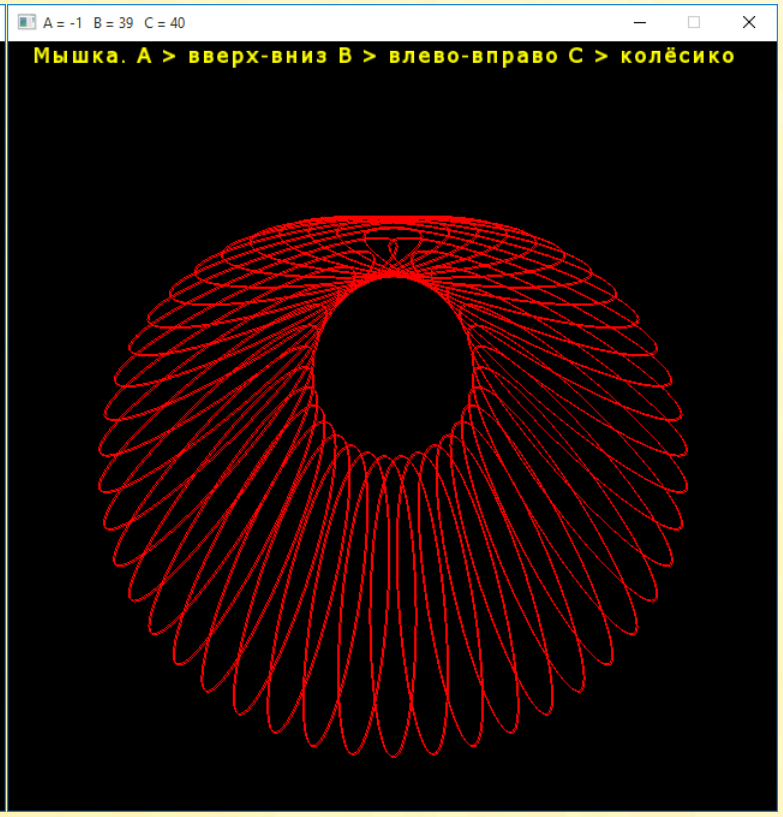
```
// ПЕРЕМЕЩАЕМ МЫШКУ
procedure Window_MouseMoved(sender: Object; e: MouseMoveEventArgs);
begin
   if (not Mouse.IsButtonPressed(Mouse.Button.Left)) then
       exit;

   // текущие координаты мышки:
   var x := e.X;
   var y := e.Y;

   // вверх-вниз:
   var dy := y - mousePos.Y;
   if (dy < -3) then
   begin
      A += 1;
      mousePos := new Vector2i(x, y);
   end
   else if (dy > 3) then
   begin
      A -= 1;
      mousePos := new Vector2i(x, y);
   end;

   // влево-вправо:
   var dx := x - mousePos.X;
   if (dx < -3) then
   begin
      B -= 1;
      mousePos := new Vector2i(x, y);
   end
   else if (dx > 3) then
   begin
      B += 1;
      mousePos := new Vector2i(x, y);
   end;
   // печатаем текущие значения параметров кривой в заголовке окна:
   wind.SetTitle('A = ' + A + '   B = ' + B + '   C = ' + C);
   // создаём кривую с новыми параметрами:
   CreateCurve();
end;
```

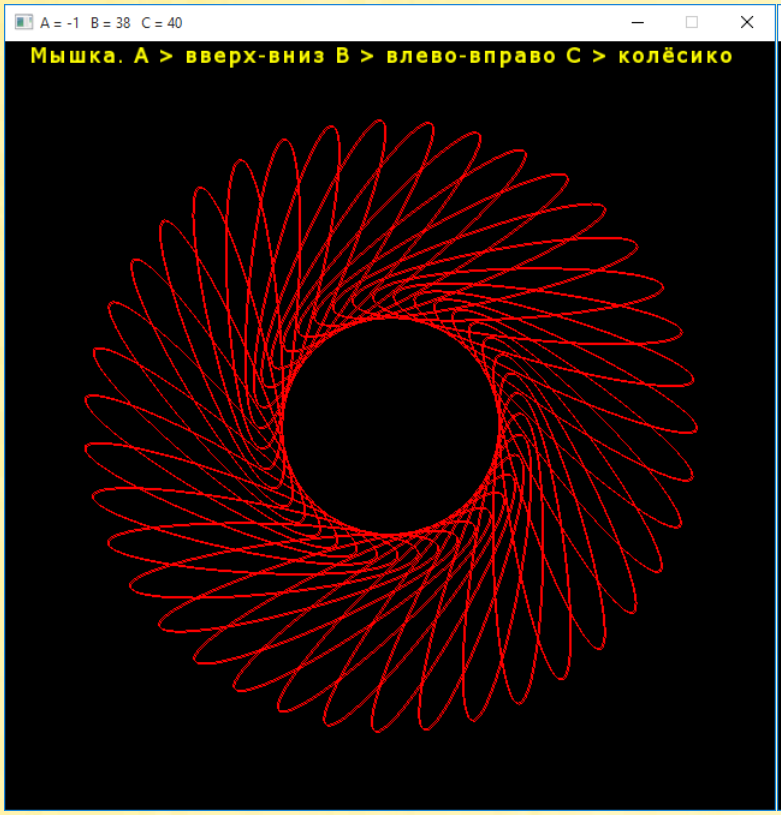
Пришло время натворить кривых линий! Занятие это увлекательное и полезное:

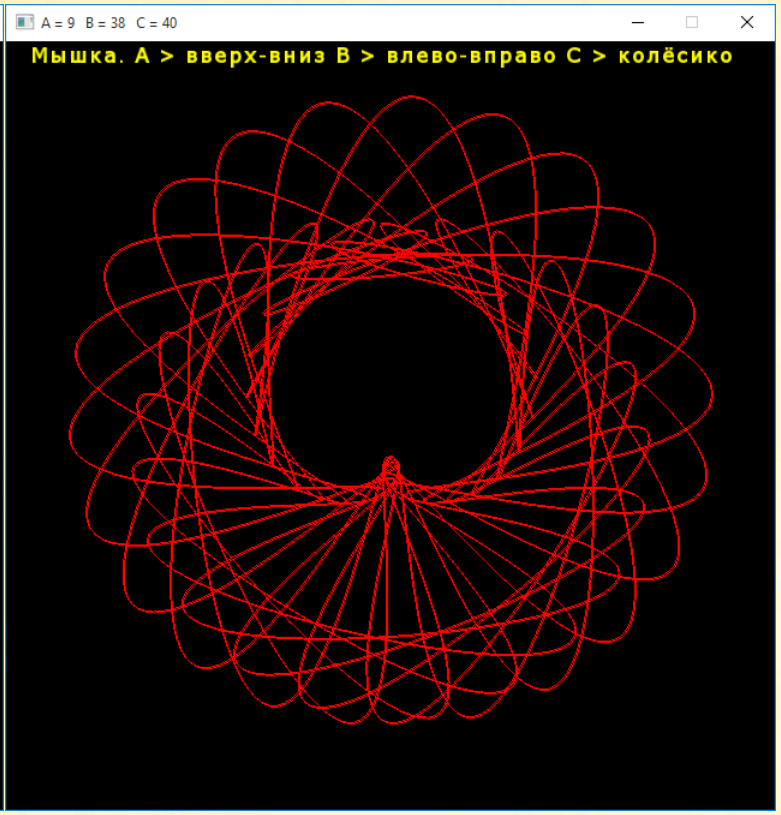

A = 10 B = 21 C = 11

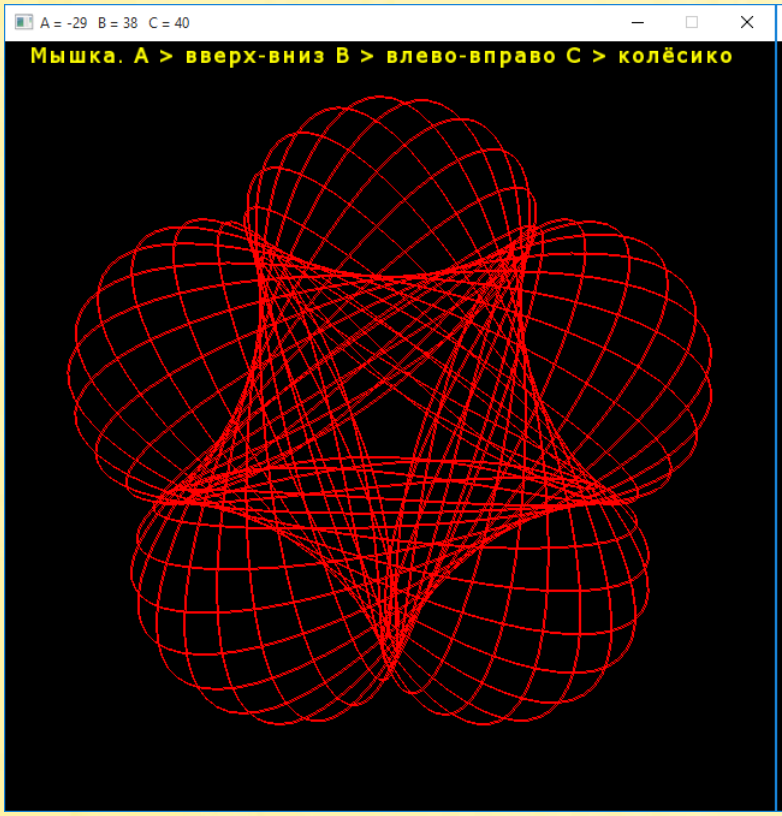

A = 10 B = 19 C = 11

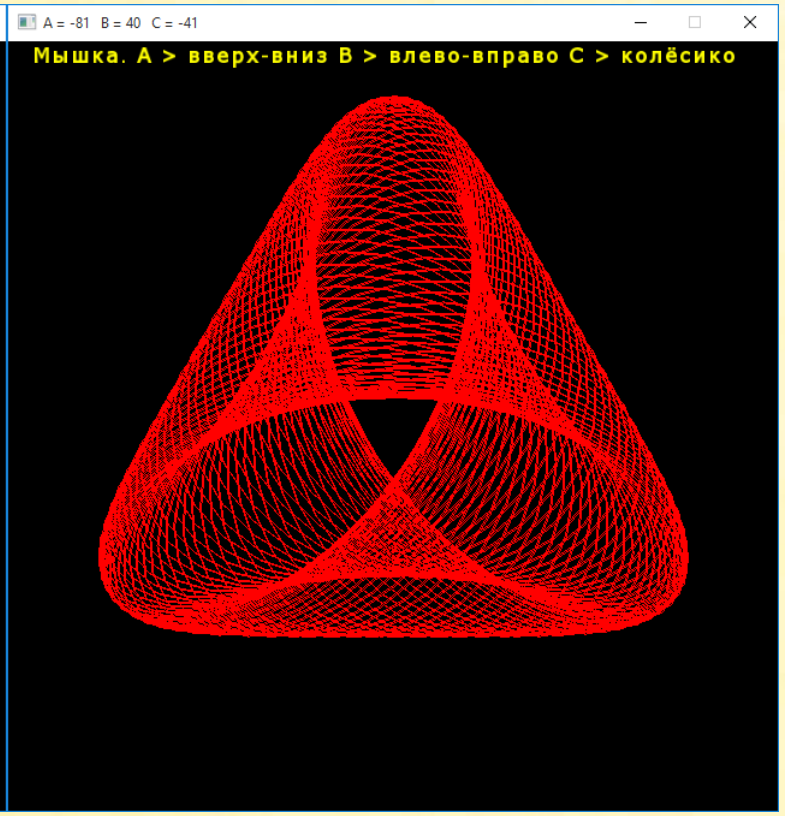

A = 10 B = 39 C = 20
Мышка. A > вверх-вниз B > влево-вправо C > колёсико

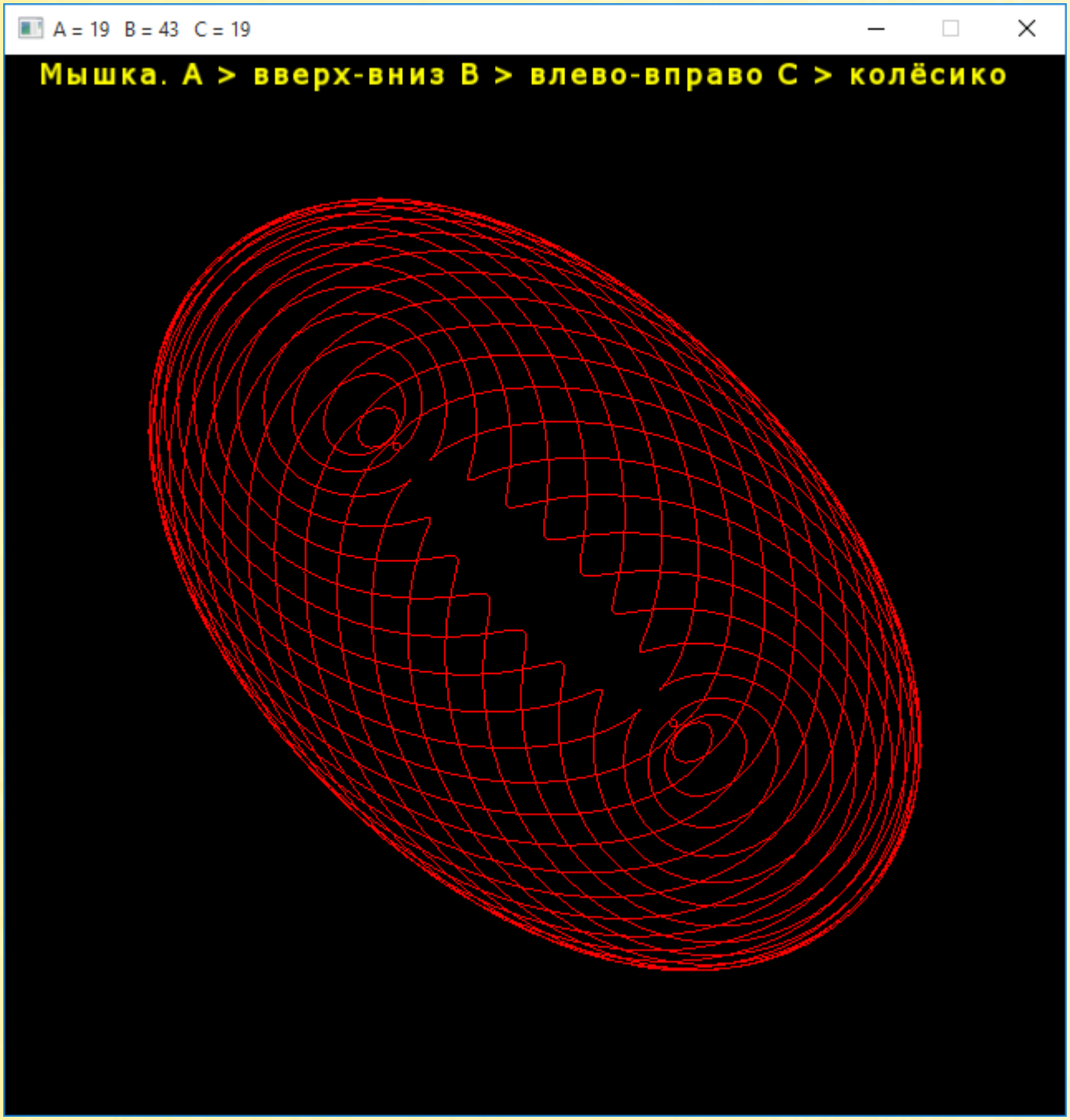

A = -1 B = 39 C = 40
Мышка. A > вверх-вниз B > влево-вправо C > колёсико

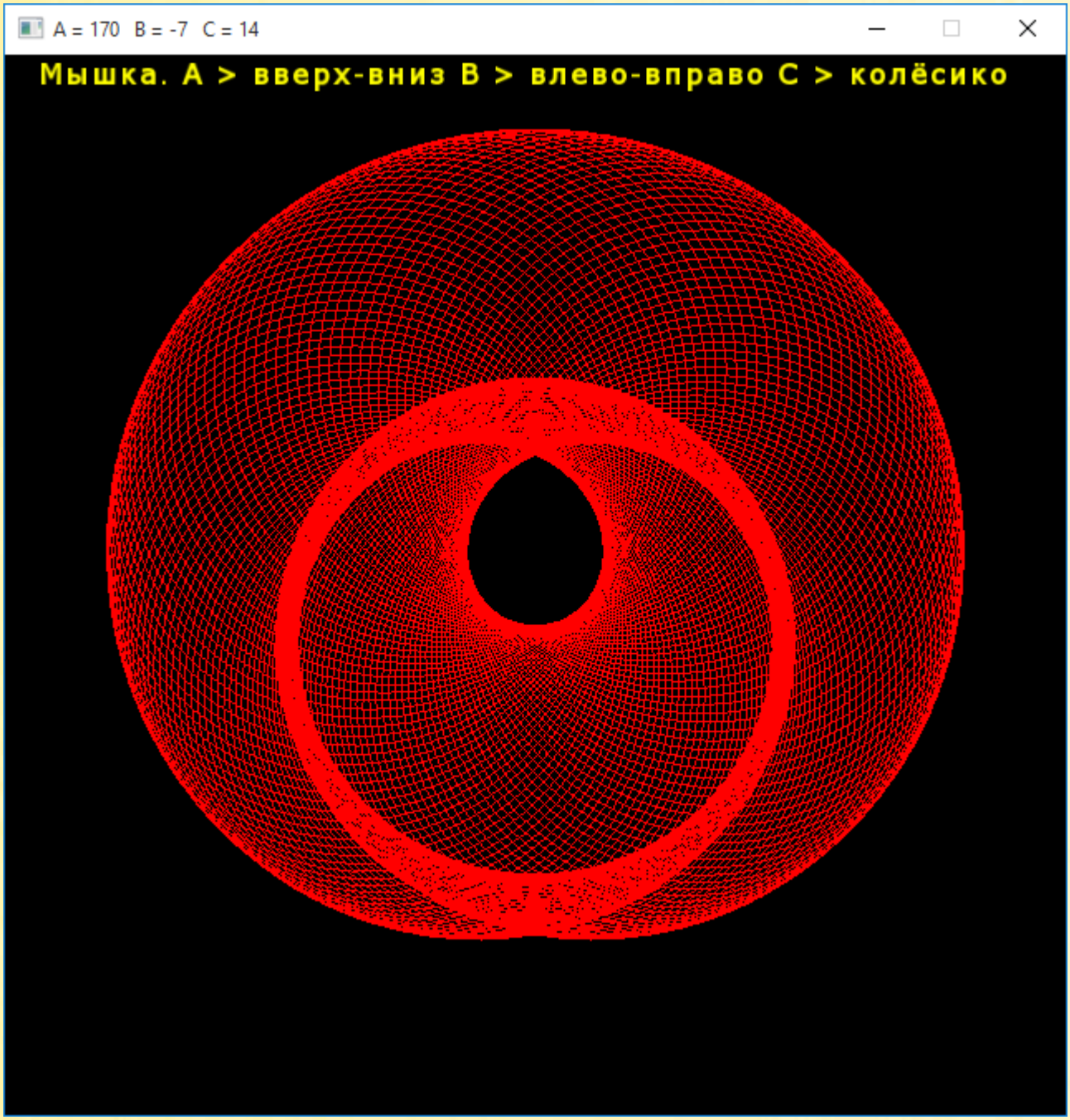

A = -1 B = 38 C = 40
Мышка. A > вверх-вниз B > влево-вправо C > колёсико

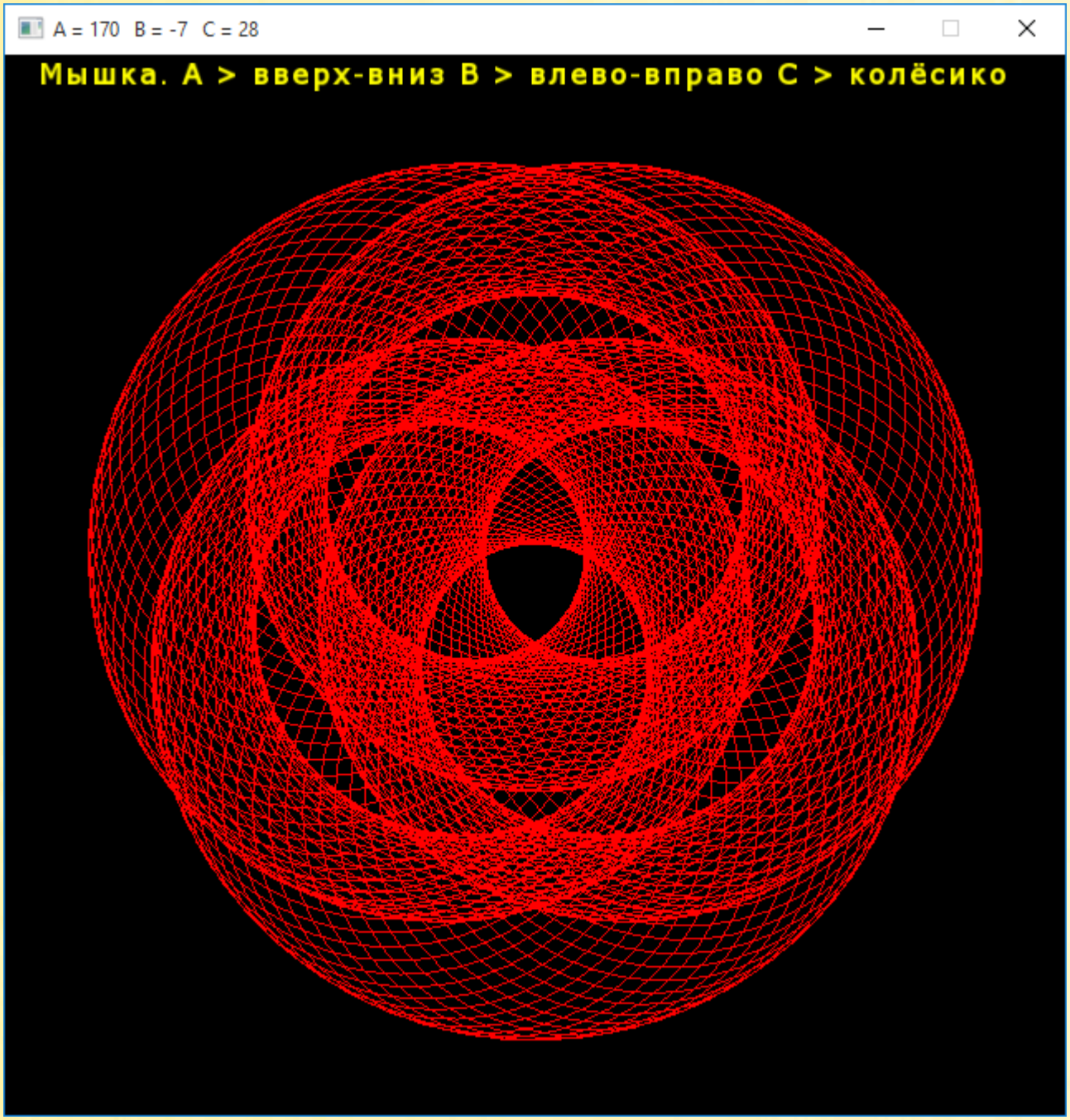

A = 9 B = 38 C = 40
Мышка. A > вверх-вниз B > влево-вправо C > колёсико

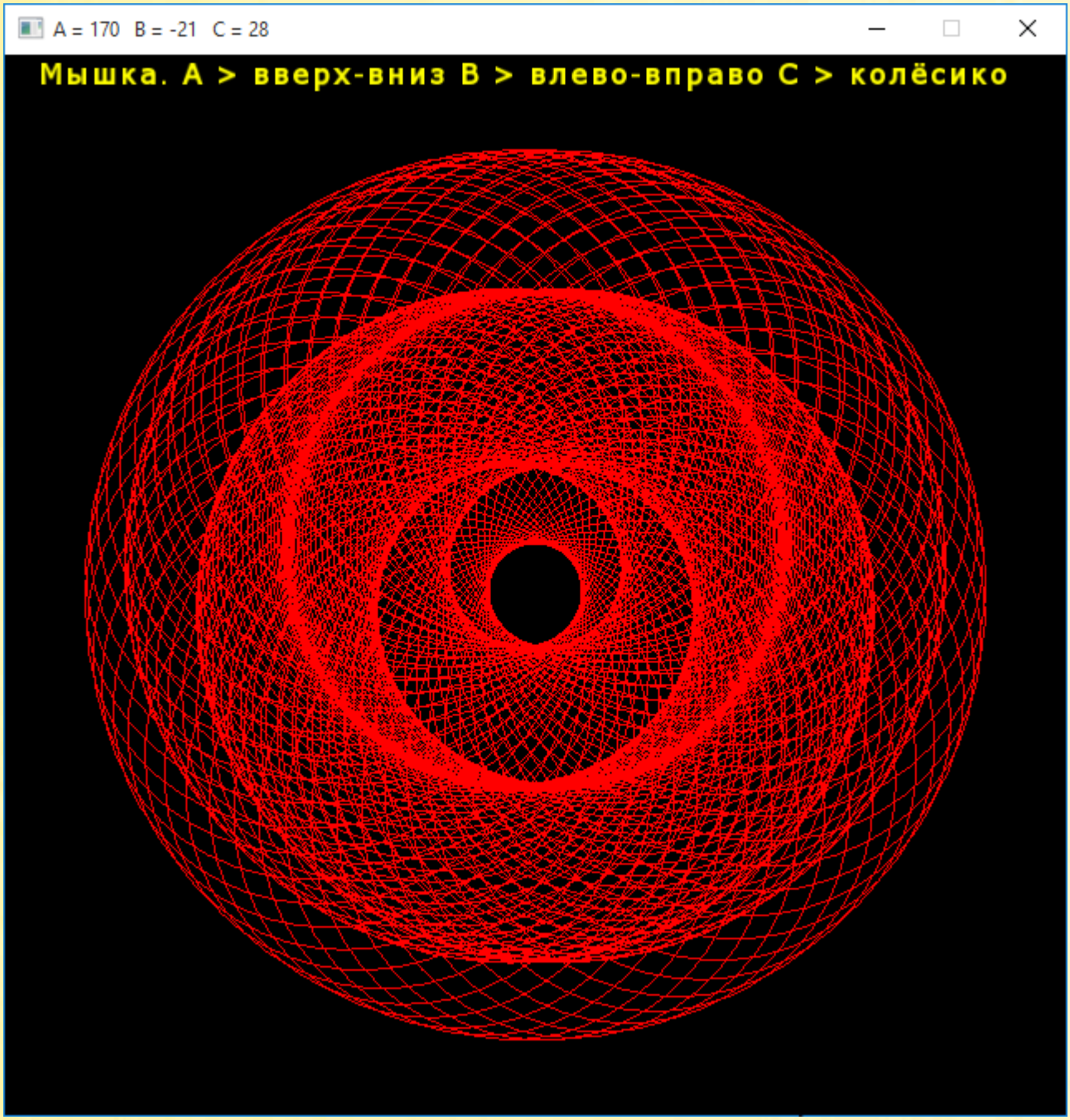

A = -29 B = 38 C = 40
Мышка. A > вверх-вниз B > влево-вправо C > колёсико

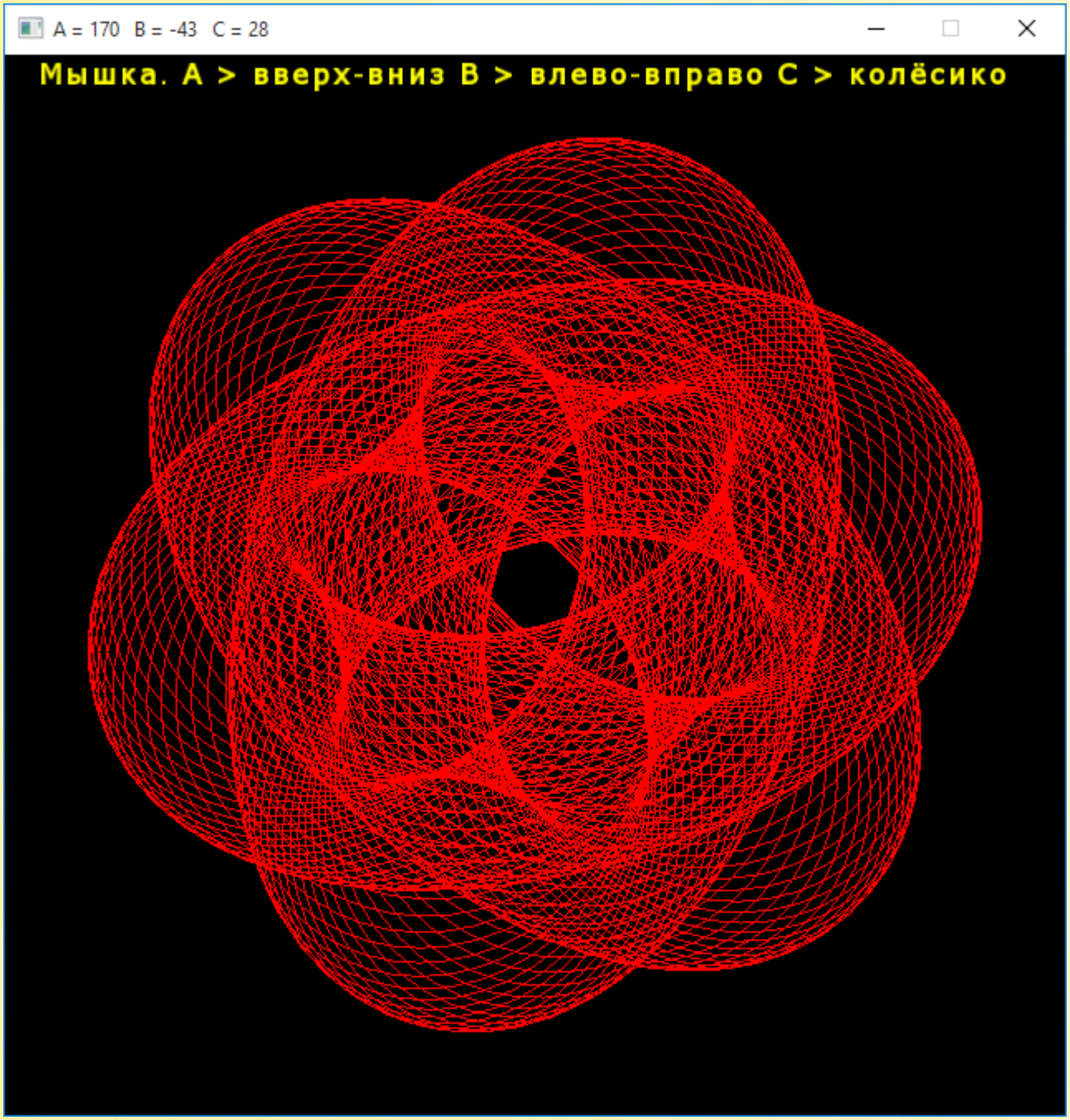

A = -81 B = 40 C = -41
Мышка. A > вверх-вниз B > влево-вправо C > колёсико

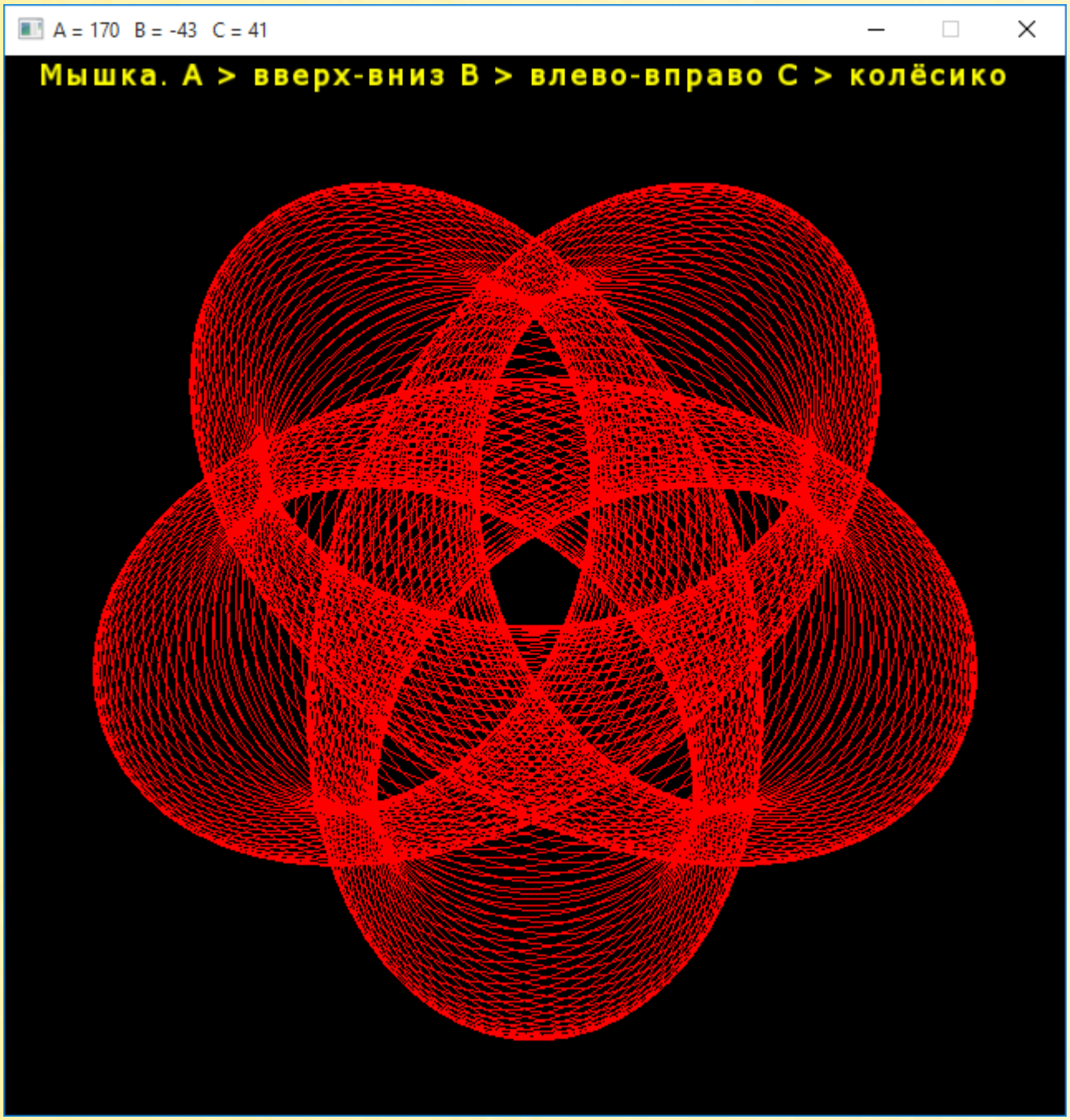

A = 19 B = 43 C = 19
Мышка. A > вверх-вниз B > влево-вправо C > колёсико

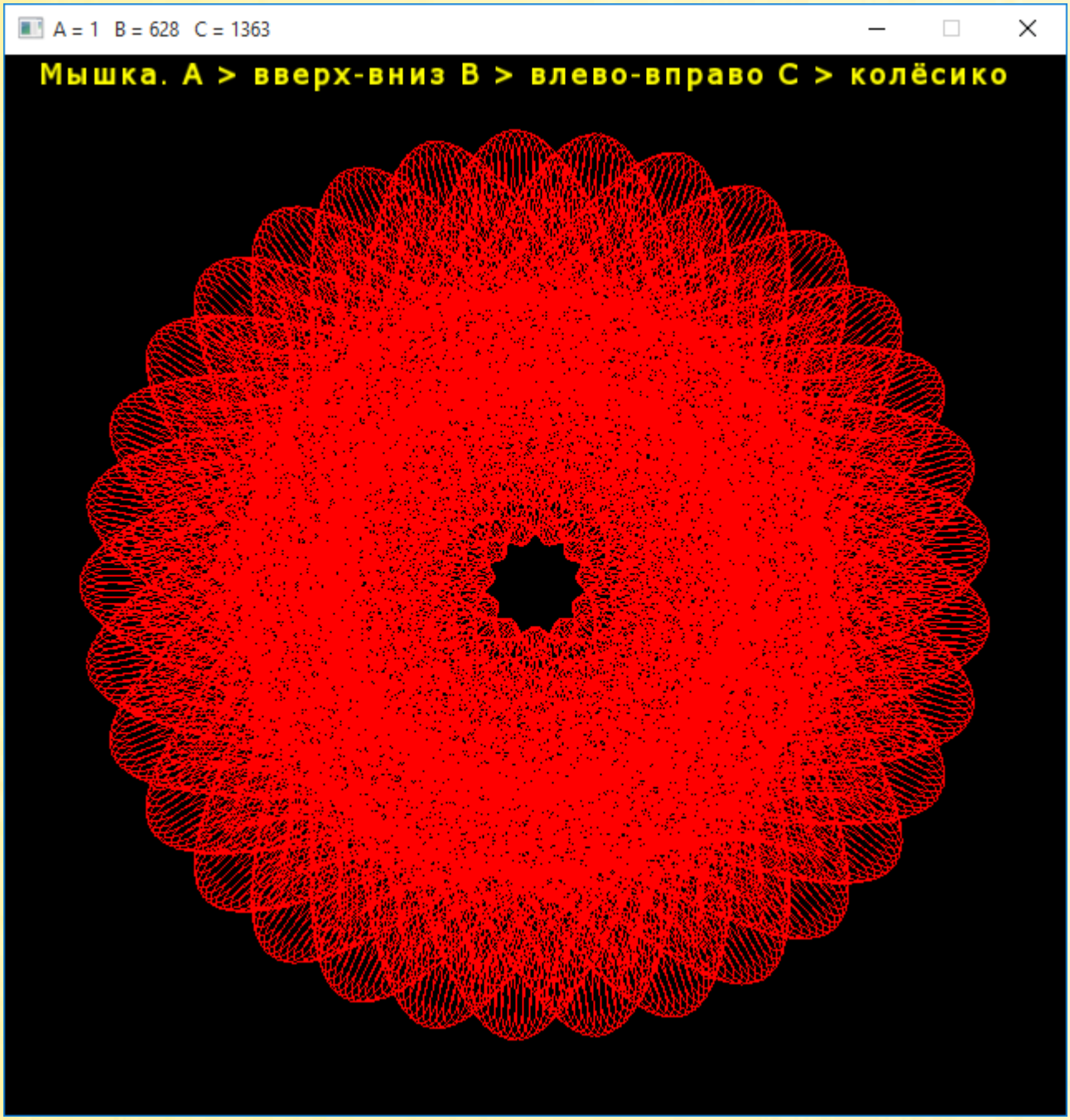

A = 170 B = -7 C = 14
Мышка. A > вверх-вниз B > влево-вправо C > колёсико


A = 170 B = -7 C = 28
Мышка. A > вверх-вниз B > влево-вправо C > колёсико


A = 170 B = -21 C = 28
Мышка. A > вверх-вниз B > влево-вправо C > колёсико


A = 170 B = -43 C = 28
Мышка. A > вверх-вниз B > влево-вправо C > колёсико


A = 170 B = -43 C = 41
Мышка. A > вверх-вниз B > влево-вправо C > колёсико


A = 1 B = 628 C = 1363
Мышка. A > вверх-вниз B > влево-вправо C > колёсико

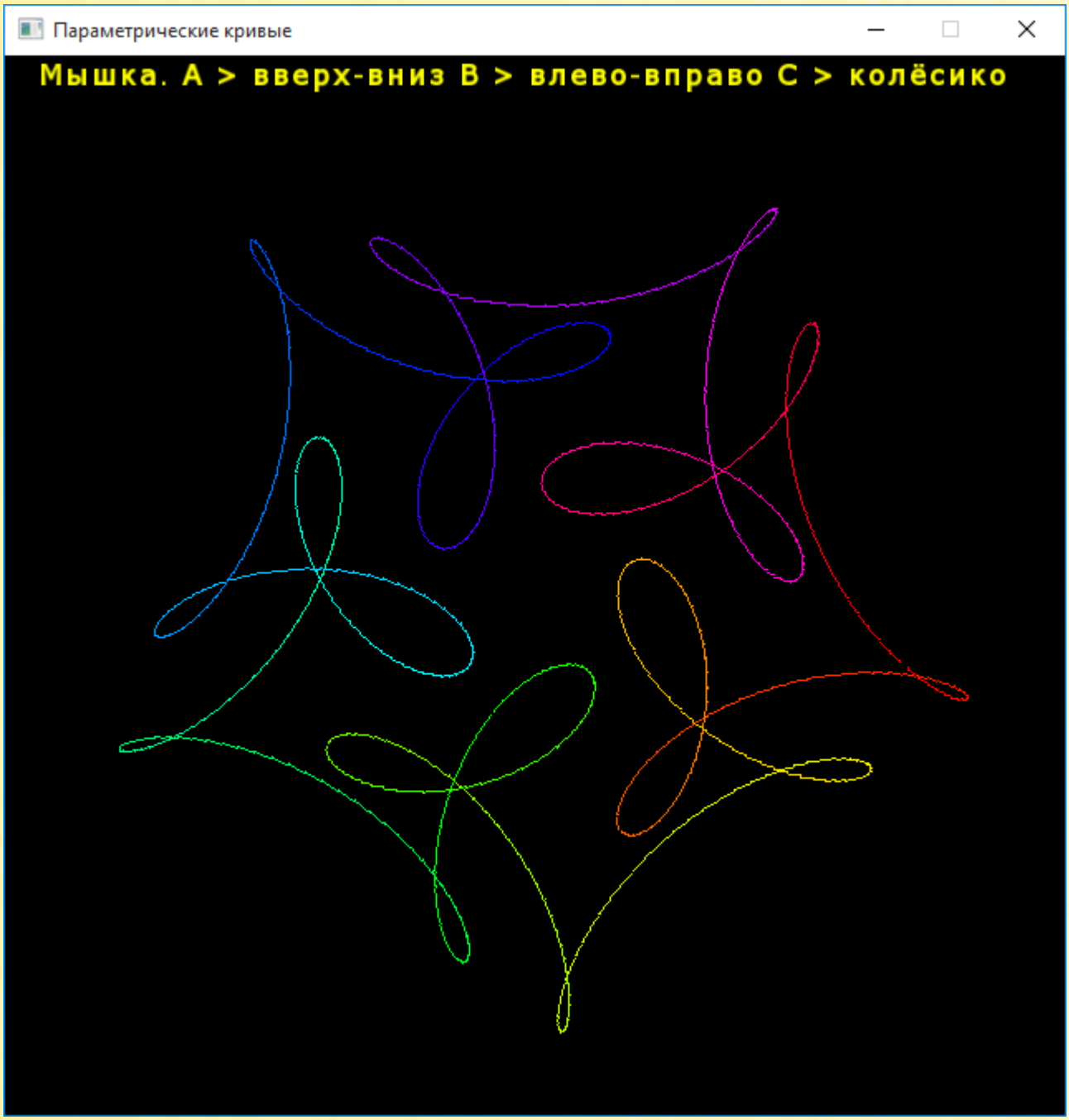
Проект *Параметрические кривые 2*

Мы сделаем параметрические кривые ещё красивее, если раскрасив их в градиентные цвета! Для этого вопользуемся функцией HSV2RGB из библиотеки RVLib.

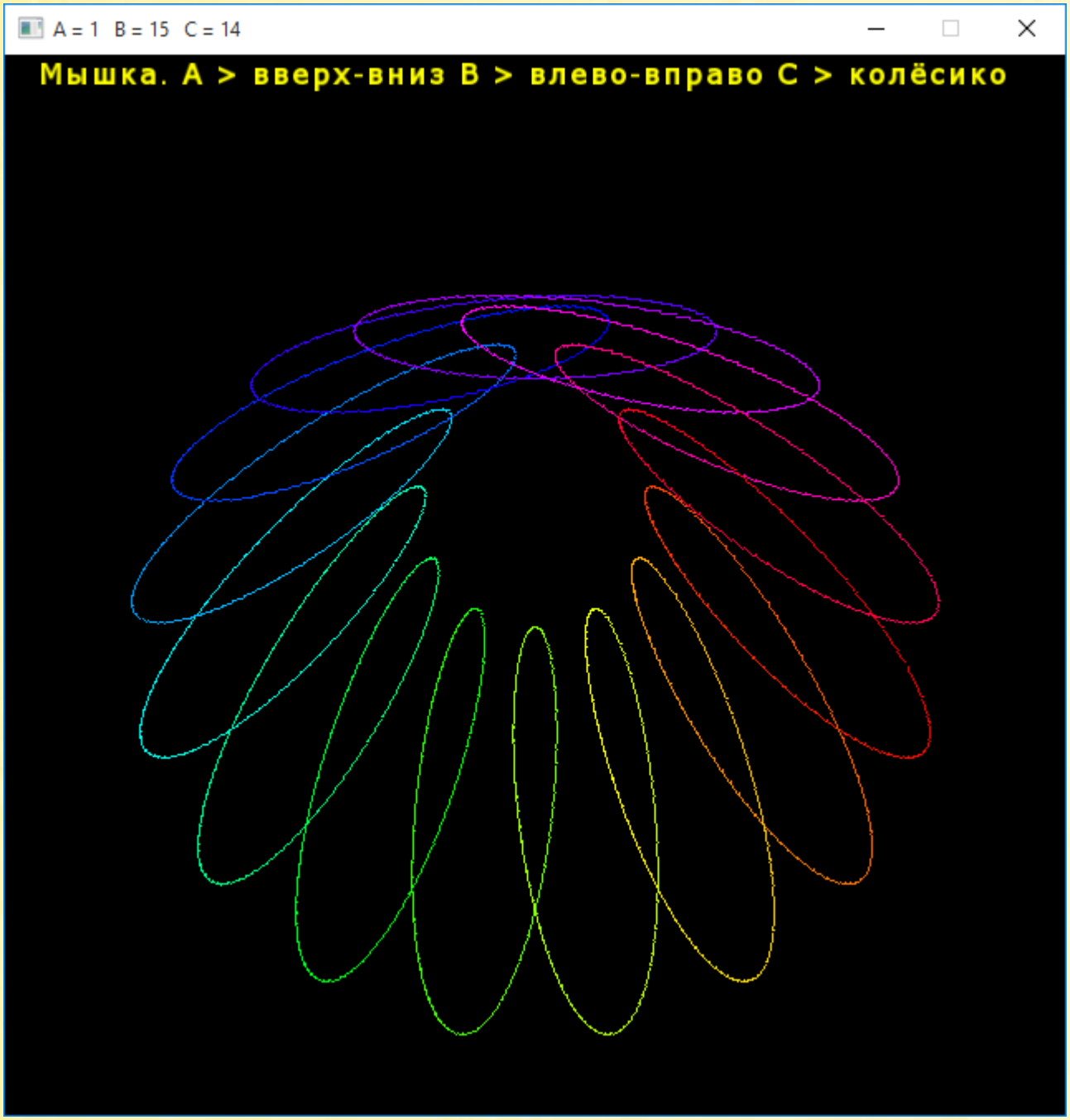
Теперь в процедуре CreateCurve каждая вершина получает свой цвет, который зависит от угла *t*:

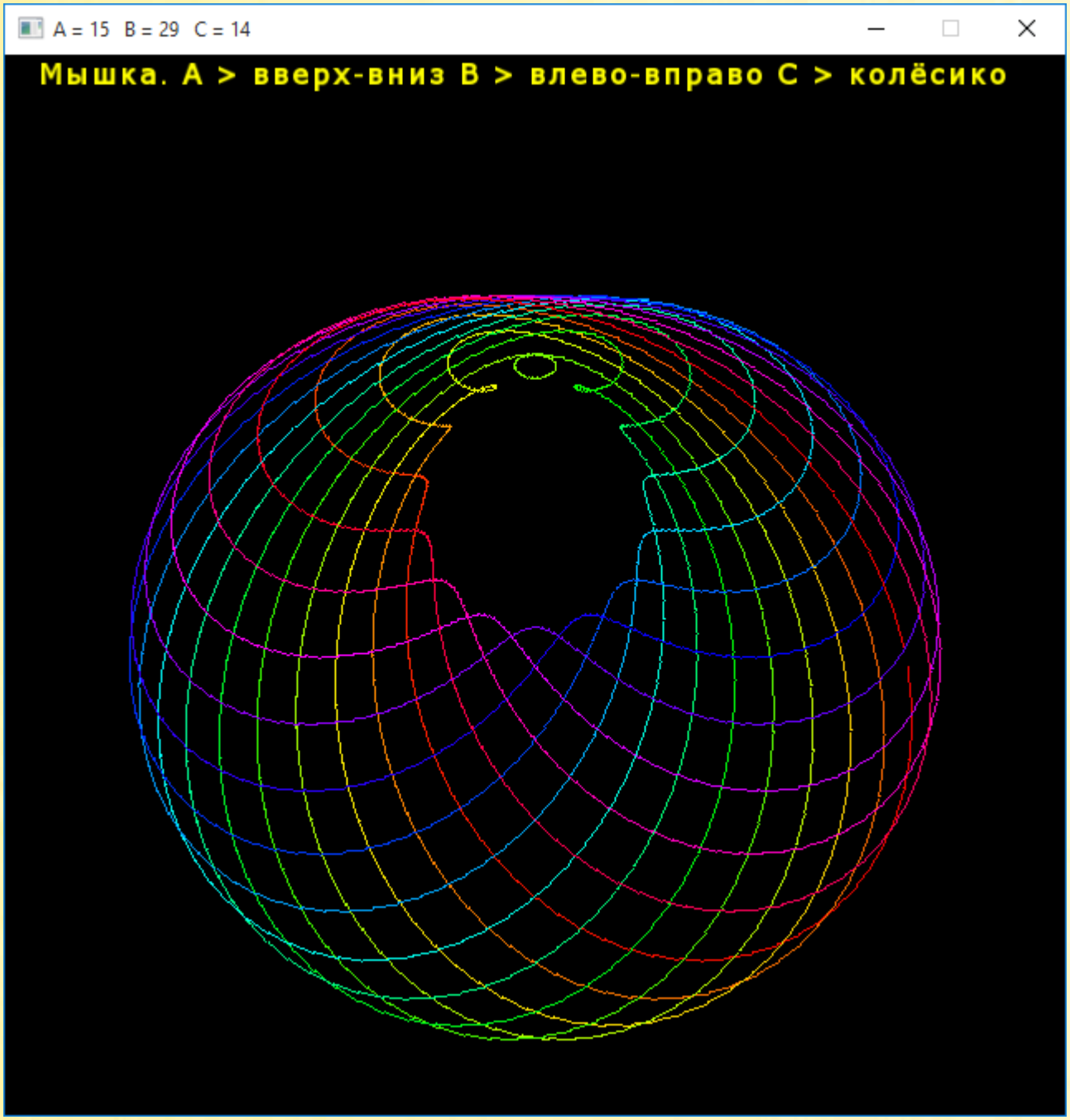
```
// СОЗДАЁМ КРИВУЮ
procedure CreateCurve();
begin

   va := new VertexArray(PrimitiveType.LinesStrip);

   //задаём цвет линии:
   var clr := new Color(255, 0, 0);
   // коэффициент увеличения:
   var k := 150;

   // шаг:
   var maxk := Max(Max(A, B), C);
   var step := Min(1.0/maxk/8, 0.01);

   // добавляем вершины в массив:
   var t := 0.0;
   while t < PI*2 do
   begin
      var x := Cos(A * t) + Cos(B * t) / 2 + Sin(C * t) / 3;
      var y := Sin(A * t) + Sin(B * t) / 2 + Cos(C * t) / 3;
      clr := HSV2RGB(t * 360.0 / PI/ 2);
      var v := new Vertex(new Vector2f(x*k + WIDTH / 2,
                                       y * k + HEIGHT / 2), clr);
      va.Append(v);

      t += step;
   end;
end;
```

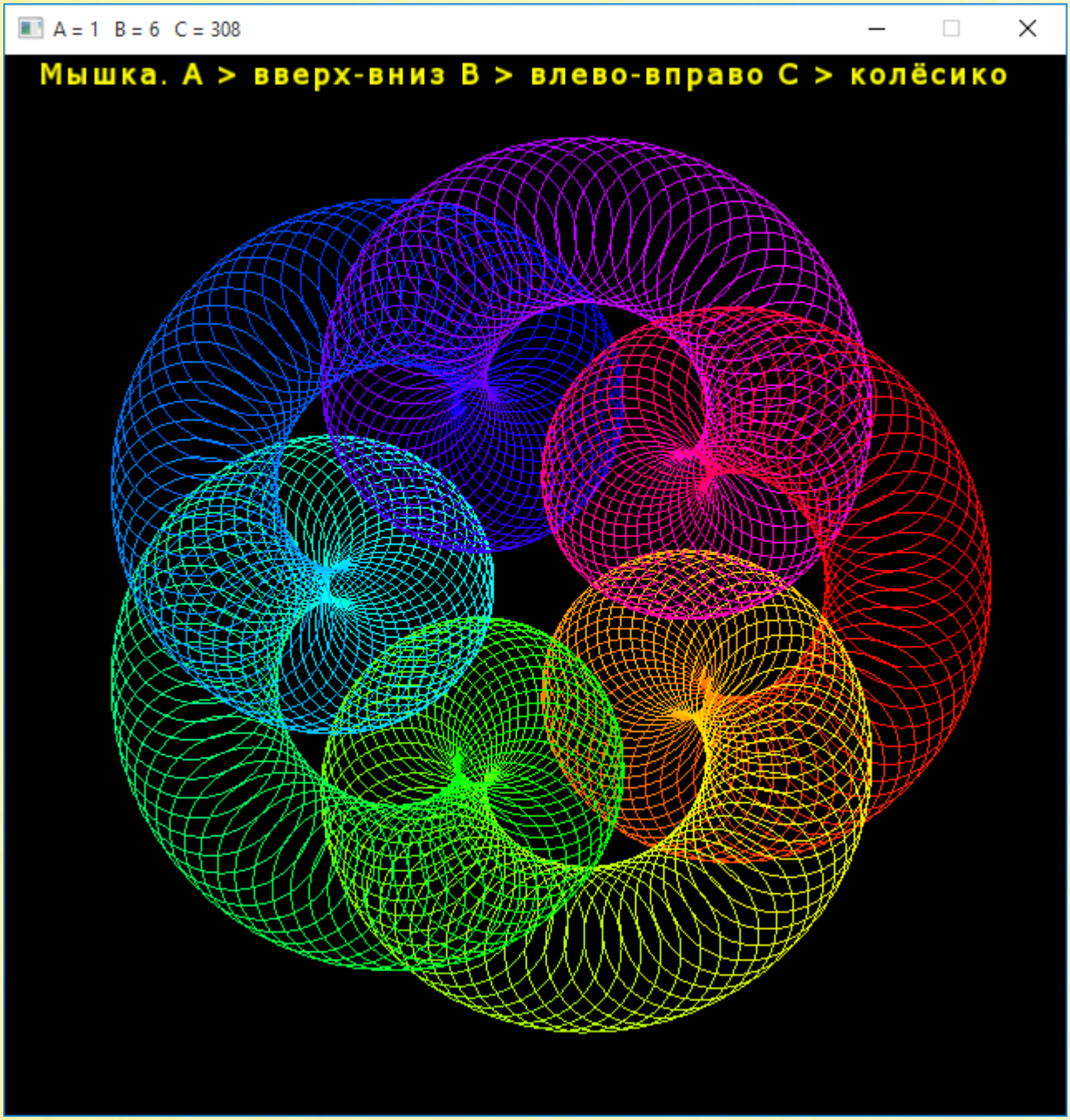
«Книжная» кривая стала разноцветной:



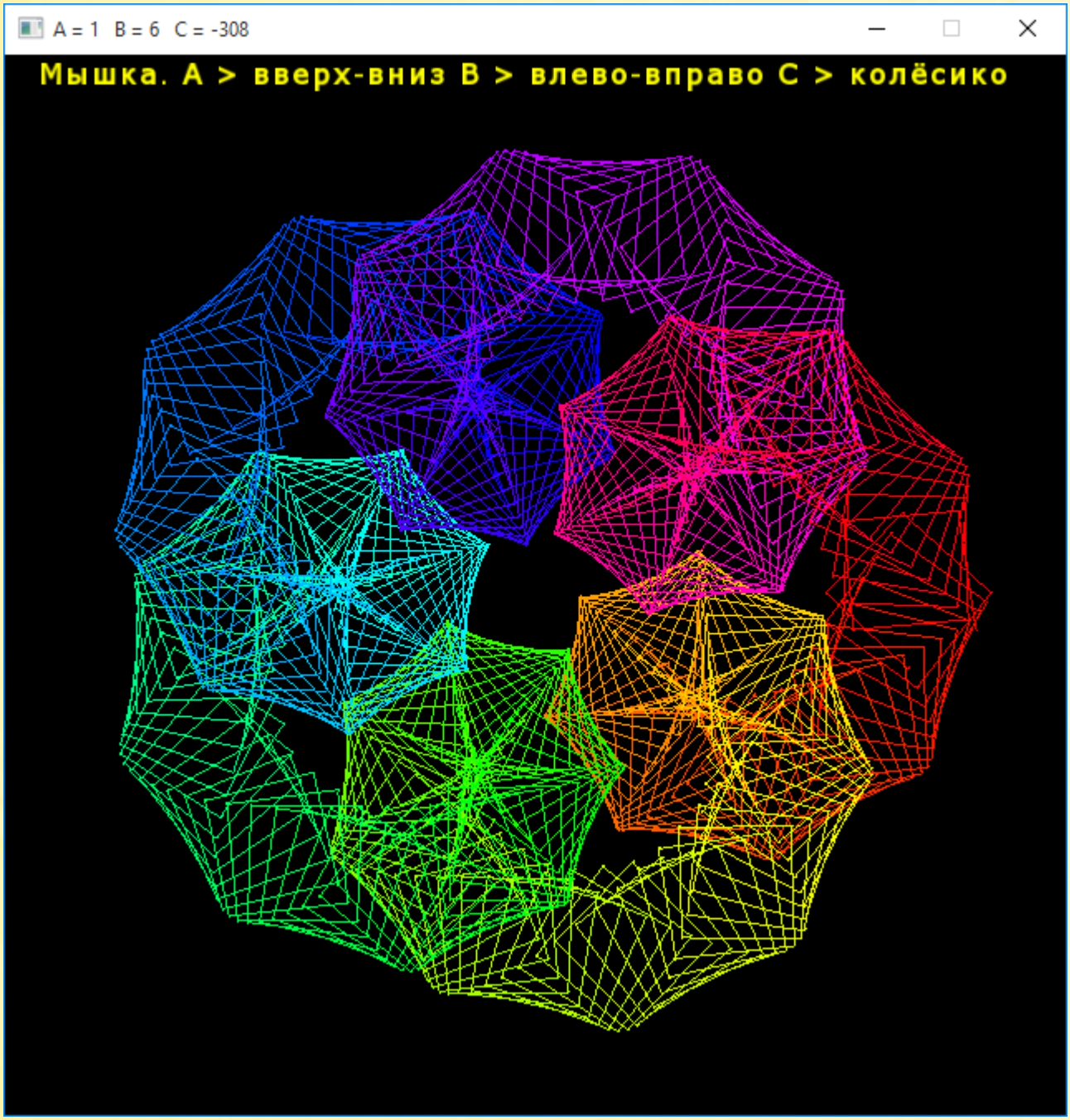
Двигайте мышку в разные стороны, крутите колёсико – и получите замечательные картинки:

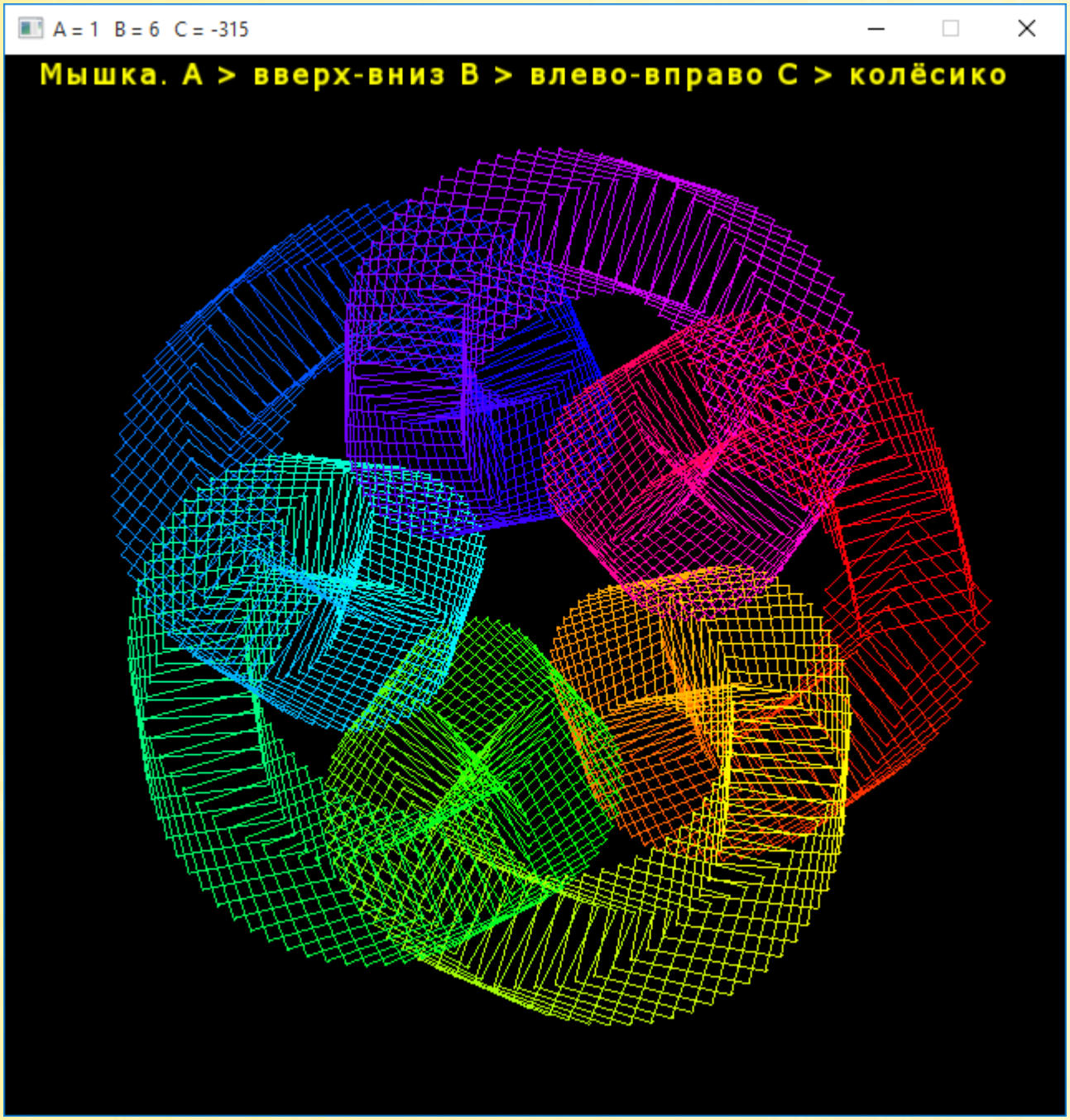

A = 1 B = 15 C = 14
Мышка. A > вверх-вниз B > влево-вправо C > колёсико


A = 15 B = 29 C = 14
Мышка. A > вверх-вниз B > влево-вправо C > колёсико



При **отрицательных** значениях параметров кривой получаются угловатые кривые, которые не менее интересны:


A = 1 B = 6 C = -308
Мышка. A > вверх-вниз B > влево-вправо C > колёсико


A = 1 B = 6 C = -315
Мышка. A > вверх-вниз B > влево-вправо C > колёсико

Проект *Параметрические кривые 3*

Чтобы не крутить колёсико, мы поручим это дело… нет, не мышке, а программе!

Для точного «отмера» времени создадим **часы**, и каждые 100 миллисекунд будем обновлять кривую на экране. В цикле изменяется только значение параметра **C**, а другие параметры можно изменять мышкой.

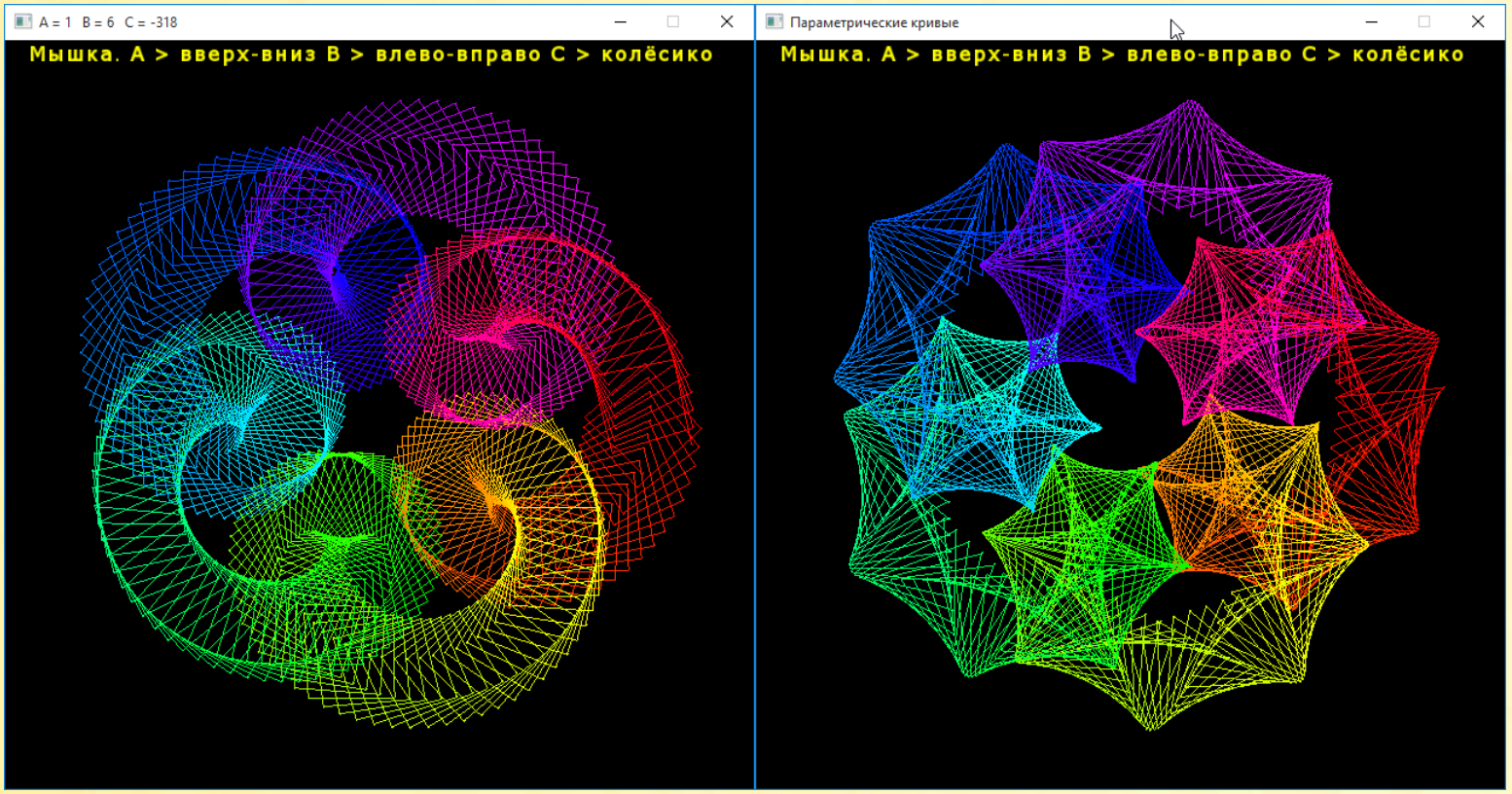
Будьте осторожны с мышкой! При некоторых значениях параметров программа зависает.

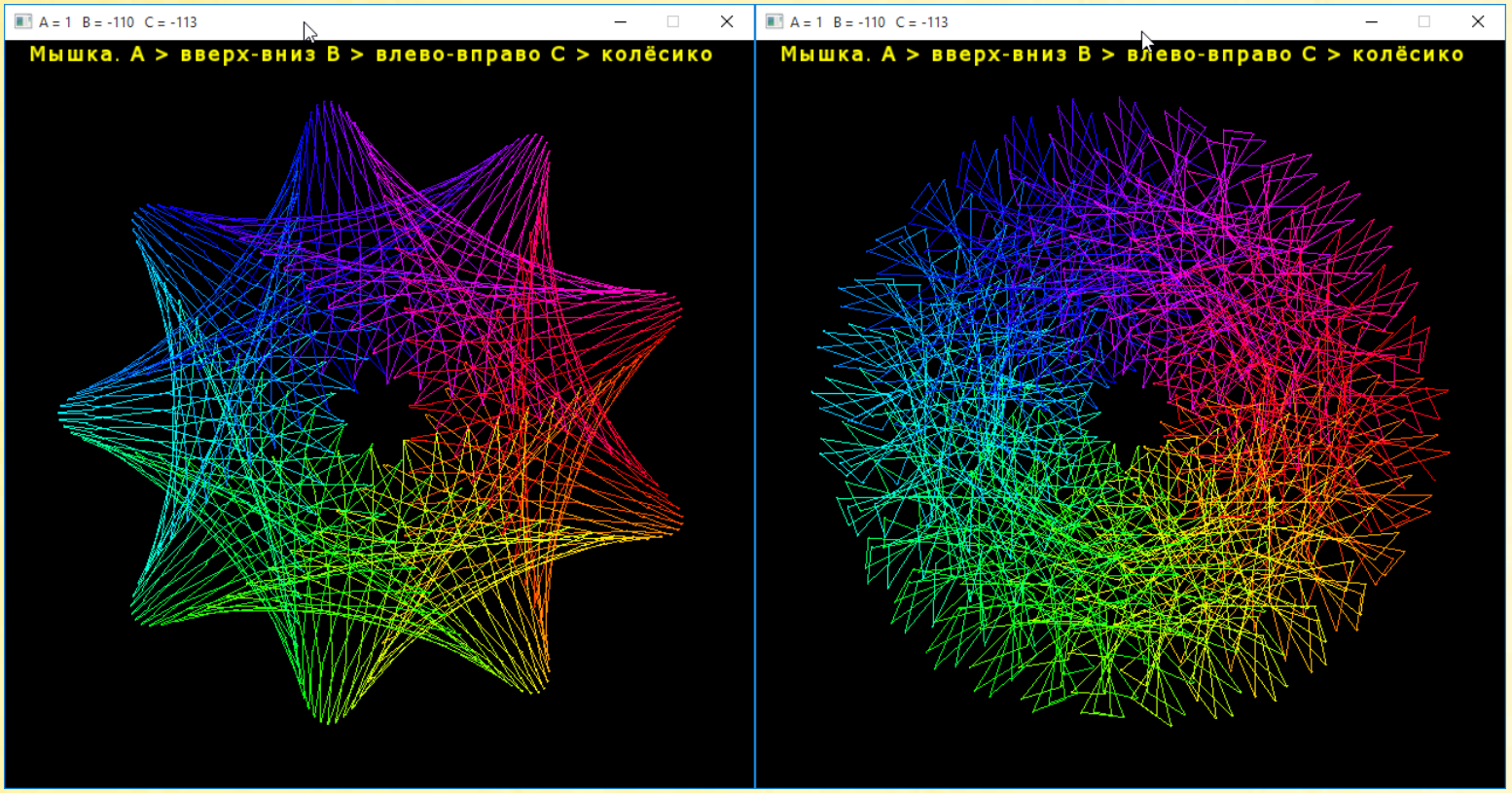
Значения параметров **A** и **B** можно задать в начале главного блока.

```
begin
   // создаём главное окно:
   CreateWindow(WIDTH, HEIGHT, 'Параметрические кривые');

   var &clock := new Clock();

   // создаём кривую:
   A := 1;
   B := 6;
   C := 14;
   CreateCurve();


   // игровой цикл:
   while (wind.IsOpen) do
   begin
      // вызываем все обработчики событий:
      wind.DispatchEvents();

      // очищаем экран:
      wind.Clear(BACKGROUND);

      // следующая кривая:
      var dt := clock.ElapsedTime.AsMilliseconds();
      if (dt > 100) then
      begin
```

```
            C -= 1;
            CreateCurve();
            &clock.Restart();
        end;

        // чертим кривую на экране:
        wind.Draw(va);

        // печатаем надпись:
        wind.Draw(txt);

        // показываем на экране:
        wind.Display();
    end;

end.
```

Дух захватывает от такой красоты. Лепота!





Проект *Параметрические кривые 4*

Пойдём ещё дальше и анимируем построение кривой!

Переменные из процедуры *CreateCurve* сделаем глобальными переменными, чтобы их можно было использовать и в главном блоке:

```
// текущий угол:
t := 0.0;
// шаг:
step := 0.0;
// коэффициент увеличения:
k := 150;
```

Сама процедура *CreateCurve* сократилась до процедуры NewCurve:

```
// СОЗДАЁМ НОВУЮ КРИВУЮ
procedure NewCurve();
begin
   va := new VertexArray(PrimitiveType.LinesStrip);
   t := 0;
   // шаг:
   var maxk := Max(Max(A, B), C);
   step := Min(1.0 / maxk / 8, 0.005);
end;
```

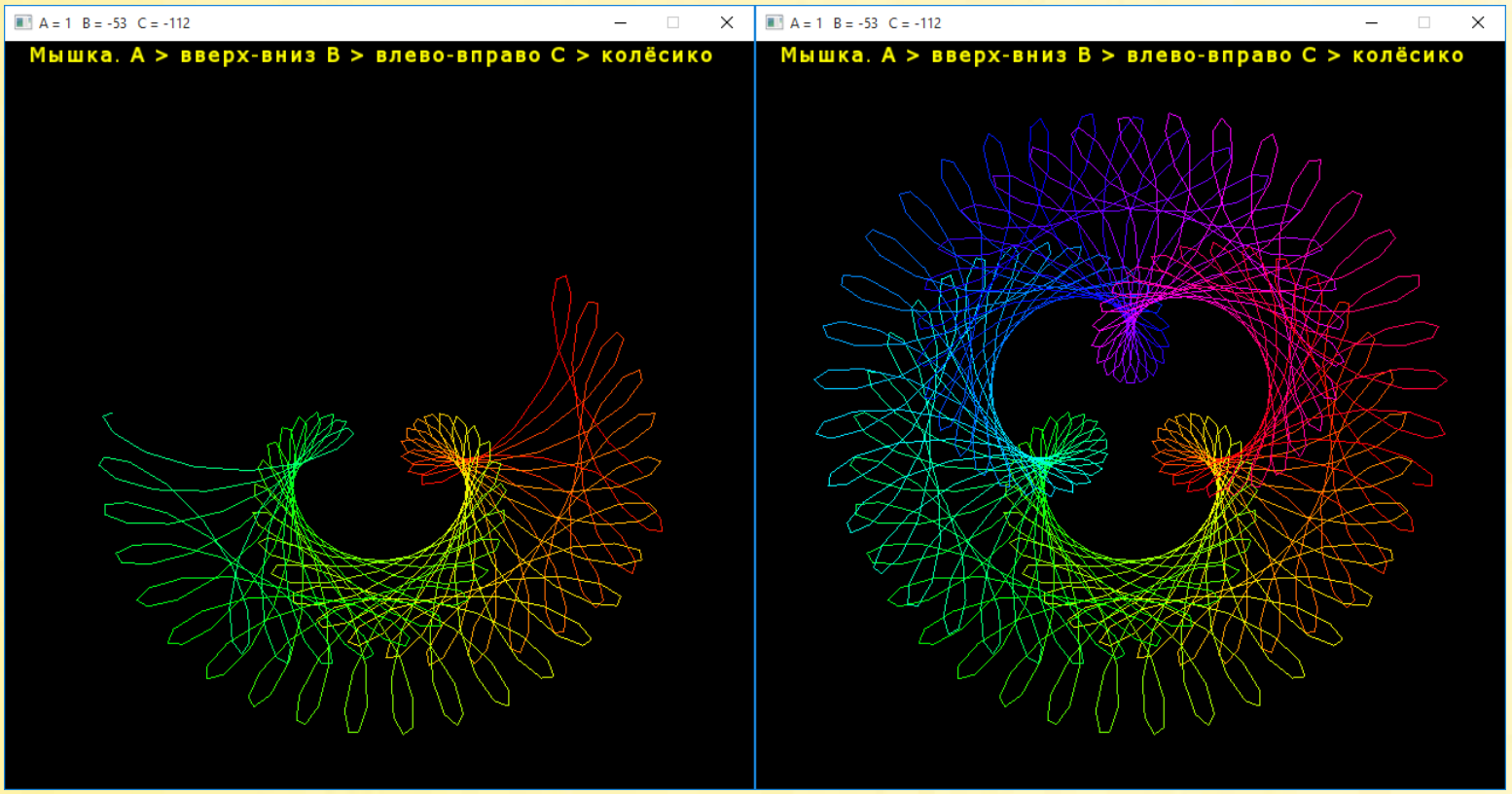
При любых изменениях параметров кривой следует вызывать не процедуру *CreateCurve*, а процедуру *NewCurve*.

Теперь вычисление координат точек кривой производятся в главном блоке:

```
begin
   // создаём главное окно:
   CreateWindow(WIDTH, HEIGHT, 'Параметрические кривые');

   // создаём кривую с параметрами по умолчанию:
   NewCurve();

   // игровой цикл:
   while (wind.IsOpen) do
   begin
      // вызываем все обработчики событий:
      wind.DispatchEvents();

      // кривая построена:
      if (t > PI * 2) then
          continue;

      // очищаем экран:
      wind.Clear(BACKGROUND);

      // вычисляем координаты следующей точки кривой:
      var x := Cos(A * t) + Cos(B * t) / 2 + Sin(C * t) / 3;
      var y := Sin(A * t) + Sin(B * t) / 2 + Cos(C * t) / 3;
      // её цвет:
      var clr := HSV2RGB(t * 360.0 / PI / 2);
      var v := new Vertex(new Vector2f(x * k + WIDTH / 2,
                                       y * k + HEIGHT / 2), clr);
```

```
        // добавляем точку в массив:
        va.Append(v);

        // чертим кривую на экране:
        wind.Draw(va);

        // следующий угол:
        t += step;

        // печатаем надпись:
        wind.Draw(txt);

        // показываем на экране:
        wind.Display();
    end;
end.
```

Параметры кривой можно изменять на лету с помощью мышки, как и раньше:



Смотреть графические мультики очень интересно! Завораживающее зрелище!

Проект *Треугольное кольцо*

Мы ещё не охватили своим вниманием вершины типа `PrimitiveType.TrianglesStrip`. В этом проекте мы их охватим.

Нам понадобятся 2 **глобальные переменные**:

```
// массив вершин:
va: VertexArray := nil;
// число вершин:
numVertex := 18;
```

Крутя колёсико в обе стороны, мы изменяем **число вершин** в разумных пределах:

```
// КРУТИМ КОЛЁСИКО МЫШКИ
procedure Window_MouseWheelMoved(sender: Object; e: MouseWheelEventArgs);
begin
   if (e.Delta > 0) then
       numVertex -= 1
   else
       numVertex += 1;

   if (numVertex < 3) or (numVertex > 18) then
       numVertex := 18;

   wind.SetTitle('numPoints = ' + numVertex);
end;
```

В процедуре **Draw** вычисляем координаты вершин треугольников и помещаем их в массив *VertexArray*:

```
procedure Draw();
begin
   // размеры кольца:
   var maxRadius := 180;
   var minRadius := 60;
   // центр кольца:
```

```
    var xc := WIDTH / 2;
    var yc := HEIGHT / 2;
    // начальный угол:
    var angle := 0.0;
    // приращение угла:
    var angleStep := PI / numVertex;

    // создаём массив вершин:
    va := new VertexArray(PrimitiveType.TrianglesStrip);
    for var i := 0 to numVertex do
    begin
        // цвет вершины:
        var clr := HSV2RGB(angle * 360.0 / PI / 2);
        // координаты:
        var px := xc + Cos(angle) * maxRadius;
        var py := yc + Sin(angle) * maxRadius;
        // добавляем вершину в массив:
        var v := new Vertex(new Vector2f(px, py), clr);
        va.Append(v);
        angle += angleStep;
        px := xc + Cos(angle) * minRadius;
        py := yc + Sin(angle) * minRadius;
        v := new Vertex(new Vector2f(px, py), clr);
        va.Append(v);
        angle += angleStep;
    end;
```

После чего печатаем на экране:

```
    // чертим треугольники:
    wind.Draw(va);
end;
```

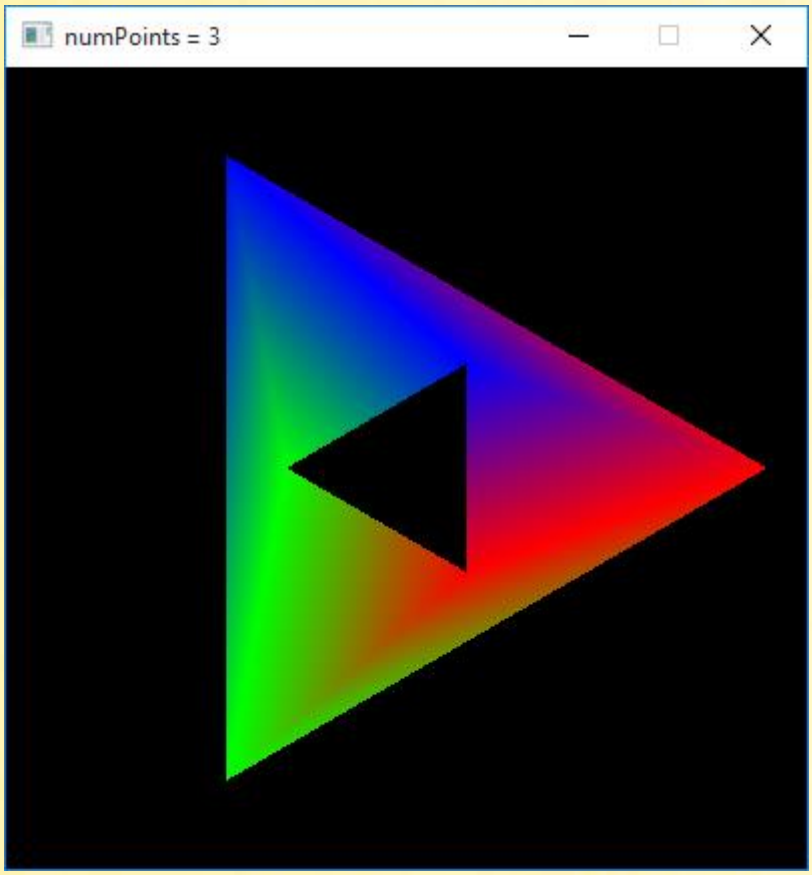
В главном блоке обновляем картинку:

```
begin
    // создаём главное окно:
    CreateWindow(WIDTH, HEIGHT, 'Треугольное кольцо');
    // цвет фона окна:
    wind.Clear(BACKCOLOR);
```

```
    // игровой цикл:
    while (wind.IsOpen) do
    begin
      // вызываем все обработчики событий:
      wind.DispatchEvents();

      // очищаем экран:
      wind.Clear(BACKCOLOR);

      // чертим треугольники:
      Draw();

      // показываем на экране:
      wind.Display();
    end;

end.
```

Так как у треугольников нет контуров, то вы увидите вот такой цветной бублик:

Самый маленький «бублик» - треугольный, но тоже красивый:



Проект *Треугольное кольцо 2*

Увеличим размеры окна до максимума:

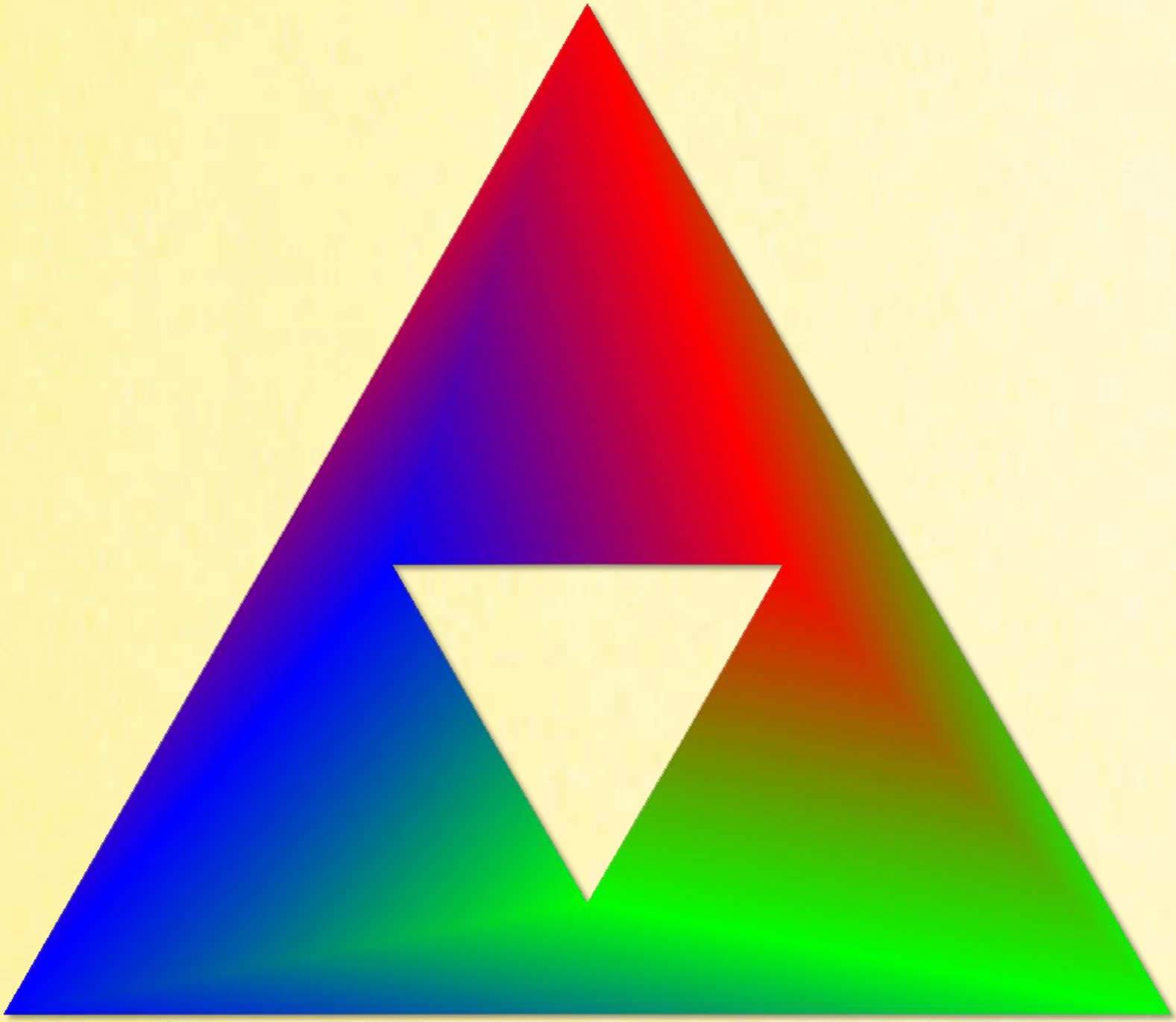
```
const
   // размеры окна:
   WIDTH = 1400;
   HEIGHT = 1100;
```

И аналогично – размеры кольца:

```
procedure Draw();
begin
   // размеры кольца:
```

```
var maxRadius := 600;//180;
var minRadius := 200;//60;
```

Делаем треугольное кольцо, сохраняем картинку на диске. Поворачиваем и убираем чёрный фон в *Фотошопе* – получилась замечательная картинка, сделанная своими руками:



Проект *Треугольное кольцо 3*

Лучше всего у нас получился треугольник с дыркой, так почему бы не сделать **размеры дырки** управляемыми?

Число вершин всегда равно трём:

```
const
   // число вершин:
   numVertex = 3;
```

Радиус внешней окружности переведём в глобальную переменную:

```
// мин. размеры кольца:
minRadius := 200;
```

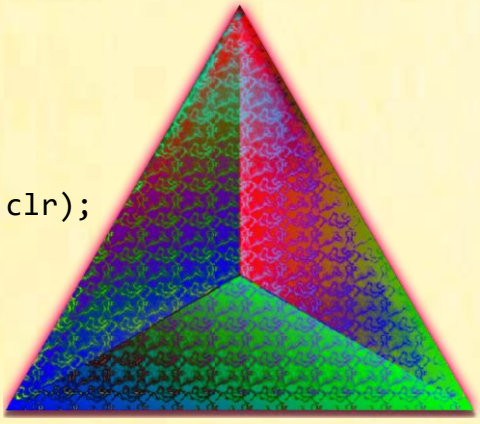
При кручении колёсика мышки этот радиус изменяется в обе стороны:

```
// КРУТИМ КОЛЁСИКО МЫШКИ
procedure Window_MouseWheelMoved(sender: Object; e: MouseWheelEven-
tArgs);
begin
   if (e.Delta > 0) then
       minRadius -= 1
   else
      minRadius += 1;

   wind.SetTitle('minRadius = ' + minRadius);
end;
```

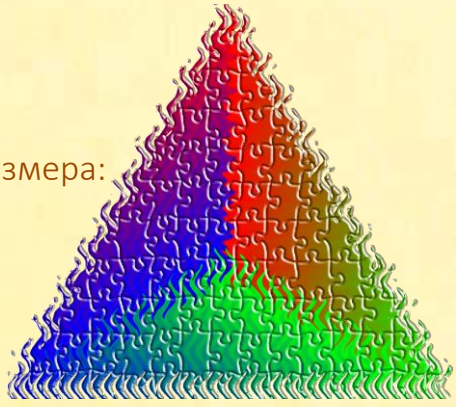
В **процедуре Draw** поворачиваем треугольник вершиной *вверх*:

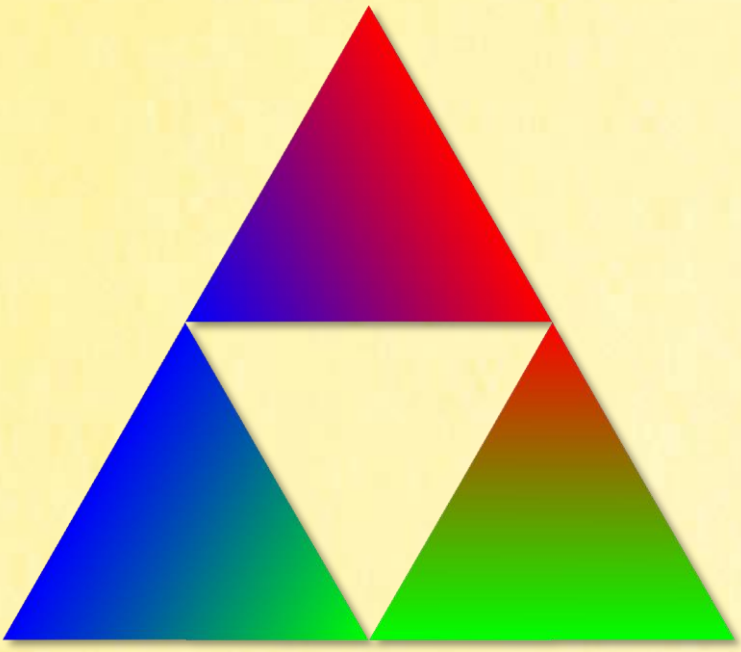
```
procedure Draw();
begin
   // размеры кольца:
```

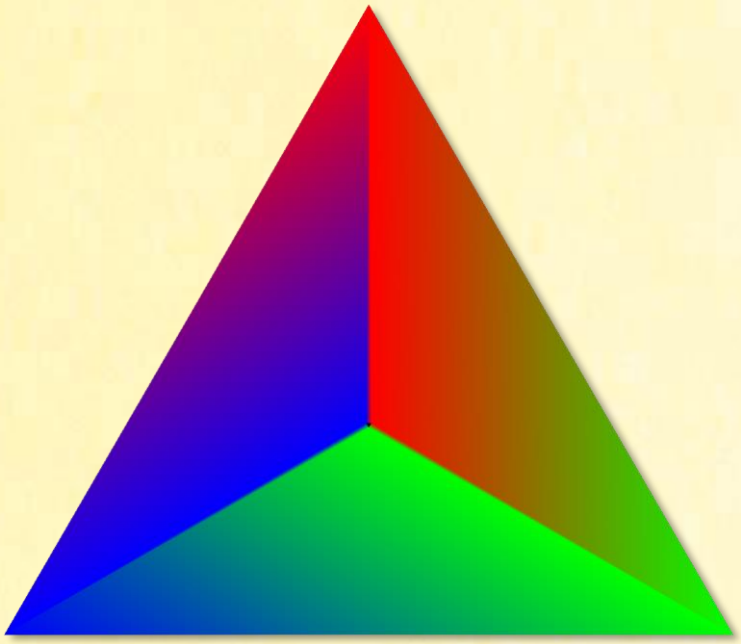
```
    var maxRadius := 700;
    // центр кольца:
    var xc := WIDTH / 2;
    var yc := HEIGHT / 2 + 100;
    // начальный угол:
    var angle := 0.0;
    // приращение угла:
    var angleStep := PI / numVertex;

    // создаём массив вершин:
    va := new VertexArray(PrimitiveType.TrianglesStrip);
    for var i := 0 to numVertex do
    begin
      // цвет вершины:
      var clr := HSV2RGB(angle * 360.0 / PI / 2);
      // координаты:
      var ugol := angle - PI / 2;
      var px := xc + Cos(ugol) * maxRadius;
      var py := yc + Sin(ugol) * maxRadius;

      // добавляем вершину в массив:
      var v := new Vertex(new Vector2f(px, py), clr);
      va.Append(v);
      angle += angleStep;
      ugol := angle - PI / 2;
      px := xc + Cos(ugol) * minRadius;
      py := yc + Sin(ugol) * minRadius;
      v := new Vertex(new Vector2f(px, py), clr);
      va.Append(v);
      angle += angleStep;
    end;

    // чертим треугольники:
    wind.Draw(va);
end;
```



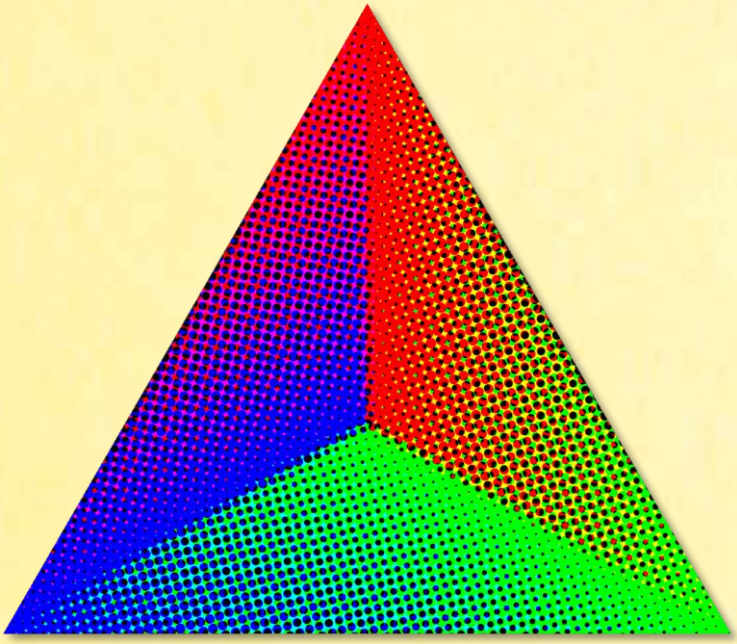
Теперь вы можете делать дырки любого размера:

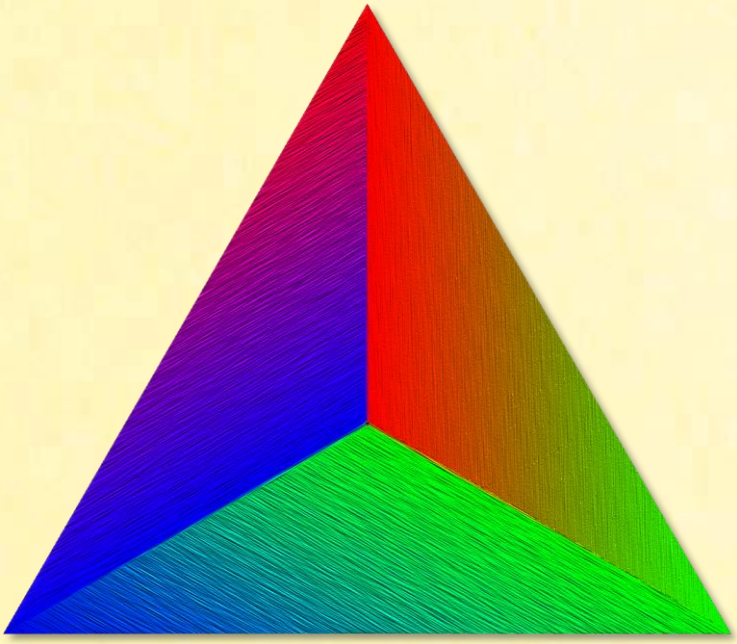


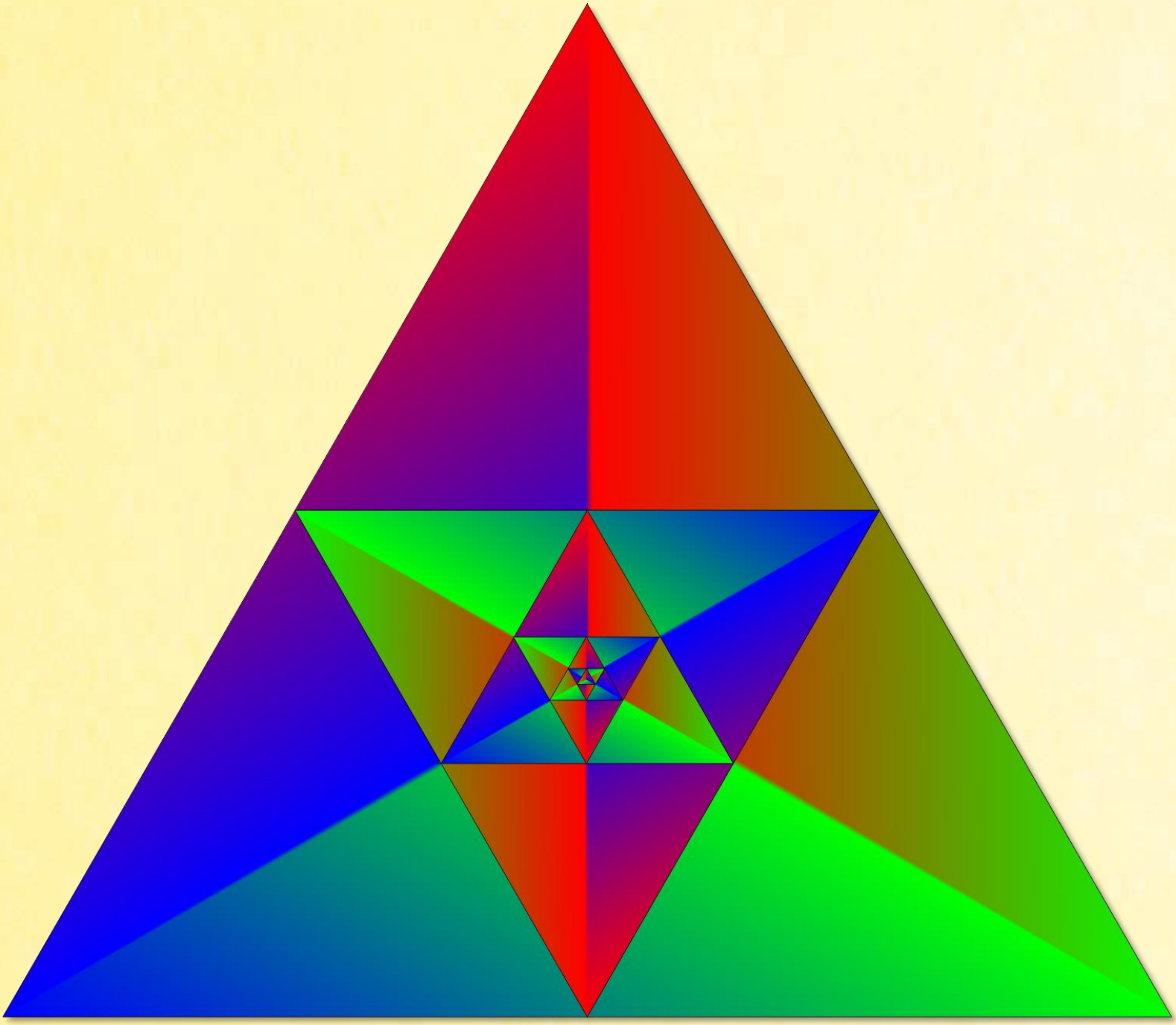


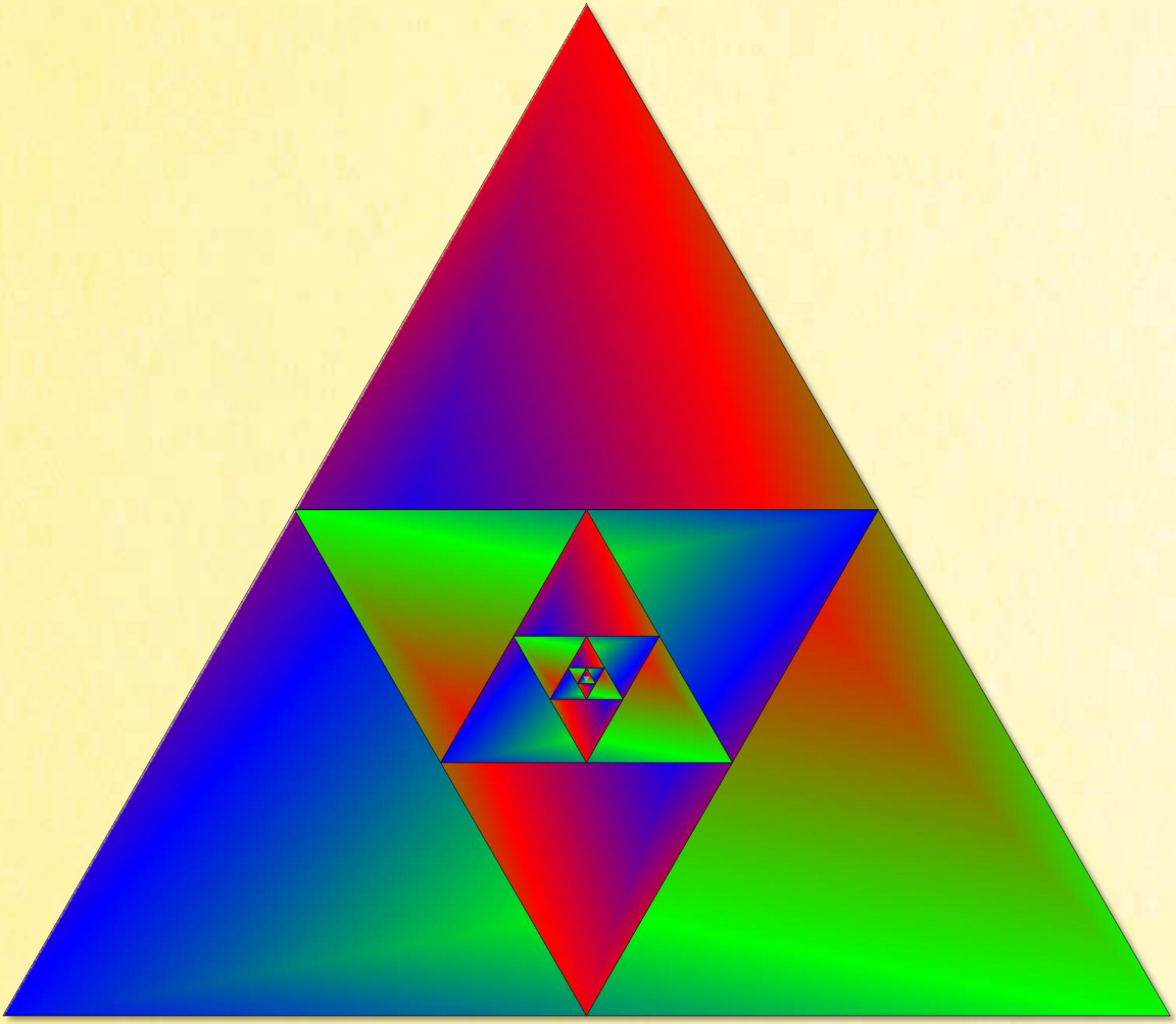


А если этого мало, то продолжайте творческое программирование в *Фотошопе*:









Класс Texture (Текстура)

Texture – это класс изображений, которые выводятся на экран.

Текстуры хранятся в **памяти видеокарты**, поэтому доступ к ним очень быстрый. Различные графические объекты (прямоугольники, круги, многоугольники, спрайты) получают ссылки на текстуры и печатают их на экране. Сами текстуры только хранят изображения, но не могут напечать себя на экране без посредников.

Текстуры нельзя уничтожать, пока на них ссылаются объекты программы.

Созданную текстуру нельзя изменить, но можно загрузить в неё другую текстуру, картинку или массив пикселей.

В текстуру можно загрузить с диска графические файлы в следующих **форматах**: *bmp, png, tga, jpg, gif, psd, hdr* и *pic*.

Для полностью **непрозрачных** текстур годятся картики в форматах *bmp* и *jpg*, для текстур с частично или полностью **прозрачными** участками – png.

Обратите внимание, что текстуру нельзя непосредственно напечатать на экране! Ссылку на неё нужно передать какому-либо графическому объекту.

Прямоугольники типа *RectangleShape* имеют свойство **Texture**, которому можно присвоить ранее созданную текстуру. Вот как это делается.

Загружаем текстуру из файла на диске:

```
var tex := new Texture('Media/ChessBoard.png');
```

Создаём прямоугольник (квадрат) по размерам текстуры:

```
var rect := new RectangleShape(new Vector2f(600, 600));
```

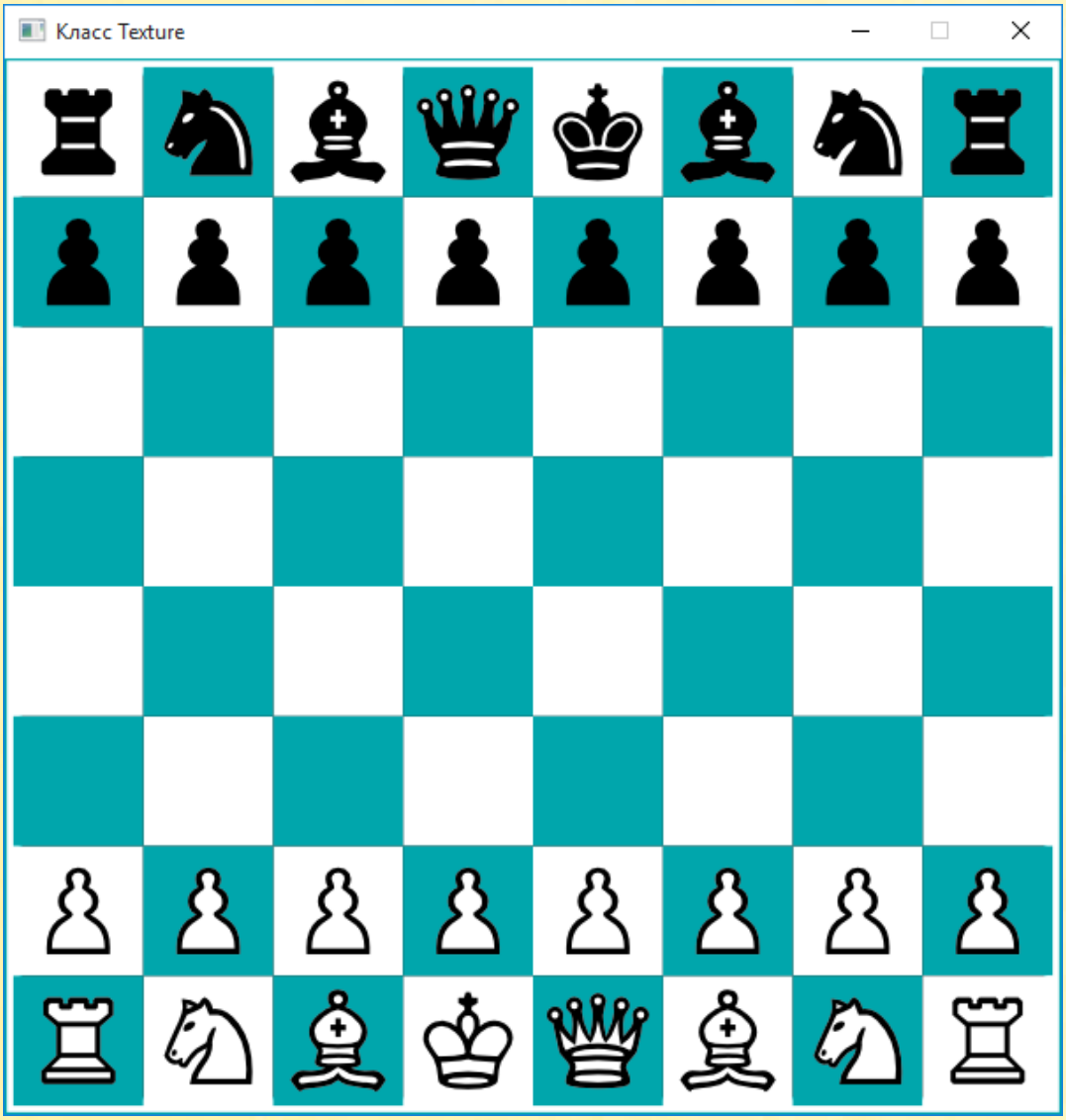
Присваиваем текстуру прямоугольнику:

```
rect.Texture := tex;
```

И рисуем его в окне приложения:

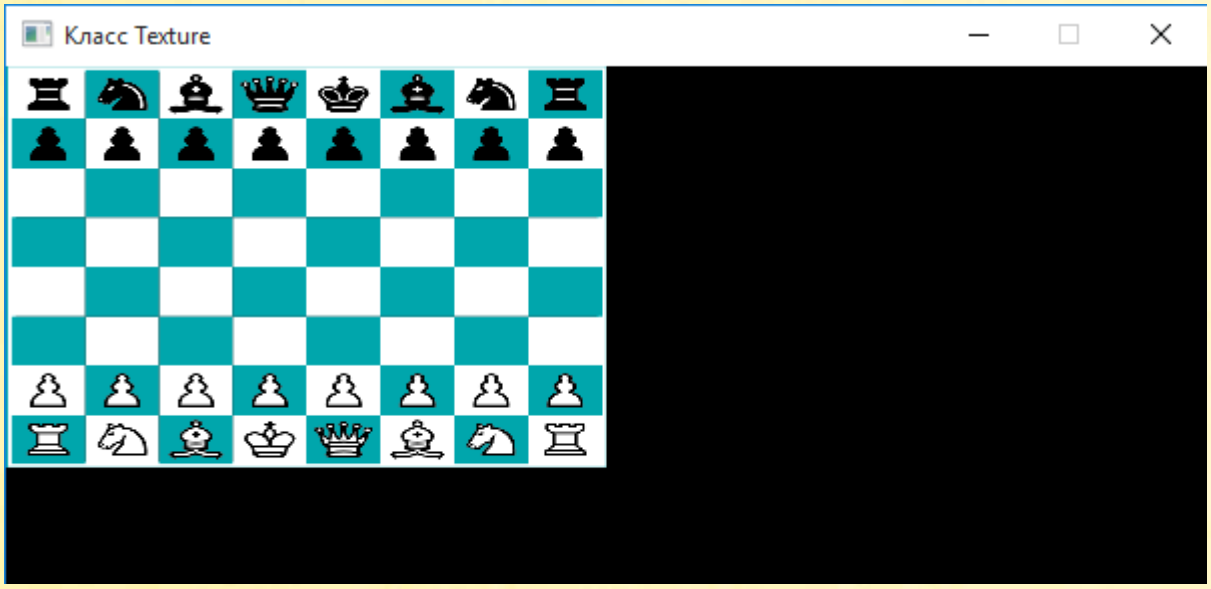
```
wind.Draw(rect);
// показываем на экране:
wind.Display();
```

Текстура – на экране:



Текстура вписывается в прямоугольник. Если его размеры отличаются от размеров текстуры, то она **масштабируется**. При этом могут измениться и пропорции:

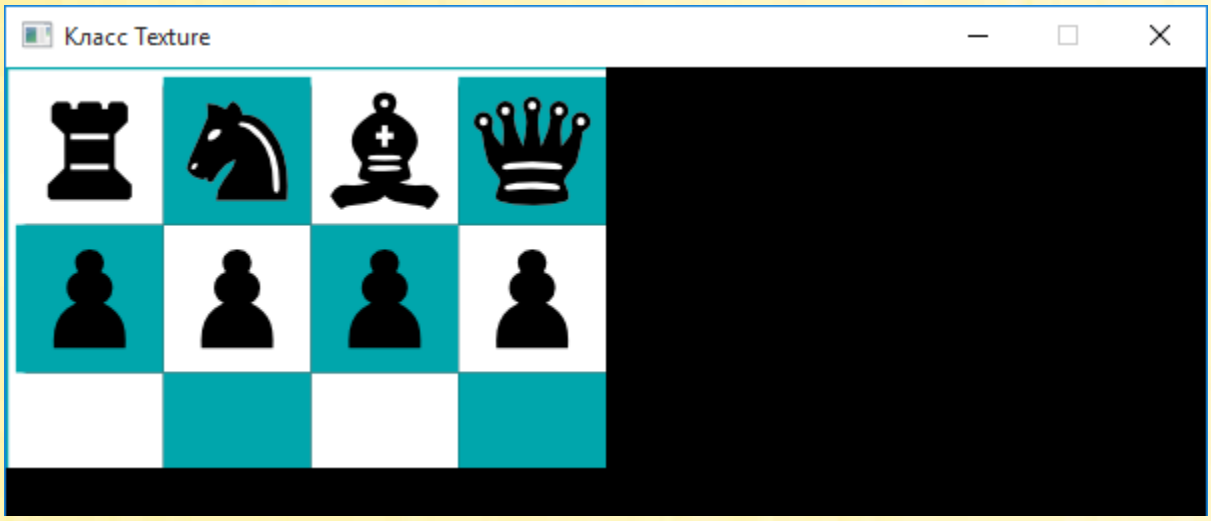
```
var tex := new Texture('Media/ChessBoard.png');
//var rect := new RectangleShape(new Vector2f(600, 600));
var rect := new RectangleShape(new Vector2f(300, 200));
rect.Texture := tex;
wind.Draw(rect);
// показываем на экране:
wind.Display();
```



Иногда нужно показать на экране только **часть текстуры**. На этот случай у прямоугольников имеется свойство **TextureRect**, которое устанавливает или возвращает прямоугольную область, которая вырезается из заданной текстуры:

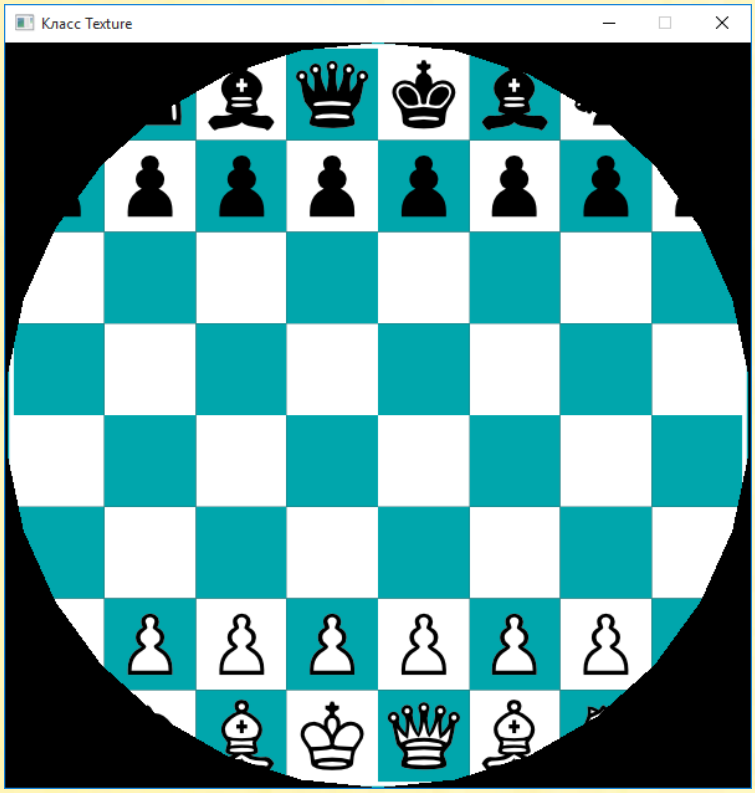
```
var tex := new Texture('Media/ChessBoard.png');
//var rect := new RectangleShape(new Vector2f(600, 600));
var rect := new RectangleShape(new Vector2f(300, 200));
rect.TextureRect := new IntRect(0,0, 300,200);
rect.Texture := tex;
wind.Draw(rect);
// показываем на экране:
wind.Display();
```

При этом текстура не масштабируется, а на экране печатается только её часть:



Свойства *Texture* и *TextureRect* имеются и у кругов:

```
var tex := new Texture('Media/ChessBoard.png');
var circle := new CircleShape(300);
//circle.TextureRect := new IntRect(0, 0, 300, 200);
circle.Texture := tex;
wind.Draw(circle);
wind.Display();
```

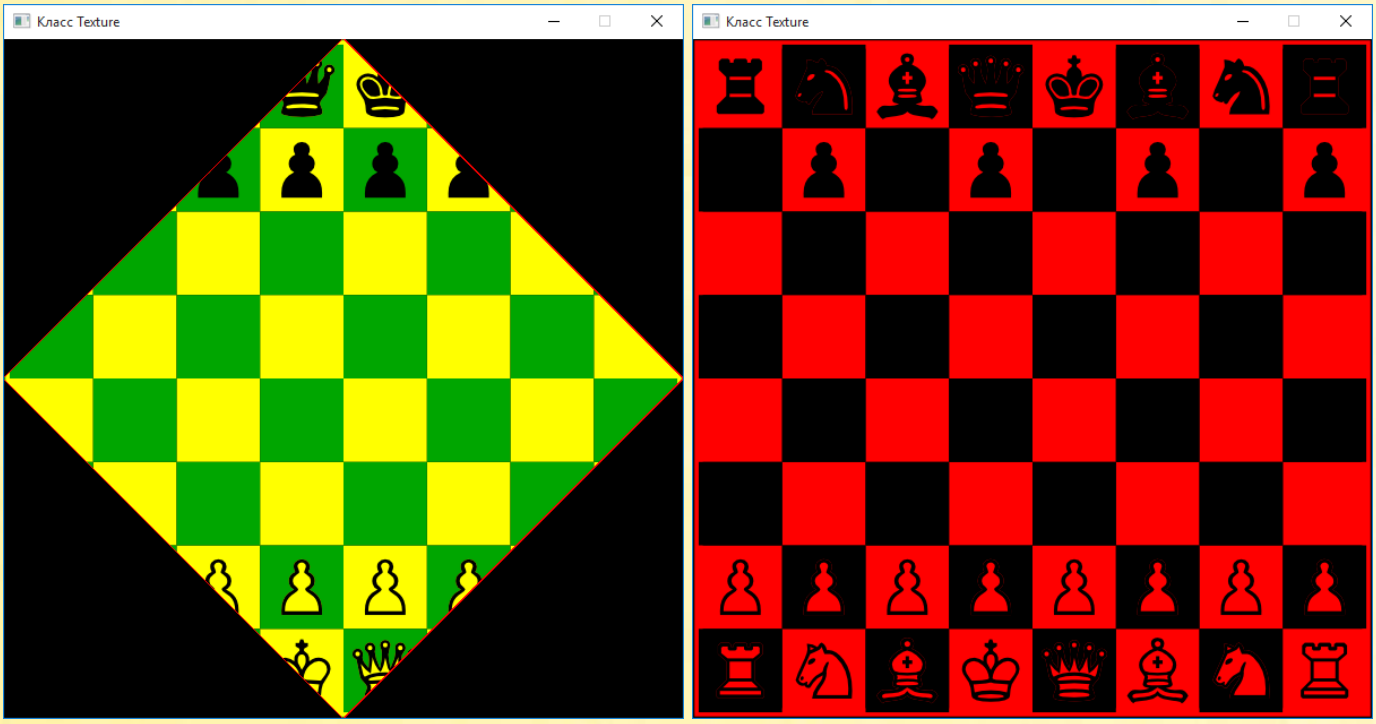


Но прямоугольная текстура обрезается по границам круга:

Те же самые свойства вы найдёте и у **многоугольников**.

```
var tex := new Texture('Media/ChessBoard.png');
var h := 600;
var w := 600;
var convex := new ConvexShape(4);
convex.Position := new Vector2f(0, 0);
convex.FillColor := Color.Yellow;
convex.OutlineColor := Color.Red;
convex.OutlineThickness := 1;
convex.SetPoint(0, new Vector2f(0, h / 2));
convex.SetPoint(1, new Vector2f(w / 2, 0));
convex.SetPoint(2, new Vector2f(w, h / 2));
convex.SetPoint(3, new Vector2f(w / 2, h));
convex.Texture := tex;
wind.Draw(convex);
wind.Display();
```

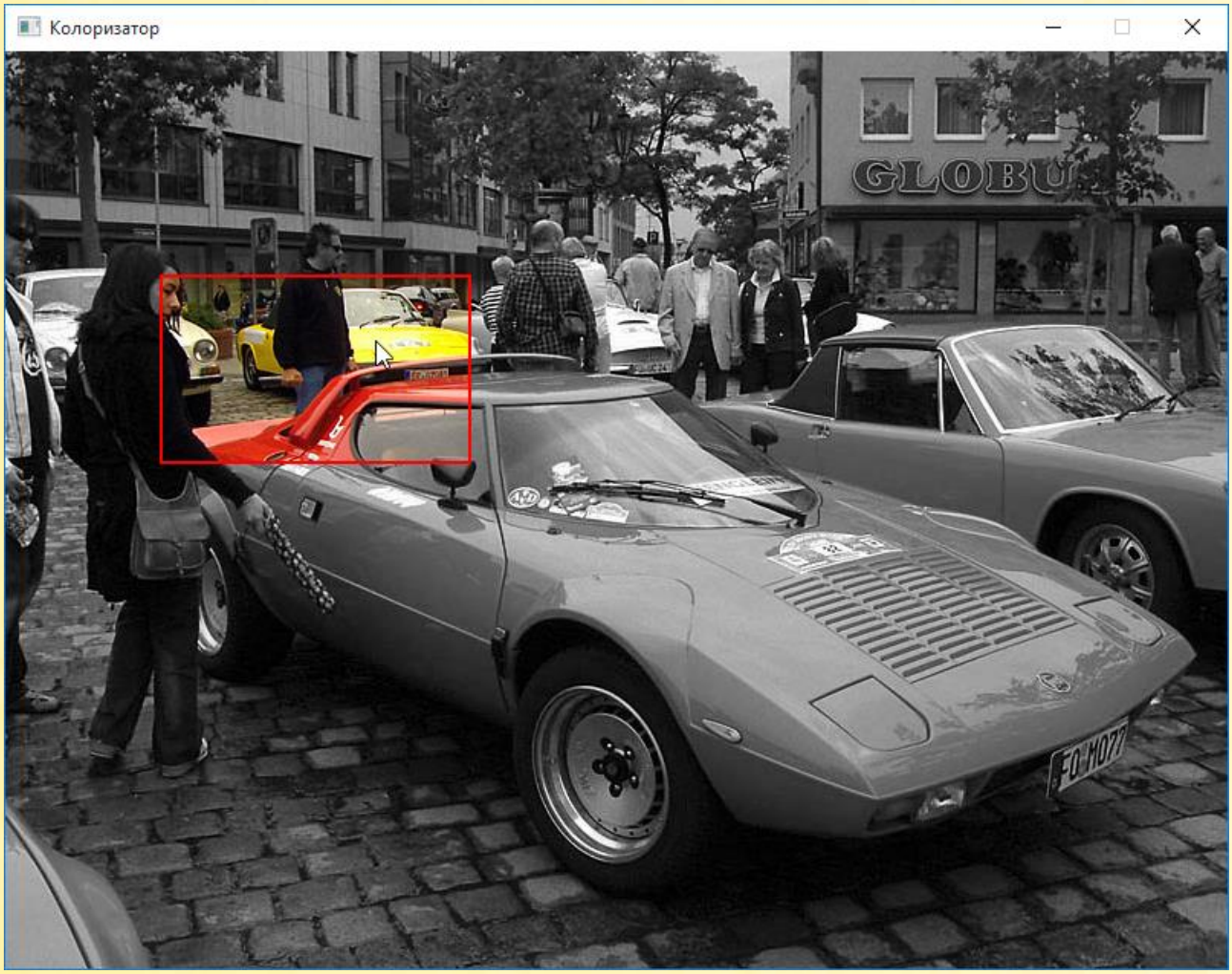
Обратите внимание, что текстуры можно дополнительно **тонировать** цветом Fill-Color фигуры:



Проект *Колоризатор*

В этом проекте мы немного пофокусничаем!

В окне приложения вы видите чёрно-белую картинку, по которой мышкой можно двигать прямоугольную рамку. А в рамке серое изображение превращается в цветное:



Секрет фокуса, как обычно, очень простой.

Так как окно *SFML* не имеет собственной фоновой картинки, то мы будем печатать спрайт:

```
// фоновый спрайт:
backSprite: Sprite;
```

Размеры окна подгоняем под размеры картинки (и спрайта):

```
const
   // размеры окна:
   WIDTH = 800;
   HEIGHT = 600;
```

Для загрузки картинки нам потребуется текстура:

```
// серая картинка:
texGray: Texture;
```

В процедуре Setup загружаем фоновую картинку из файла и передаём её спрайту:

```
// ПОДГОТОВИТЕЛЬНЫЕ РАБОТЫ
procedure Setup();
begin
   // загружаем текстуру:
   texGray := new Texture('Media/autoGray.jpg');
   // создаём спрайт:
   backSprite := new Sprite(texGray);
end;
```

Вызываем процедуру *Setup* в главном блоке:

```
begin
   // создаём главное окно:
   CreateWindow(WIDTH, HEIGHT, 'Колоризатор');

   // готовимся:
   Setup();
```

Запускаем игровой цикл:

```
// игровой цикл:
while (wind.IsOpen) do
begin
   // вызываем все обработчики событий:
   wind.DispatchEvents();
   // рисуем фоновый спрайт:
   wind.Draw(backSprite);

   // показываем на экране:
   wind.Display();
end;
```

Вот он, фоновый спрайт во всей своей некрасе:

Теперь на сцене должен появиться фокусник (это вы) с колоризатором.

Колоризатор – это банальный прямоугольник:

```
// прямоугольник-колоризатор:
rectColor: RectangleShape;
```

Его размеры выбираем с присущей нам скромностью:

```
// размеры прямоугольника:
RECT_WIDTH = 200;
RECT_HEIGHT = 120;
```

В процедуре Setup создаём прямоугольник означенных размеров с красной рамкой по бокам:

```
// создаём прямоугольник-колоризатор:
rectColor := new RectangleShape(new Vector2f(RECT_WIDTH,
                                             RECT_HEIGHT));
rectColor.OutlineThickness := 2;
rectColor.OutlineColor := Color.Red;
```

В игровом цикле показываем прямоугольник на экране:

```
// рисуем прямоугольник:
wind.Draw(rectColor);
```

Пока он белый и неподвижный:

Давайте дадим ему жизни! И начнём с мышиных передвижений.

Тут уж нам не обойтись без **обработчиков событий**:

```
// обработчик перемещения мышки:
wind.MouseMoved += Window_MouseMoved;
// обработчик нажатия кнопки мышки:
wind.MouseButtonPressed += Window_MouseButtonPressed;
// обработчик отпускания кнопки мышки:
wind.MouseButtonReleased += Window_MouseButtonReleased;
```

По традиции мы записываем их в процедуре **CreateWindow**.

Двигательный процесс начинается после нажатия на кнопку мышки, и мы попадаем в процедуру Window_MouseButtonPressed. Здесь мы должны убедиться, что мышка находится на прямоугольнике:

```
// НАЖИМАЕМ КНОПКУ МЫШКИ
procedure Window_MouseButtonPressed(sender: object; e: MouseButtonE-
ventArgs);
begin
   if (not rectColor.GetGlobalBounds().Contains(e.X, e.Y)) then
       exit;
```

Текущие координаты мышки следует запомнить в глобальной переменной mousePos:

```
// координаты мышки:
mousePos: Vector2f;
```

Они нам ещё пригодятся:

```
// запоминаем координаты мышки:
var mouseX := e.X;
var mouseY := e.Y;
mousePos := new Vector2f(mouseX, mouseY);
```

И здесь же важно поднять флажок flgMove:

```
// флаг перетаскивания колоризатора:
flgMove: boolean;
```

```
  flgMove := true;
end;
```

Он сообщает программе, что мышка крепко взялась лапками за колоризатор и готова его тащить.

Мышка уцепилась за колоризатор и весело побежала в процедуру Window_MouseMoved. Но быстро сказка сказывается, да не быстро дело делается! В этой процедуре мы должны убедиться, что мышка держит не только прямоугольник, но и кнопку на своей голове:

```
// ПЕРЕМЕЩАЕМ МЫШКУ
procedure Window_MouseMoved(sender: object; e: MouseMoveEventArgs);
begin
   if (not flgMove) then
       exit;
   if (not Mouse.IsButtonPressed(Mouse.Button.Left)) then
       exit;
```

По текущим координатам мышки:

```
// текущие координаты мышки:
var x := e.X;
var y := e.Y;
```

Определяем вектор её перемещения:

```
// вектор перемещения мышки:
var off := new Vector2f(x, y) - mousePos;
```

За перемещениями мышки следует прямоугольник:

```
// вычисляем новые координаты колоризатора:
rectColor.Position += new Vector2f(off.X, off.Y);
```

И в конце процедуры Window_MouseMoved необходимо запомнить новые координаты мышки:

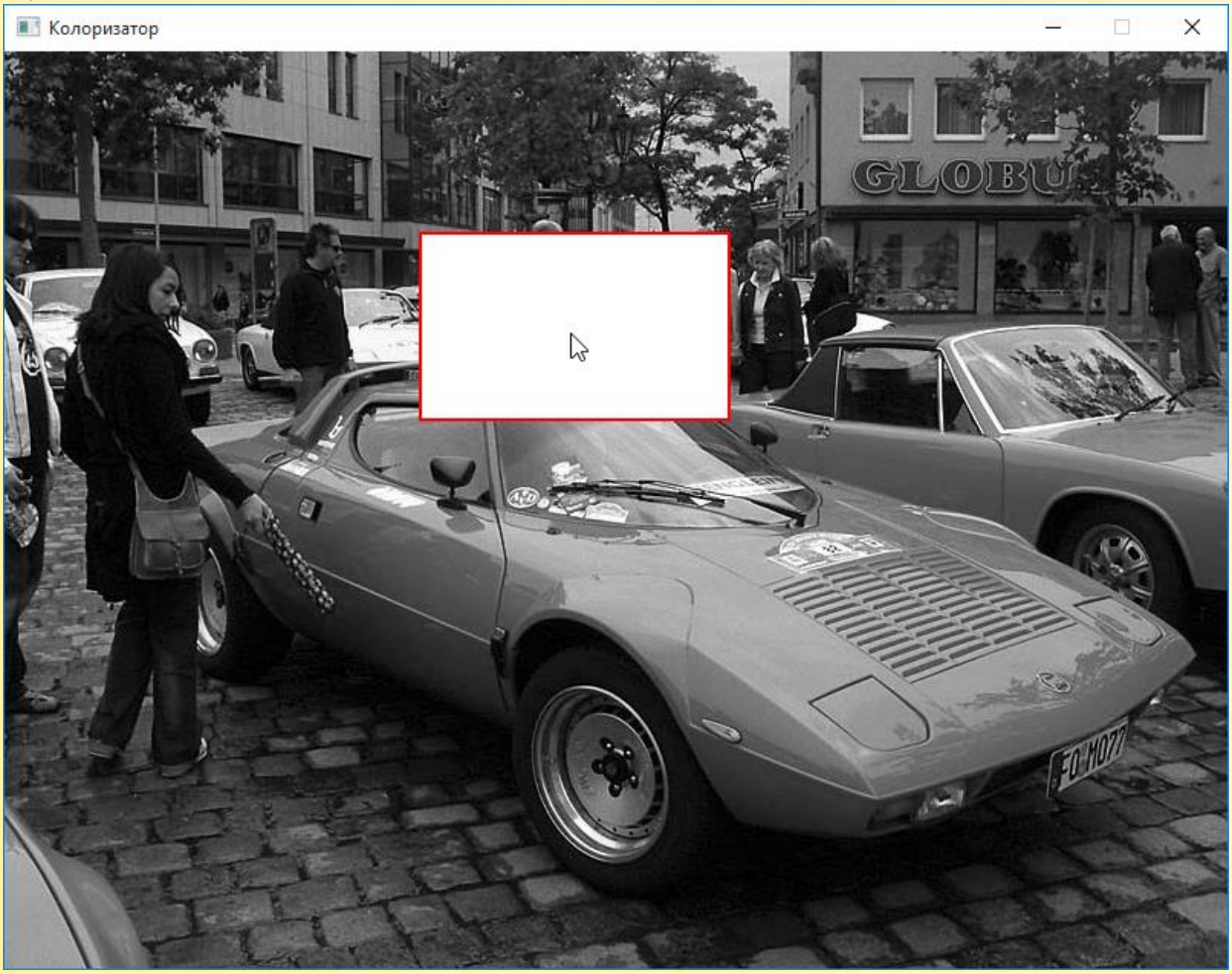
```
   // запоминаем новые координаты мышки:
   mousePos := new Vector2f(x, y);
end;
```

Казалось бы, можно двигать колоризатор, но нет! В процедуре *Window_MouseButtonReleased* нужно сбросить флажок **flgMove**, когда мышка отпустит кнопку:

```
// ОТПУСКАЕМ КНОПКУ МЫШКИ
procedure Window_MouseButtonReleased(sender: object;  e: MouseButtonE-
ventArgs);
begin
   flgMove := false;
end;
```

Если этого не сделать, то мышка будет хватать колоризатор даже мимо него (проверьте!).

Мышка отлично справляется со своими обязанностями, но колоризатор не действует!



Фокус не удался? – Пока да. Но у хорошего фокусника всегда имеется потайной карманчик с кроликом, голубем, ленточкой и цветной картинкой:

Вот эта картинка. Очень даже цветная! И её, эту картинку, мы должны прикрутить к колоризатору.

Чтобы загрузить картинку в программу, нам нужна новая **текстура**:

```
// цветная картинка:
texColor: Texture;
```

В процедуре **Setup** загружаем текстуру:

```
// загружаем цветную картинку:
texColor := new Texture('Media/autoColor.jpg');
```

Передаём её прямоугольнику:

```
// передаём цветную картинку колоризатору:
rectColor.Texture := texColor;
```

Но мы должны учесть, что в прямоугольнике должна быть видна только часть картинки:

```
// и вырезаем из неё прямоугольник
// по размерам колоризатора:
rectColor.TextureRect := new IntRect(0, 0, RECT_WIDTH, RECT_HEIGHT);
```

Так как сразу после старта программы прямоугольник находится в левом верхнем углу окна, то из цветной картинки нужно вырезать прямоугольник, верхний левый угол которого находится там же.

Запускаем программу. Фокус работает! В прямоугольнике тот же самый фрагмент, что и на фоновой картинке, но цветной:

Однако при перемещении колоризатора картинка в нём не обновляется, и мы видим всё тот же фрагмент картинки:

Возвращаемся в процедуру Window_MouseMoved и вырезаем из цветной текстуры прямоугольник под колоризатором:

```
// вырезаем фрагмент фотографии под прямоугольником:
rectColor.TextureRect := new IntRect(Ceil(rectColor.Position.X),
                                     Ceil(rectColor.Position.Y),
                                     RECT_WIDTH, RECT_HEIGHT);
```

Вот теперь фокус удался на славу:

Если вам картинка не по вкусу, то загрузите в неё что-нибудь более сочное и смачное!

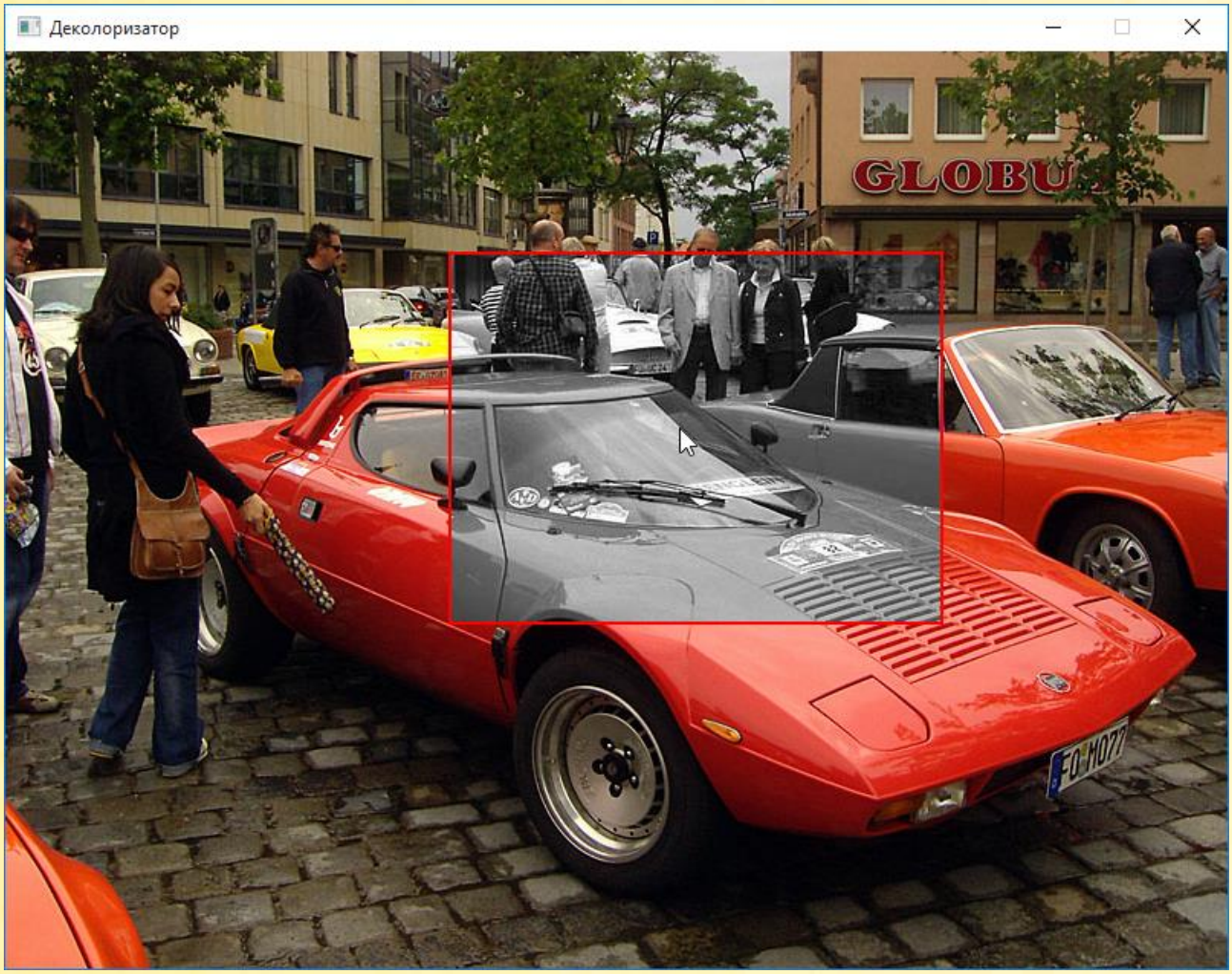
Проект *Деколоризатор*

Простым обменом текстур мы превратим колоризатор в деколоризатор:

```
// создаём спрайт:
backSprite := new Sprite(texColor);

// создаём прямоугольник-деколоризатор:
rectColor := new RectangleShape(new Vector2f(RECT_WIDTH,
                                             RECT_HEIGHT));
```

```
rectColor.OutlineThickness := 2;
rectColor.OutlineColor := Color.Red;
// передаём серую картинку деколоризатору:
rectColor.Texture := texGray;
```

Теперь фоновая картинка цветная, а в рамке – серая:



Проект *Фотоувеличитель*

А теперь переделаем *колоризатор* в **фотоувеличитель**! Но не в тот прибор, который в былые времена служил для печати фотографий, а в компьютерную **лупу**, которая умеет увеличивать фотографии.

Лупы бывают и прямоугольные, но обычно они имеют круглую форму, поэтому место прямоугольника в нашем проекте займёт круг:

```
// круг-увеличитель:
circle: CircleShape;
```

Вот такого радиуса:

```
// радиус круга:
RADIUS = 120;
```

В процедуре Setup создаём круг и передаём ему цветную картинку:

```
// ПОДГОТОВИТЕЛЬНЫЕ РАБОТЫ
procedure Setup();
begin
   // загружаем цветную картинку:
   texColor := new Texture('Media/autoColor.jpg');

   // создаём круг-увеличитель:
   circle := new CircleShape(RADIUS);
   // контур:
   circle.OutlineThickness := 2;
   circle.OutlineColor := Color.Red;

   // передаём цветную картинку увеличителю:
   circle.Texture := texColor;
   // и вырезаем из неё прямоугольник
   // по размерам увеличителя:
   circle.TextureRect := new IntRect(0, 0, RADIUS*2, RADIUS*2);
```

Точно такая же картинка будет и у фонового спрайта:

```
   // создаём спрайт:
   backSprite := new Sprite(texColor);
end;
```

Чёрно-белая картинка теперь не нужна, поскольку и фон, и увеличенный фрагмент мы берём из цветной текстуры.

Так как мы хотим изменять масштаб фрагмента в увеличителе, то придётся пойти на хитрость. Когда мышка передвигается с нажатой кнопкой, но без круга, то изменяется масштаб:

```
// НАЖИМАЕМ КНОПКУ МЫШКИ
procedure Window_MouseButtonPressed(sender: object; e: MouseButtonE-
ventArgs);
begin
   if (not circle.GetGlobalBounds().Contains(e.X, e.Y)) then
       exit;

   // запоминаем координаты мышки:
   var mouseX := e.X;
   var mouseY := e.Y;
   mousePos := new Vector2f(mouseX, mouseY);
   flgMove := true;
end;
```

```
// ПЕРЕМЕЩАЕМ МЫШКУ
procedure Window_MouseMoved(sender: object; e: MouseMoveEventArgs);
begin
   if (Mouse.IsButtonPressed(Mouse.Button.Left)) then
   begin
      // текущие координаты мышки:
      var x := e.X;
      var y := e.Y;

      if (not flgMove) then
      begin
         // перемещение:
         var dy := mousePos.Y-y;
         zoom += 0.005*dy;
         if (zoom <= 0.5) then
             zoom := 0.5;

         circle.TextureRect := new IntRect(Ceil(circle.Position.X),
                                           Ceil(circle.Position.Y),
                                           Ceil(RADIUS * 2 / zoom),
```

```
                                                Ceil(RADIUS * 2 / zoom));
            wind.SetTitle('Увеличение = ' + zoom);
        end
        else
        begin
            // вектор перемещения мышки:
            var off := new Vector2f(x, y) - mousePos;
            // вычисляем новые координаты увеличителя:
            circle.Position += new Vector2f(off.X, off.Y);
            // вырезаем фрагмент фотографии под кругом:
            circle.TextureRect := new IntRect(Ceil(circle.Position.X),
                                              Ceil(circle.Position.Y),
                                              Ceil(RADIUS * 2 / zoom),
                                              Ceil(RADIUS * 2 / zoom));
        end;
        // запоминаем новые координаты мышки:
        mousePos := new Vector2f(x, y);
    end;
end;
```

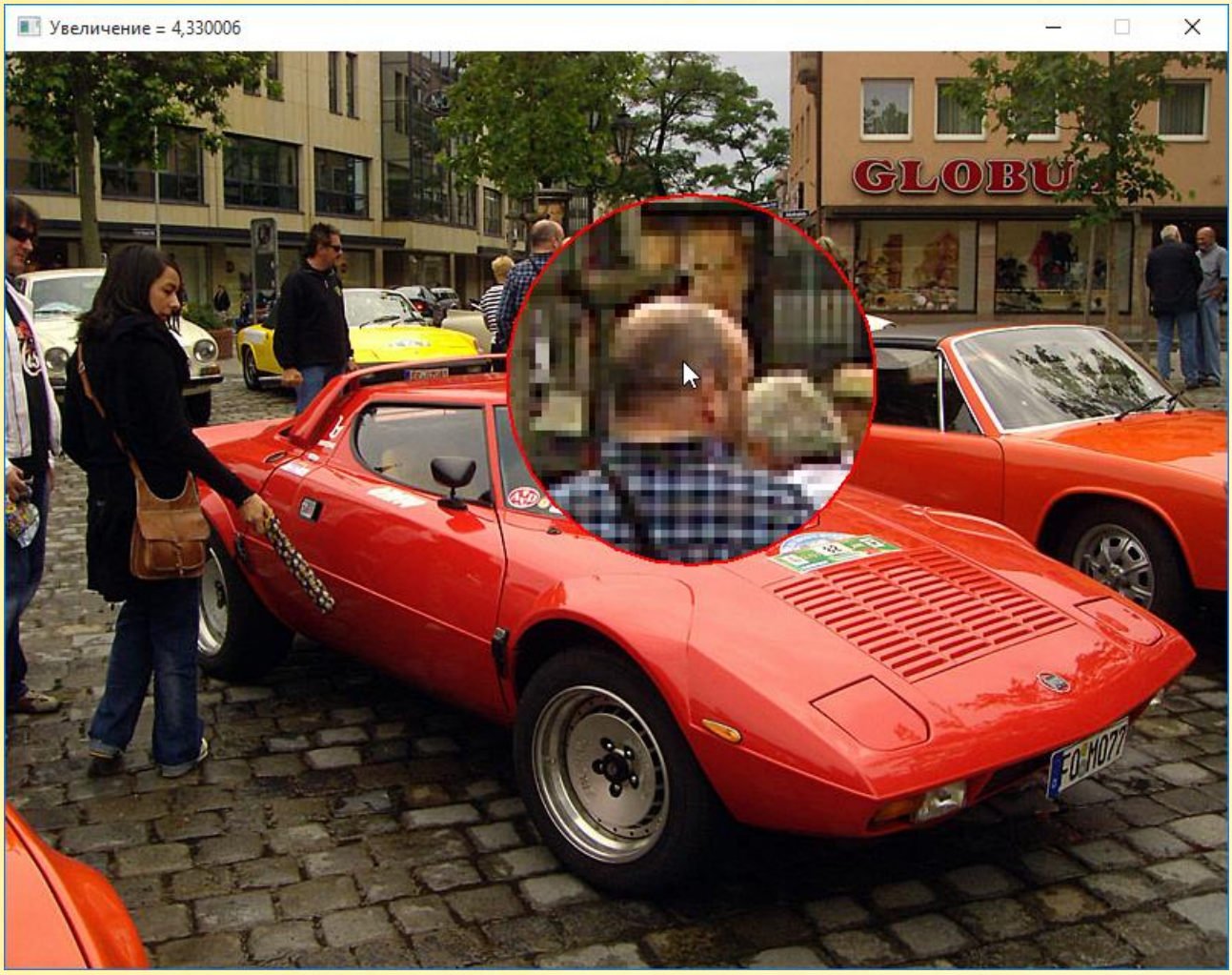
Если увеличение меньше 1 (то есть фрагмент не увеличивается, а уменьшается), то на краях экрана возникают «помехи», так как текстуры не хватает, чтобы заполнить весь круг. Но уменьшать картинку вряд ли есть смысл, поэтому закроем глаза на эти мелкие графические недоразумения.

Начальное **увеличение** равно 1

```
// увеличение:
zoom := 1.0;
```

И картинка под кругом остаётся без изменений:

Увеличиваем изображение – наша лупа работает на славу:



Проект *Картинные каналы*

В этом проекте мы используем наши достижения для манипулирования отдельными пикселями картинки типа *Image*.

В процедуре **Setup** загружаем обе картинки- текстуры и создаём фоновый **спрайт**:

```
// ПОДГОТОВИТЕЛЬНЫЕ РАБОТЫ
procedure Setup();
begin
```

```
   // загружаем текстуры:
   texGray := new Texture('Media/autoGray.jpg');
   // загружаем цветную картинку:
   texColor := new Texture('Media/autoColor.jpg');
   // создаём спрайт:
   backSprite := new Sprite(texGray);
```

Пиксели текстуры изменить нельзя, поэтому создаём из цветной текстуры **картинку** типа *Image*:

```
// создаём картинку из цветной текстуры:
var img := texColor.CopyToImage();
```

Свойство **Pixels** возвращает массив байтов картинки:

```
// получаем массив пикселей:
var pix := img.Pixels;
```

Каждый пиксель занимает в массиве 4 байта. **Первый байт** – красная составляющая цвета пикселя. **Второй байт** – зелёная составляющая цвета пикселя. **Третий байт** – синяя составляющая цвета пикселя. **Четвёртый байт** – прозрачность цвета пикселя.

Пусть мы хотим посмотреть красный канал цветной картинки. Красные составляющие пикселей занимают в памяти ячейки 0, 4, 8, 12 и так далее. Их мы оставляем без измениния. Так же как и прозрачность. Зелёные и синие составляющие имеют в массиве индексы на 1 и 2 больше красных. Их следует **обнулить**:

```
// оставляем только красную составляющую цвета:
var id := 0;
var i := 0;
while i < pix.Length do
begin
   pix[i + 1] := 0;
   pix[i + 2] := 0;
   i += 4;
end;
```

Теперь в массиве pix остались неизменными только байты красного канала.

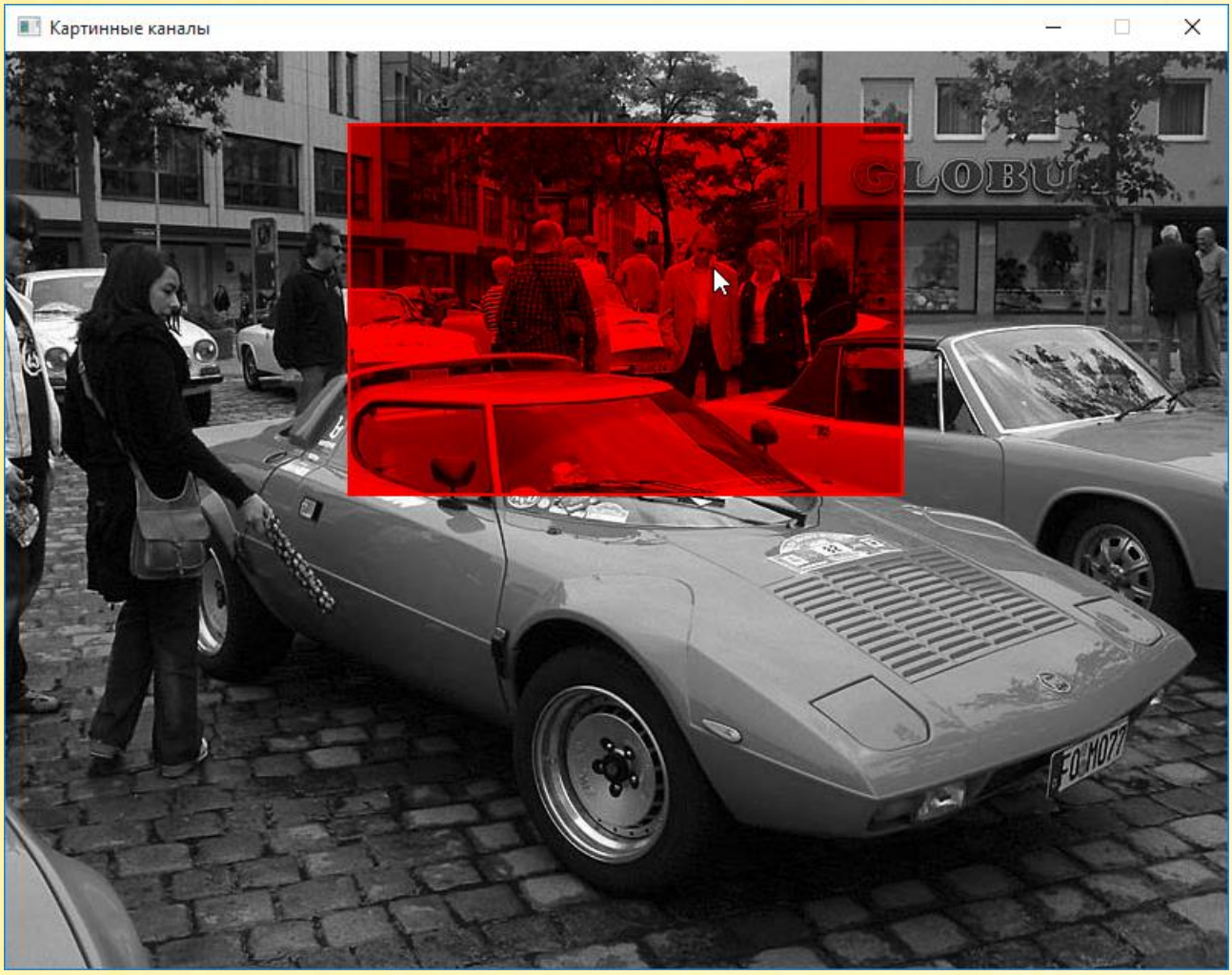
Из этого массива байтов можно создать новую текстуру. Но если цветная текстура вам больше не нужна, то вы можете использовать её:

```
// создаём новую текстуру:
//var tex := new Texture(800,600);
// заполняем пиксельным массивом:
//tex.Update(pix);
texColor.Update(pix);
```

Но не забудьте передать изменённую текстуру колоризатору:

```
   // создаём прямоугольник-колоризатор:
   rectColor := new RectangleShape(new Vector2f(RECT_WIDTH,
                                                RECT_HEIGHT));
   rectColor.OutlineThickness := 2;
   rectColor.OutlineColor := Color.Red;

   // передаём цветную картинку колоризатору:
   rectColor.Texture := texColor;
   // и вырезаем из неё прямоугольник
   // по размерам колоризатора:
   rectColor.TextureRect := new IntRect(0, 0, RECT_WIDTH,
                                              RECT_HEIGHT);

   mousePos := new Vector2f(0, 0);
end;
```

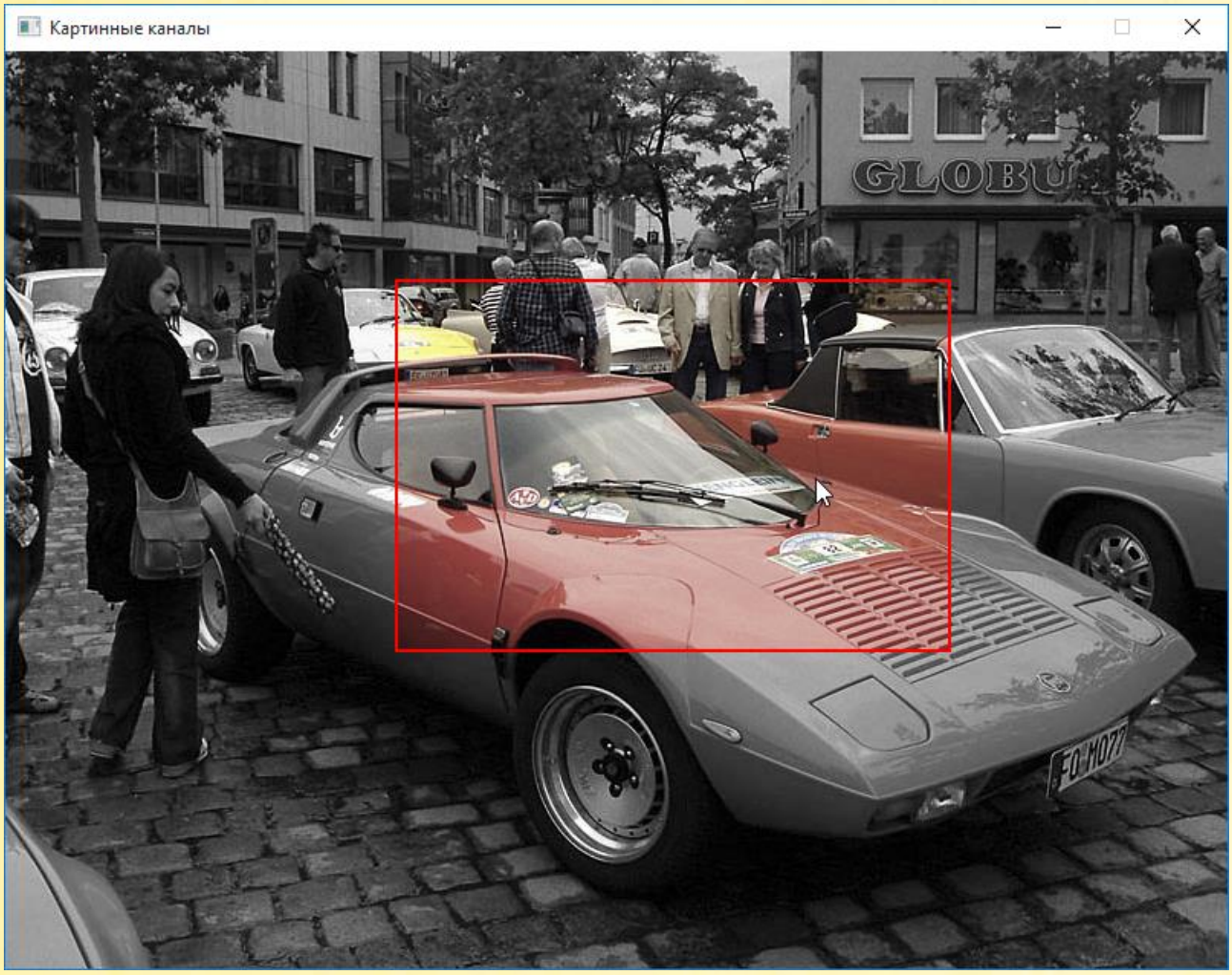
Запускаем программу – в окне колоризатора мы видим только красный канал цветной картинки:



Точно так же вы можете извлечь зелёный и синий каналы. Ещё проще изменить прозрачность картинки:

```
// изменяем прозрачность:
var id = 3;
for (var i = id; i < pix.Length; i += 4)
{
    pix[i + 0] = 111;
}
```

Через полупрозрачную цветную картинку видна и чёрно-белая:



Как видите, совсем нетрудно изменять пиксели исходной картинки.

Проект *Фильтры*

Научившись изменять пиксели исходной картинки, мы можем без труда написать несколько **фильтров**.

За основу примем проект *Деколоризатор* и быстренько напишем первый фильтр – **негатив**:

```
// ПОДГОТОВИТЕЛЬНЫЕ РАБОТЫ
```

```
procedure Setup();
begin
   // загружаем текстуры:
   texGray := new Texture('Media/autoGray.jpg');
   // загружаем цветную картинку:
   texColor := new Texture('Media/autoColor.jpg');
   // создаём спрайт:
   backSprite := new Sprite(texColor);

   // создаём прямоугольник:
   rectColor := new RectangleShape(new Vector2f(RECT_WIDTH,
                                                RECT_HEIGHT));
   rectColor.OutlineThickness := 2;
   rectColor.OutlineColor := Color.Red;

   // создаём картинку из цветной текстуры:
   var img := texColor.CopyToImage();
   // получаем массив пикселей:
   var pix := img.Pixels;

    // негатив:
    var id := 0;
    var i := id;
    while i < pix.Length do
    begin
       pix[i + 0] := 255 - pix[i + 0];
       pix[i + 1] := 255 - pix[i + 1];
       pix[i + 2] := 255 - pix[i + 2];
      i += 4;
    end;

    // заполняем текстуру пиксельным массивом:
    texGray.Update(pix);

    // передаём серую картинку колоризатору:
    rectColor.Texture := texGray;
    // и вырезаем из неё прямоугольник
    // по размерам колоризатора:
    rectColor.TextureRect := new IntRect(0, 0, RECT_WIDTH,
                                             RECT_HEIGHT);

    mousePos := new Vector2f(0, 0);
end;
```

В рамке можно видеть негативное (инвертированное) изображение исходной картинки:

Очень полезный фильтр, превращающий цветное изображение в чёрно-белое:

```
// оттенки серого:
var id := 0;
var i := id;
while i < pix.Length do
begin
   var lum := Ceil(pix[i + 0]*0.3 + pix[i + 1]*0.59 +
                   pix[i + 2]*0.11);
   pix[i + 0] := lum;
   pix[i + 1] := lum;
   pix[i + 2] := lum;
   i += 4;
```

```
end;
```

Известный «плакатный» фильтр, уменьшающий число цветов на картинке:

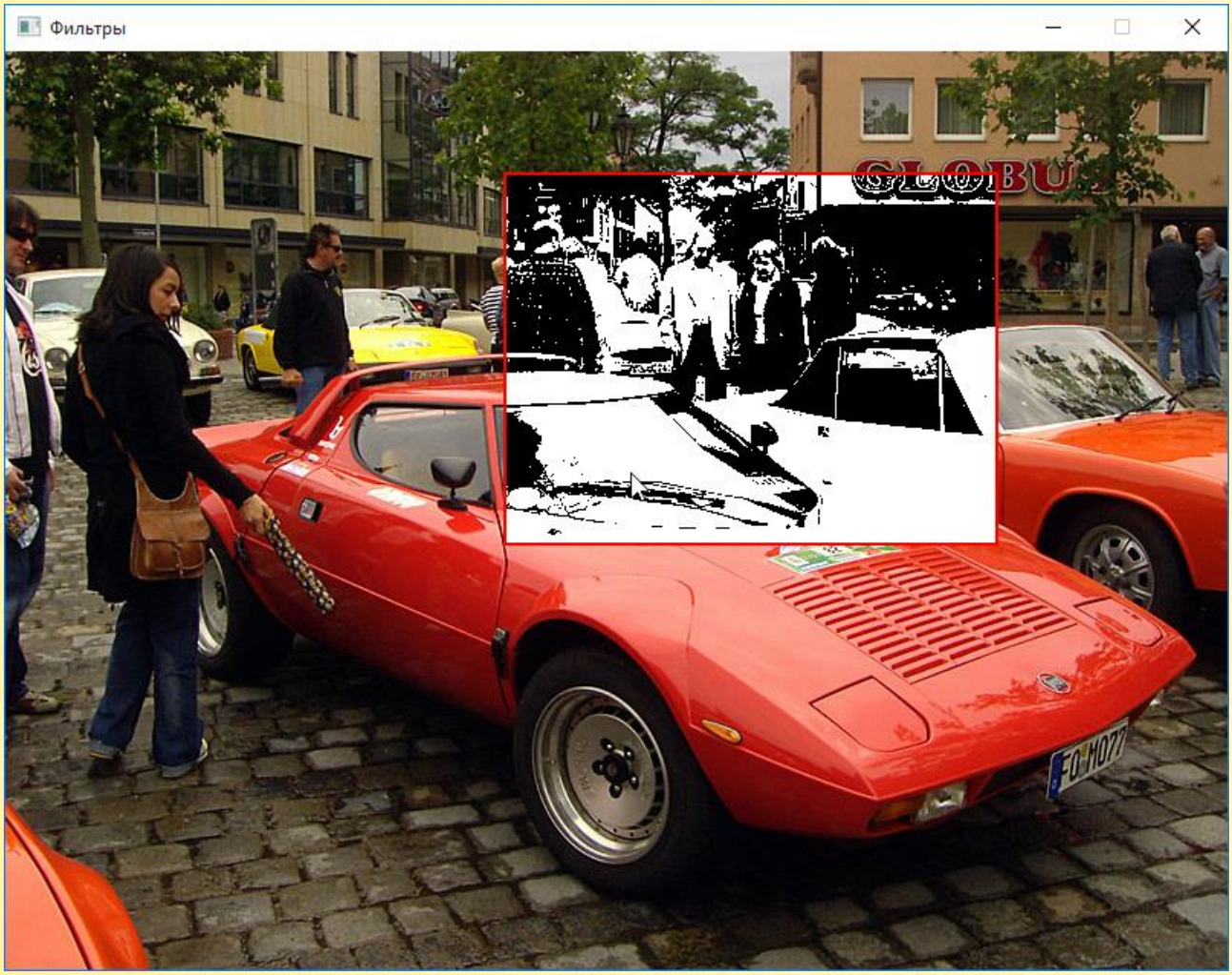
```
// постеризация:
// уровни 2..255:
var levels := 3;
var levels1 := levels - 1;
var id := 0;
var i := id;
while i < pix.Length do
begin
   var rlevel := Ceil( ((pix[i + 0] * levels) shr 8) *
                          255 / levels1);
   var glevel := Ceil( ((pix[i + 1] * levels) shr 8) *
```

```
                                255 / levels1);
        var blevel := Ceil( ((pix[i + 2] * levels) shr 8) *
                                255 / levels1);
        pix[i + 0] := rlevel;
        pix[i + 1] := glevel;
        pix[i + 2] := blevel;
        i += 4;
end;
```

Следующий фильтр оставляет на картинке только белый и чёрный цвет:

```
    // порог:
    var id := 0;
    // уровень 0..255:
```

```
    var level := 128;
    var i := id;
    while i < pix.Length do
    begin
       var max := Math.Max(Math.Max(pix[i + 0], pix[i + 1]),
                                    pix[i + 2]);
       var clr := max < level ? 0 : 255;
       pix[i + 0] := clr;
       pix[i + 1] := clr;
       pix[i + 2] := clr;
       i += 4;
end;
```



Проект *Текстурные координаты*

Пиксели текстуры можно привязать к *вершинам*, которые хранятся в массиве *VertexArray*.

В процедуре Setup загружаем картинку из файла:

```
// ПОДГОТОВИТЕЛЬНЫЕ РАБОТЫ
procedure Setup();
begin
   // загружаем цветную картинку:
   texColor := new Texture('Media/autoColor.jpg');
```

Создаём массив вершин va:

```
// заполняем массив вершин пикселями изображения:
var va := new VertexArray(PrimitiveType.Points);
```

И заполняем его вершинами. Каждая вершина хранит координаты пикселя на экране и на текстуре. В нашем примере они совпадают:

```
// по размерам картинки:
for var j := 0 to HEIGHT-1 do
    for var i := 0 to WIDTH-1 do
    begin
       // координаты пикселя картинки:
       var pos := new Vector2f(i, j);
       // создаём вершину с координатами
       // пикселя на экране и на текстуре:
       var v := new Vertex(pos, new Vector2f(i,j));
       // добавляем вершину в массив:
       va.Append(v);
    end;
```

Это значит, что массив хранит пиксели картинки точно в таком же порядке, как и в оригинале. Но массив вершин ничего не знает о самой текстуре. Её следует задать в экземпляре структуры RenderStates:

```
// задаём текстуру для визуализации вершин:
var states := new RenderStates(texColor);
```

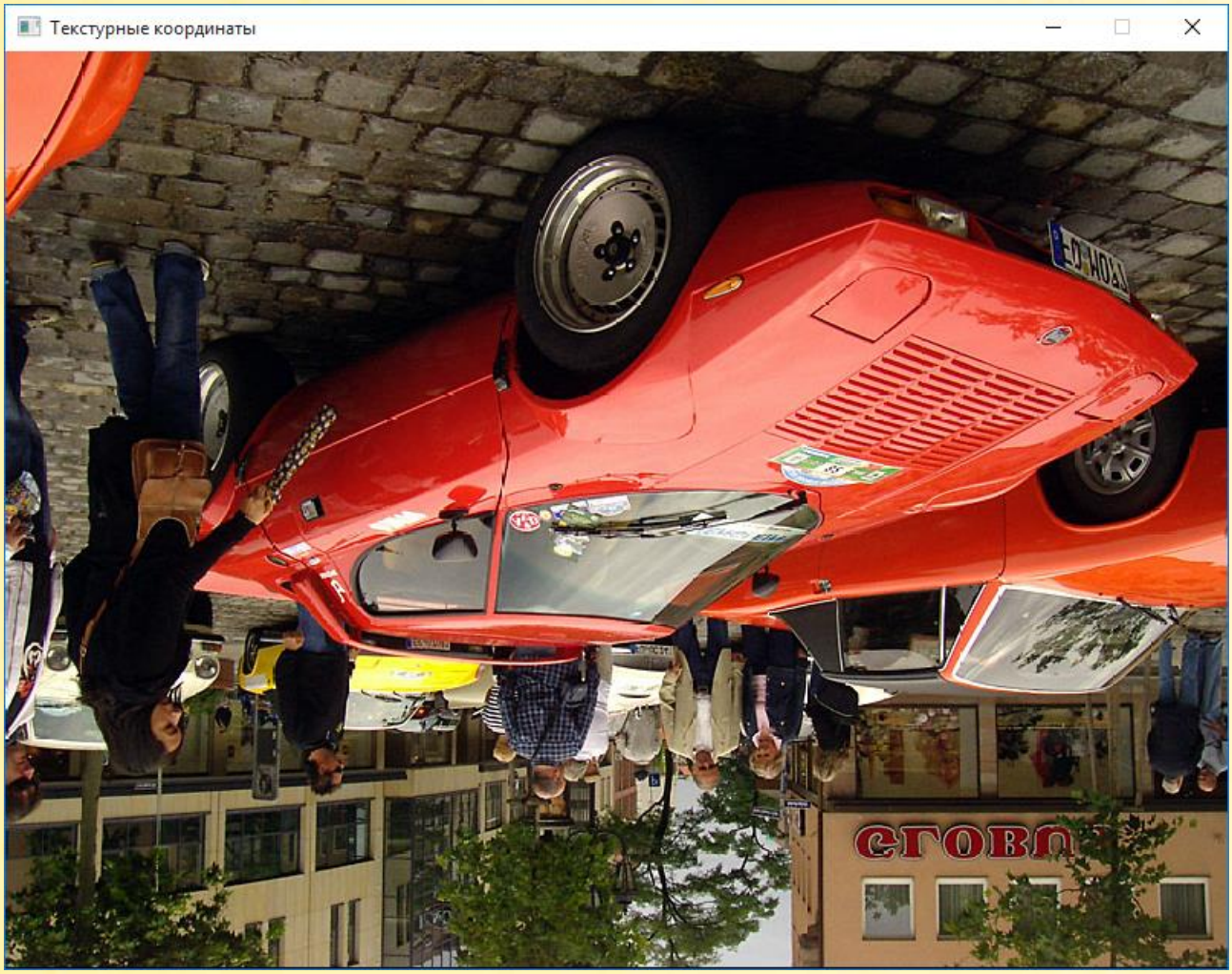
В процедуре Draw мы сообщаем программе, что хотим напечатать вершины, которые хранят координаты заданной текстуры:

```
    // печатаем все вершины на экране:
    wind.Draw(va, states);
    // обновляем экран:
    wind.Display();
end;
```

Запускаем программу – все пиксели на своих местах:

Если бы требовалось только напечатать картинку на экране, то не нужно было бы и городить массив с пикселями. Но мы можем заполнять массив координатами пикселей так, что картинка на экране перевернётся вверх ногами:

```
//var v := new Vertex(pos, new Vector2f(i,j));
var v := new Vertex(pos, new Vector2f(i, HEIGHT - j));
```



Аналогично вы можете отразить картинку и по горизонтали:

```
var v = new Vertex(pos, new Vector2f(WIDTH - i, HEIGHT));
```

Проект *Текстурные координаты 2*

Пиксели текстуры в полном нашем распоряжении, поэтому мы можем делать с ними всё, что захотим.

В этом проекте мы «запылим» картинку чёрными пикселями.

Нам понадобятся такие глобальные переменные:

```
mousePos: Vector2f;
// уровень запылённости картинки:
dust := 0.0;
// массив вершин:
va:VertexArray := nil;
// режим визуализации:
states: RenderStates;
```

Нажимаем кнопку мышки на картинке и запоминаем её координаты:

```
// НАЖИМАЕМ КНОПКУ МЫШКИ
procedure Window_MouseButtonPressed(sender: object; e: MouseButtonE-
ventArgs);
begin
   // запоминаем координаты мышки:
   var mouseX := e.X;
   var mouseY := e.Y;
   mousePos := new Vector2f(mouseX, mouseY);
end;
```

При перемещении мышки по вертикали изменяем значение поля dust:

```
// ПЕРЕМЕЩАЕМ МЫШКУ
procedure Window_MouseMoved(sender: object; e: MouseMoveEventArgs);
begin
   if (Mouse.IsButtonPressed(Mouse.Button.Left)) then
   begin
      // текущие координаты мышки:
```

```
        var x := e.X;
        var y := e.Y;

        // перемещение:
        var dy := mousePos.Y - y;
        // изменяем запылённость картинки:
        dust += dy/10;

        // создаём новый массив вершин:
        CreateVertex();

        // запоминаем новые координаты мышки:
        mousePos := new Vector2f(x, y);
    end;
end;
```

И тут же обновляем массив вершин:

```
// СОЗДАЁМ МАССИВ ВЕРШИН
procedure CreateVertex();
begin
    var v : Vertex;
    // заполняем массив вершин пикселями изображения:
    va := new VertexArray(PrimitiveType.Points);
    // по размерам картинки:
    for var j := 0 to HEIGHT-1 do
        for var i := 0 to WIDTH-1 do
        begin
            // координаты пикселя картинки:
            var pos := new Vector2f(i, j);
```

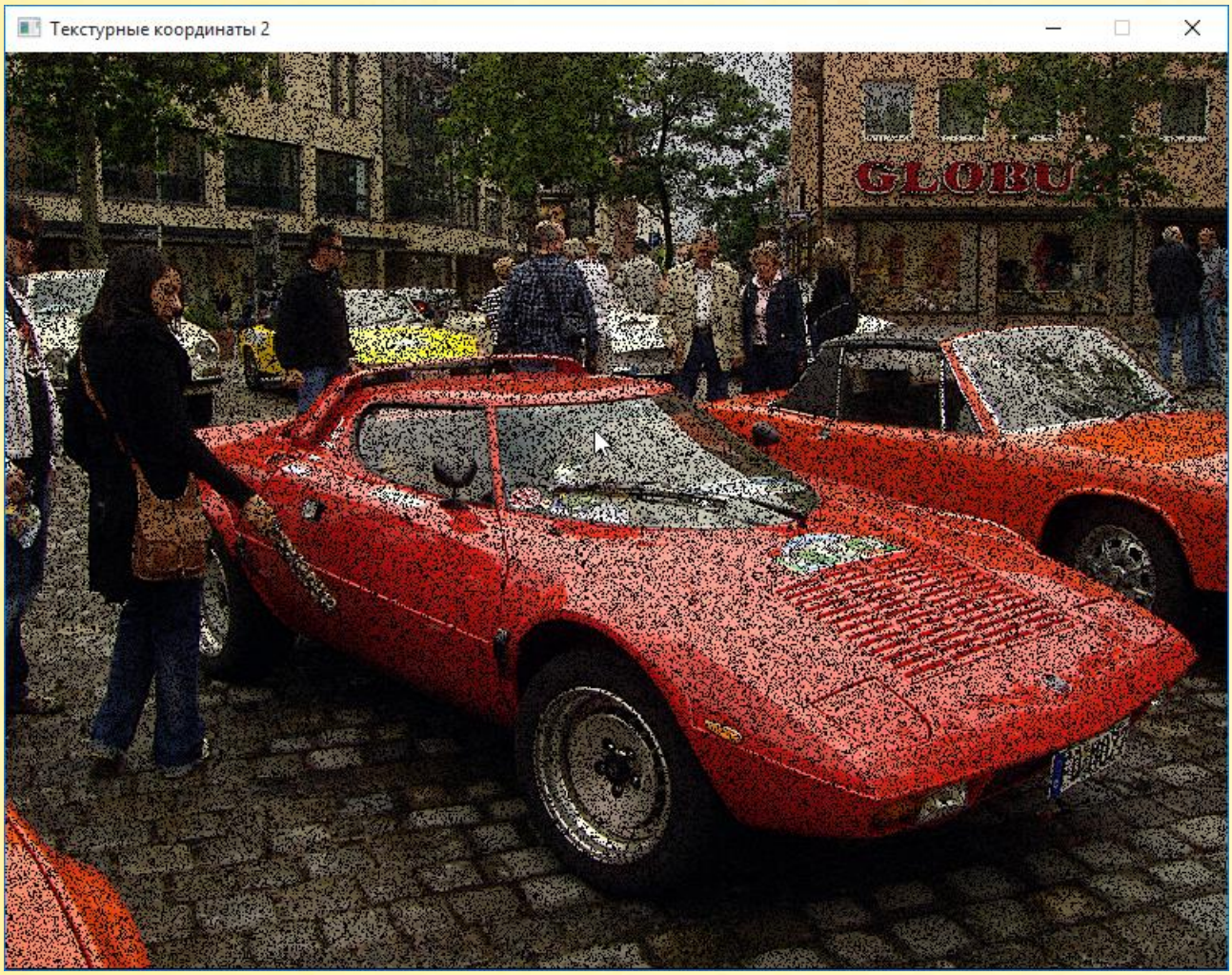
Если случайное число не меньше значения запылённости, то запоминаем в вершине координату текстуры, иначе закрашиваем её в чёрный цвет:

```
    // создаём вершину с координатами
    // пикселя на экране и на текстуре:
    if (rand.Next(100) >= dust) then
        v := new Vertex(pos, new Vector2f(i, j))
    else
        v := new Vertex(pos, Color.Black);
    // добавляем вершину в массив:
```

```
    va.Append(v);
end;
```

Массив вершин заполнен. Можно показывать картинку на экране:

```
    // печатаем все вершины на экране:
    wind.Draw(va, states);
    // обновляем экран:
    wind.Display();
end;
```



Эффект запылённости, конечно, не очень красив, но он показывает, как можно манипулировать пикселями текстуры.

Класс Image (Картинки)

Картинки типа *Image* хранятся в оперативной памяти компьютера, поэтому доступ к ним гораздо более медленный, чем к текстурам. По этой причине они не используются графическими объектами. Но из картинок или из их прямоугольных частей можно создать **текстуры**.

Картинки легко **изменять** - каждый пиксель доступен для чтения и записи. Эти операции очень быстрые, поэтому можно создавать новые картинки непосредственно в работающей программе. Картинки можно **сохранять** на диске для последующего использования.

Если изображения в программе **не изменяются**, то создавайте **текстуры**.
Если изображения в программе **изменяются**, то создавайте **картинки**.

Форматы графических файлов, из которых создаются картинки, такие же, как у текстур.

Непосредственно на экран вывести картинку нельзя, из неё нужно создать **текстуру**. О печати текстур мы уже говорили выше, поэтому перейдём к примерам.

Проект *Радиальные волны*

Если вы бросали камни в воду, то, конечно, обратили внимание на то, что волны, которые разбегаются от места падения камня, совершенно *круглые*. Чтобы загнуть волны в дугу, нам придётся перейти от прямоугольных координат к **полярным**, что мы и сделаем с помощью **формулы**:

```
rad := Sqrt((x - cx) * (x - cx) + (y - cy) * (y - cy)) / wl;
```

А вот формула для вычисления **цвета пикселей**:

```
iclr := Ceil(255 * (1 + Sin(rad)) / 2);
```

За основу возьмём проект *Синусоидные полоски* и переработаем его так, чтобы пиксели изменялись на картинке типа *Image*.

Текущая длина волн хранится в глобальной переменной **wl**:

```
// длина волны синусоиды:
wl: single := 10;
```

Рисовать пиксели мы будем на картинке **img**:

```
// картинка:
img: Image;
```

Затем обновим **текстуру**:

```
// текстура:
tex : Texture;
```

И присвоим текстуру **прямоугольнику**:

```
// прямоугольник:
rect: RectangleShape;
```

В **главном блоке** создаём картинку *img* для рисования пикселями, текстуру по размерам окна (можно из картинки) и прямоугольник таких же размеров:

```
begin
   // создаём главное окно:
   CreateWindow(WIDTH, HEIGHT, 'Радиальные волны');

   // создаём картинку по размерам окна:
```

```
   img := new Image(WIDTH, HEIGHT);
   // создаём текстуру:
   tex := new Texture(WIDTH, HEIGHT);
   // создаём прямоугольник:
   rect := new RectangleShape(new Vector2f(WIDTH, HEIGHT));
```

Прикрепляем текстуру к прямоугольнику:

```
// текстура прямоугольника:
rect.Texture := tex;
```

В игровом цикле рисуем картинку и печатаем текст:

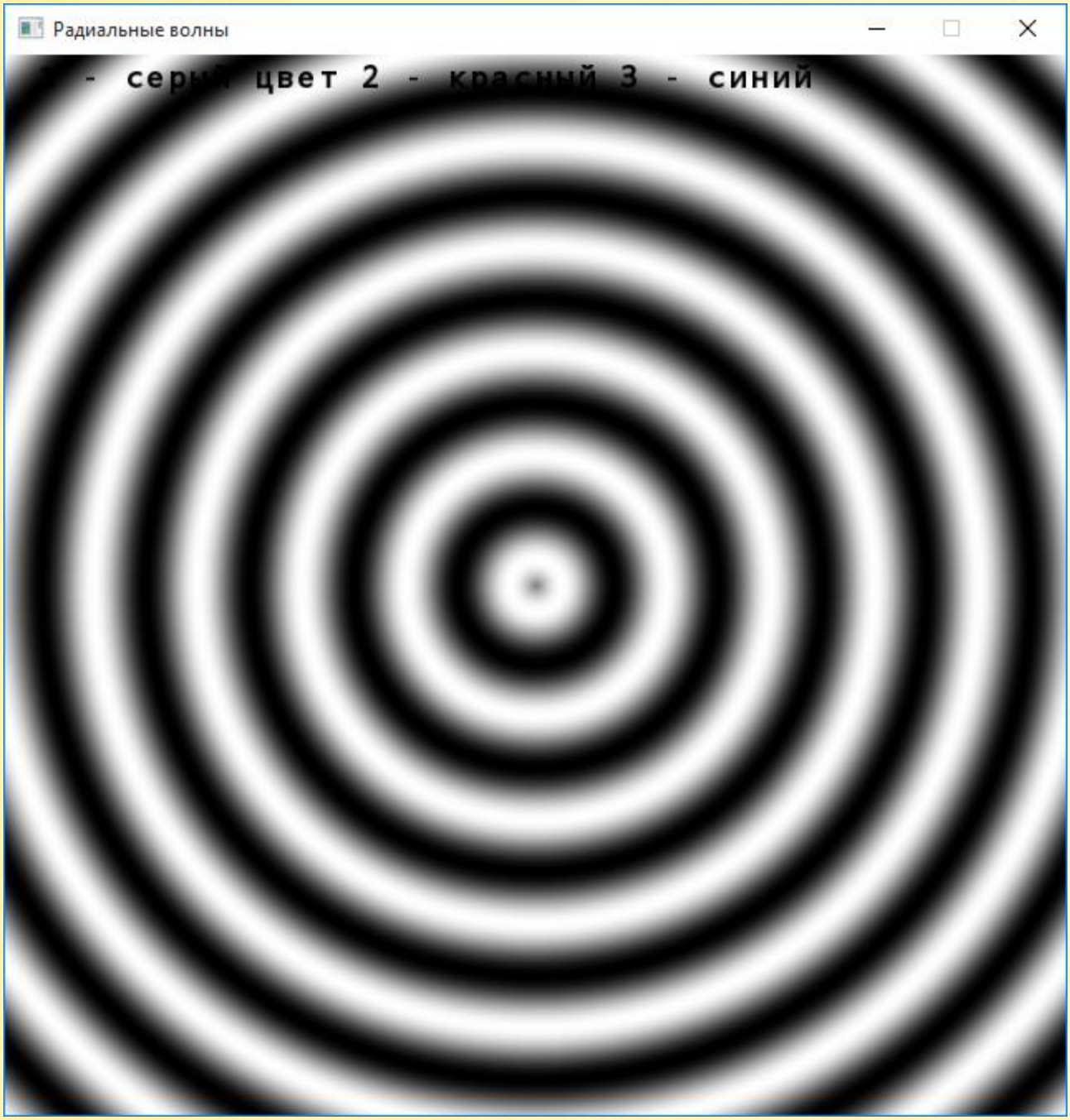
```
// игровой цикл:
while (wind.IsOpen) do
begin
   // вызываем все обработчики событий:
   wind.DispatchEvents();

   // рисуем пиксели:
   Draw();

   // печатаем надпись:
   wind.Draw(txt);

   // показываем на экране:
   wind.Display();
end;
```

Непосредственно рисование картинки происходит в процедуре Draw:

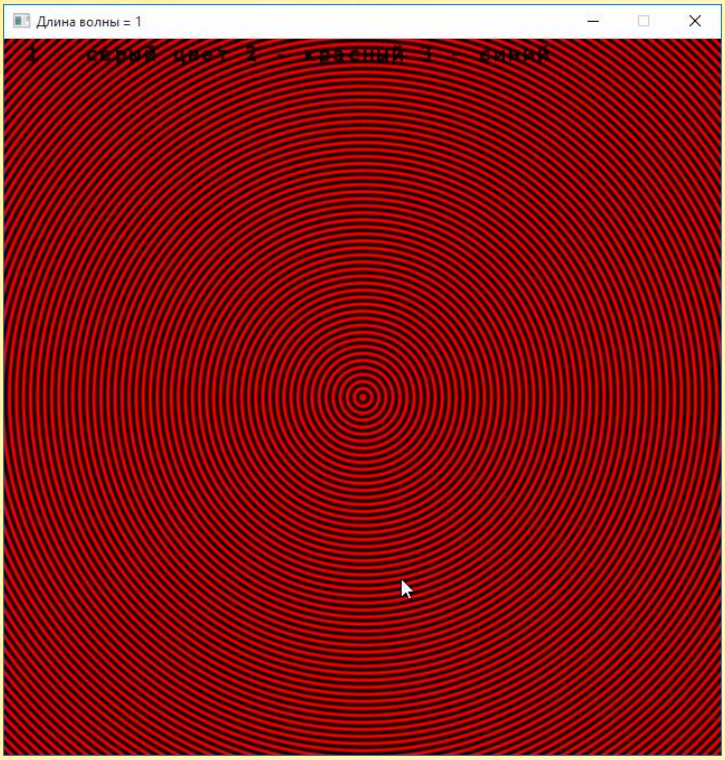
```
// РИСУЕМ ПИКСЕЛИ
procedure Draw();
begin
    // цвет пикселя:
    var clr: Color;

    // координаты центра волн:
    var cx := WIDTH div 2;
```

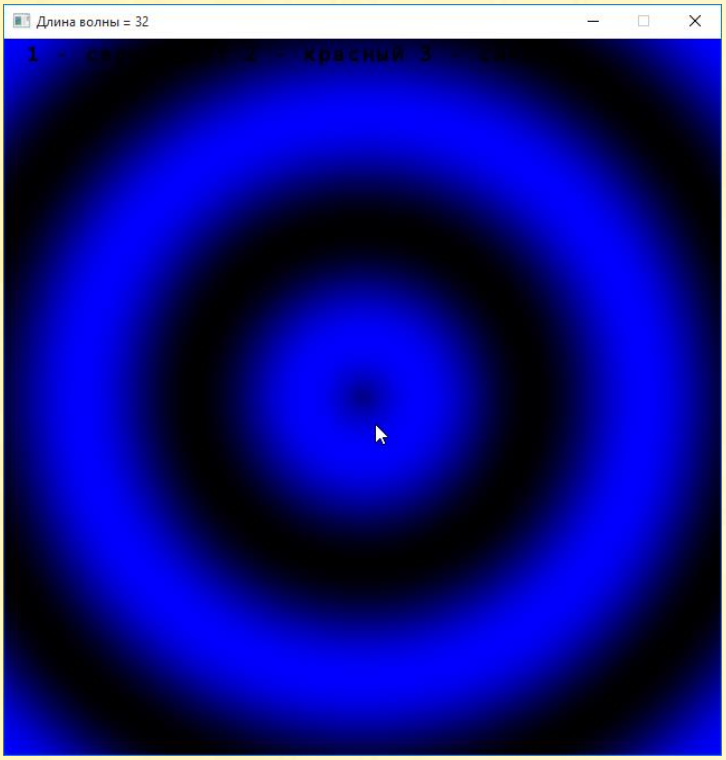
```
    var cy := HEIGHT div 2;

    // рисуем волны:
    for var y := 0 to HEIGHT-1 do
        for var x := 0 to WIDTH-1 do
        begin
           // угол:
           var rad := Sqrt((x - cx) * (x - cx) +
                           (y - cy) * (y - cy)) / wl;
           // цвет очередного пикселя:
           var iclr := Ceil(255 * (1 + Sin(rad)) / 2);

           // серые полоски:
           if (nColor = 1) then
               clr := new Color(iclr, iclr, iclr)
           // красные полоски:
           else if (nColor = 2) then
               clr := new Color(iclr, 0, 0)
           else
               // синие полоски:
               clr := new Color(0, 0, iclr);

            // закрашиваем пиксель:
            img.SetPixel(x, y, clr);
        end;
        // обновляем текстуру:
        tex.Update(img);
        // рисуем текстурированный прямоугольник:
        wind.Draw(rect);
end;
```

Сразу после запуска программы волны имеют серый, невыразительный цвет:



Но их можно перекрасить в красный (цифровая клавиша 2) или в синий цвет (цифровая клавиша 3), а также изменить длину волны, двигая мышку вверх и вниз:





Проект *Радиальные волны А*

Как и в проекте *Синусоидные полоски*, заставим волны двигаться и плескаться!

Сделать это совсем просто:

```
// РИСУЕМ ПИКСЕЛИ
procedure Draw(t : double := 0);
begin
    // цвет пикселя:
    var clr: Color;

    // координаты центра волн:
    var cx := WIDTH div 2;
    var cy := HEIGHT div 2;

    // рисуем волны:
    for var y := 0 to HEIGHT-1 do
```

```
        for var x := 0 to WIDTH-1 do
        begin
           // угол:
           var rad := Sqrt((x - cx) * (x - cx) +
                           (y - cy) * (y - cy)) / wl;
           // цвет очередного пикселя:
           var iclr := Ceil(255 * (1 + Sin(rad + t)) / 2);
```

```
// время:
var t := 0.0;
```

```
// игровой цикл:
while (wind.IsOpen) do
begin
   // вызываем все обработчики событий:
   wind.DispatchEvents();

   // рисуем пиксели:
   Draw(t);
   t += 0.9;
```

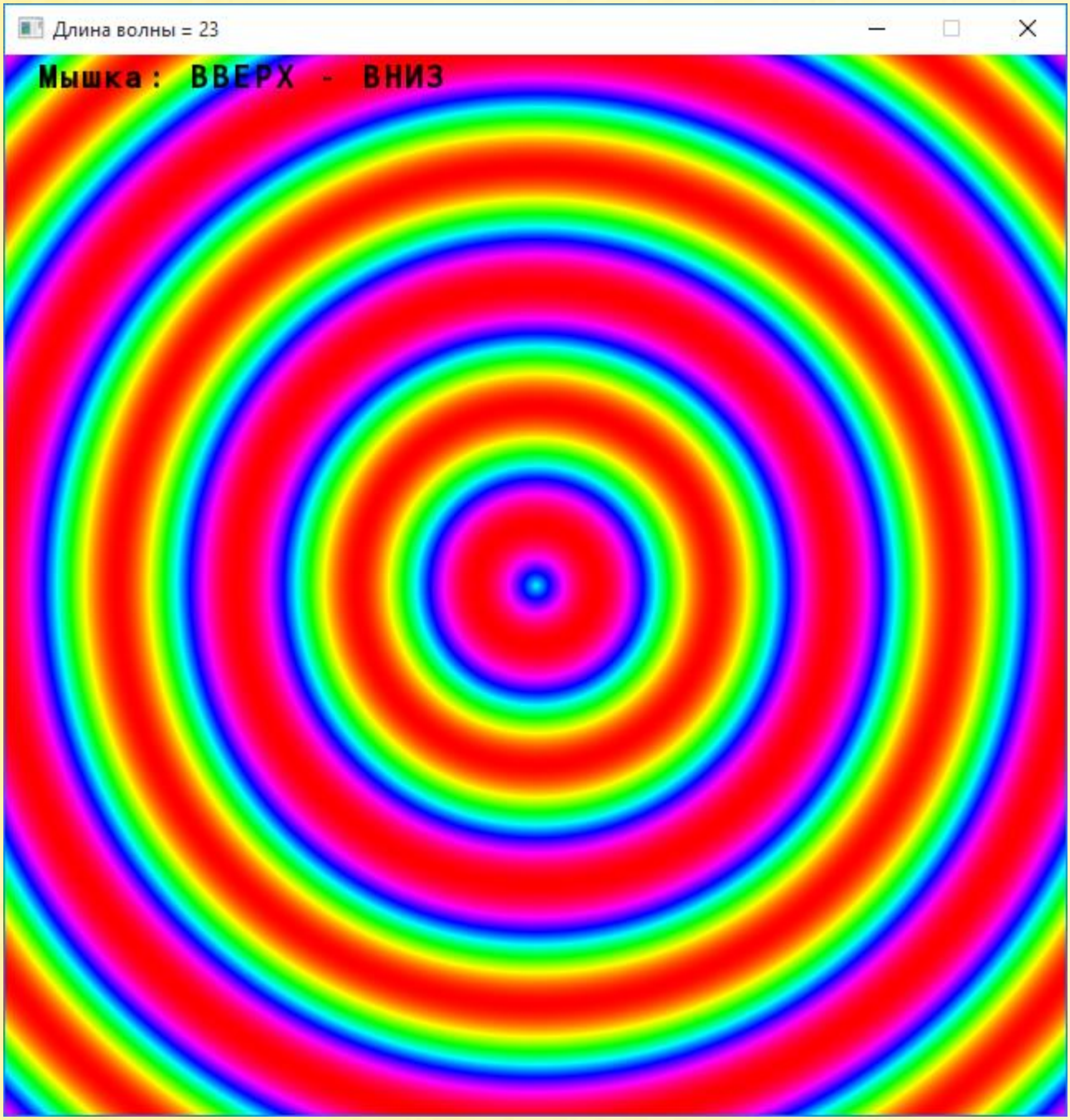
Если время положительное, то волны сходятся к центру окна, если отрицательное – расходятся от центра.

Проект *Радиальные волны 2*

Изменим способ закрашивания пикселей:

```
// РИСУЕМ ПИКСЕЛИ
procedure Draw();
begin
    // цвет пикселя:
    var clr: Color;

    // координаты центра волн:
    var cx := WIDTH div 2;
    var cy := HEIGHT div 2;

    // рисуем волны:
    for var y := 0 to HEIGHT-1 do
        for var x := 0 to WIDTH-1 do
        begin
           // угол:
           var rad := Sqrt((x - cx) * (x - cx) +
                           (y - cy) * (y - cy)) / wl;
           // цвет очередного пикселя:
           var iclr := 360 * (1 + Sin(rad)) / 2;
           clr := HSV2RGB(iclr);
           // закрашиваем пиксель:
           img.SetPixel(x, y, clr);
        end;
        // обновляем текстуру:
        tex.Update(img);
        // рисуем текстурированный прямоугольник:
        wind.Draw(rect);
end;
```

И получим цветные волны:


Длина волны = 23
Мышка: ВВЕРХ - ВНИЗ

Проект *Радиальные волны 2A*

Опять **анимируем** волны:

```
// РИСУЕМ ПИКСЕЛИ
procedure Draw(t : double := 0);
begin
    // цвет пикселя:
    var clr: Color;

    // координаты центра волн:
    var cx := WIDTH div 2;
    var cy := HEIGHT div 2;

    // рисуем волны:
    for var y := 0 to HEIGHT-1 do
        for var x := 0 to WIDTH-1 do
        begin
           // угол:
           var rad := Sqrt((x - cx) * (x - cx) +
                           (y - cy) * (y - cy)) / wl;
           // цвет очередного пикселя:
           var iclr := 360 * (1 + Sin(rad + t)) / 2;
```

```
// время:
var t := 0.0;
```

```
// игровой цикл:
while (wind.IsOpen) do
begin
   // вызываем все обработчики событий:
   wind.DispatchEvents();

   // рисуем пиксели:
   Draw(t);
   t -= 0.9;
```

Проект *Двойная волна*

А теперь давайте пустим **две** волны – **вертикальную** и **горизонтальную**.

По такому случаю в **формулу** для вычисления цвета пикселя добавим ещё один синус:

```
iclr := Ceil(255 * (1 + Sin(x / wx + t) * Sin(y / wy + t)) / 2);
```

Для хранения текущих значений длин волн заведём 2 **глобальные переменные**:

```
// длина волн:
wx: single := 20;
wy: single := 20;
// картинка:
img: Image ;
// текстура:
tex: Texture;
// прямоугольник:
rect: RectangleShape;
```

В процедуре CreateWindow измените поясняющую надпись:

```
// создаём текст:
txt := new Text('Мышка: ВВЕРХ - ВНИЗ | ВЛЕВО - ВПРАВО', fnt, 20);
```

Теперь у нас 2 волны, поэтому мышку нужно двигать и по **горизонтали**, и по **вертикали**.

Процедура Window_MouseMoved также претерпела изменения:

```
// ПЕРЕМЕЩАЕМ МЫШКУ
procedure Window_MouseMoved(sender: Object; e: MouseMoveEventArgs);
begin
```

```
    if (not Mouse.IsButtonPressed(Mouse.Button.Left)) then
        exit;

    // текущие координаты мышки:
    var x := e.X;
    var y := e.Y;

    var dy := y - mousePos.Y;

    if (dy < 0) then
        wY += 1
    else if dy > 0 then wY -= 1;
    if(wY < 1) then
         wY := 1;

    var dx := x - mousePos.X;

    if (dx < 0) then
        wX += 1
    else if dx > 0 then wX -= 1;
    if(wX < 1) then
         wX := 1;
    wind.SetTitle('Длина волны (гориз/верт) = ' + wX + ' ' + wY);
    // запоминаем новые координаты мышки:
    mousePos := new Vector2i(x, y);
end;
```

Волны можно рисовать точно так же, как и в предыдущем проекте, но нам нужно освоить конструктор, который помогает создавать картинку из массива пикселей:

```
clrs := new Color[WIDTH, HEIGHT];
```

В процедуре Draw заполняем его цветными пикселями во вложенных циклах *for*:

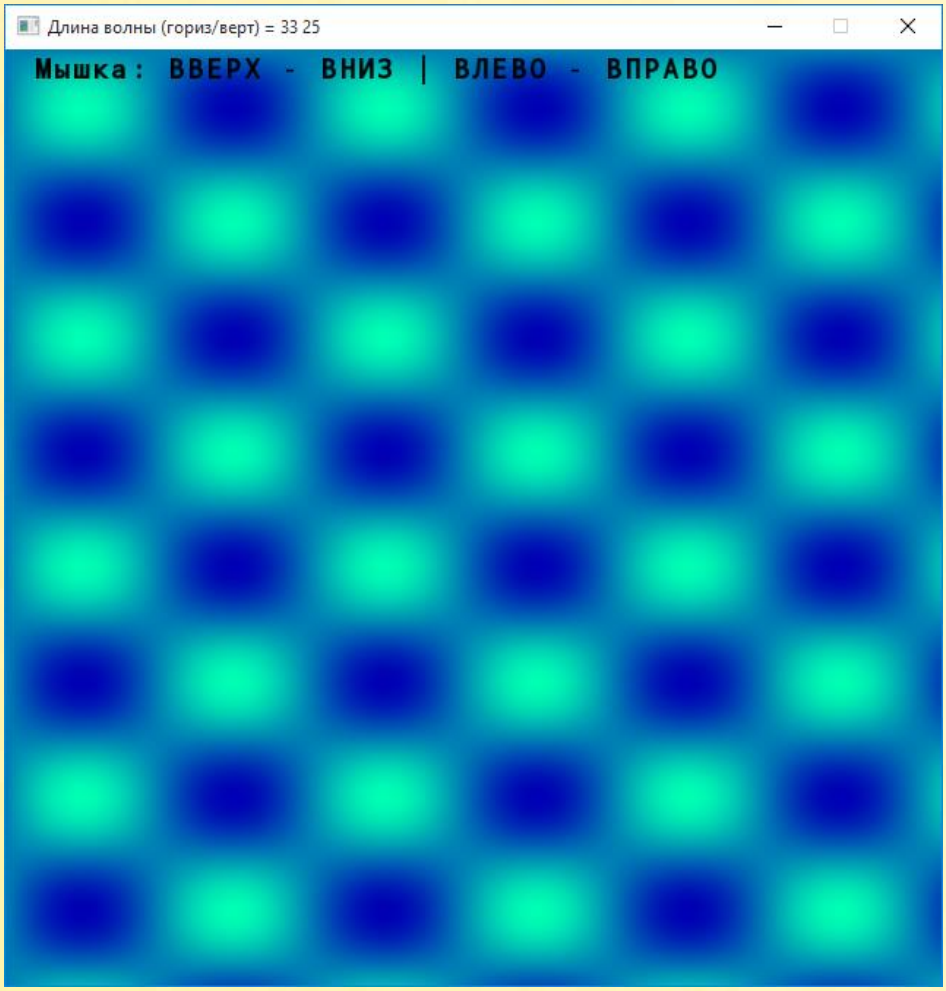
```
// рисуем волны:
for var y := 0 to HEIGHT-1 do
    for var x := 0 to WIDTH-1 do
    begin
        // цвет очередного пикселя:
        var iclr := Ceil(255 * (1 + Sin(x / wx + t) *
```

```
                                        Sin(y / wy + t)) / 2);
            // помещаем в массив:
            clrs[x,y] := new Color(0, iclr, 180);
        end;
```

Создаём новую картинку из массива пикселей:

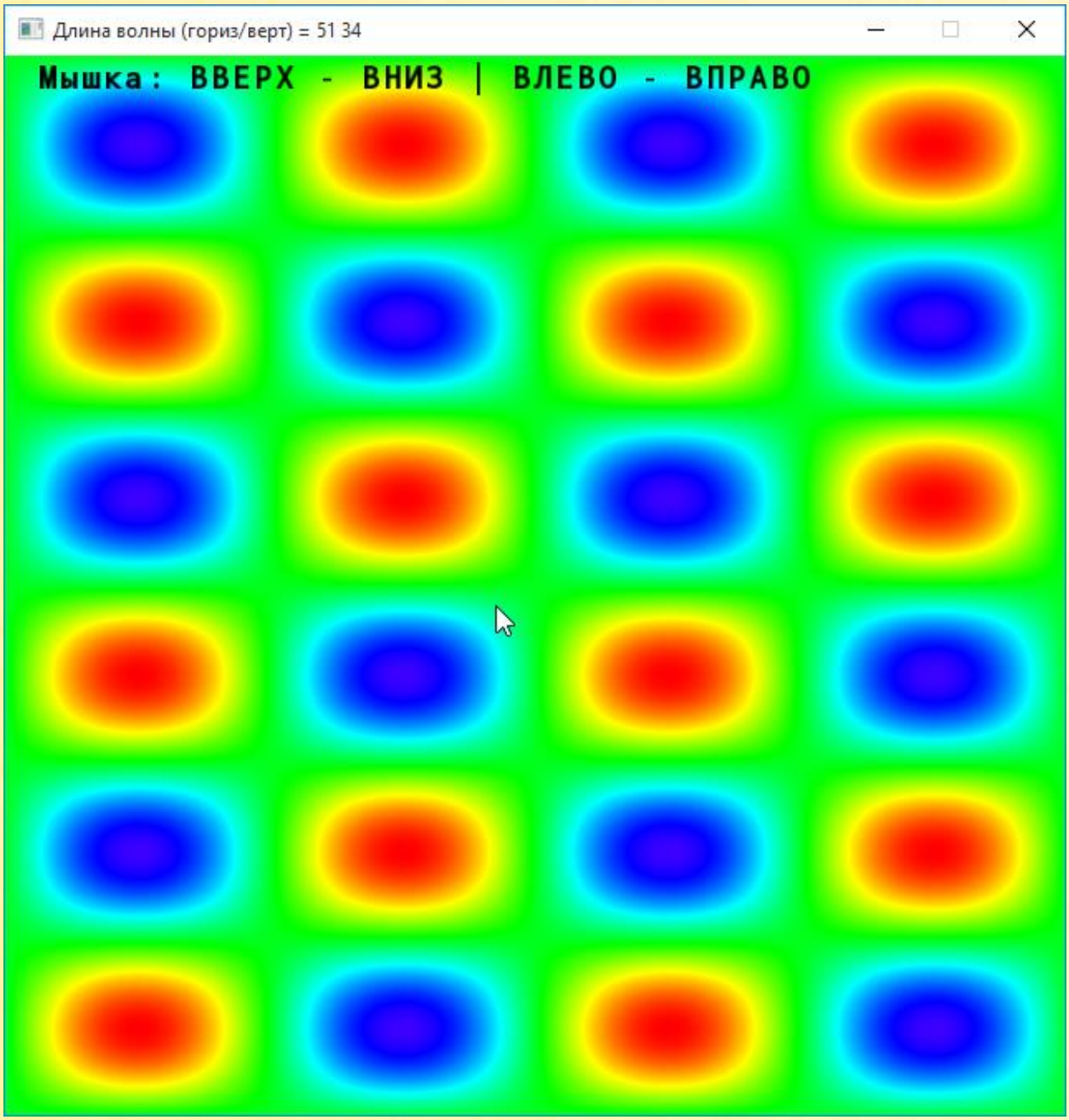
```
        img := new Image(clrs);
        // обновляем текстуру:
        tex.Update(img);
        // рисуем текстурированный прямоугольник:
        wind.Draw(rect);
end;
```

Двигайте мышку и волнуйте поверхность окна приложения:



Картинка станет ещё интереснее, если волны превратить в кружочки:

```
// помещаем в массив:
//clrs[x,y] = new Color(0, iclr, 180);

var clr = HSV2RGB(iclr);
clrs[x,y] = clr;
```



Проект *Лунки*

Этот проект почти в точности повторяет предыдущий, но в процедуре Draw мы будем использовать не массив пикселей, а **массив байтов** – цветных составляющих цвета. Картинка *WIDTH x HEIGHT* содержит в 4 раза больше байтов, чем пикселей, поскольку каждый пиксель задаётся четырьмя составляющими:

```
clrs := new byte[WIDTH * HEIGHT * 4];
```

Во вложенных циклах *for* мы вычисляем значения цветовых составляющих пикселя iclr по формуле с абсолютной величиной:

```
// РИСУЕМ ПИКСЕЛИ
procedure Draw(t : double := 0);
begin
    //рисуем волны:
    for var y := 0 to HEIGHT-1 do
        for var x := 0 to WIDTH-1 do
        begin
           // цвет очередного пикселя:
           var iclr := Ceil(255 * (1 + Abs(Sin(x / wx + t)) *
                                       Abs(Sin(y / wy + t))) / 2);
```

Находим **индекс** первого байта пикселя в массиве clrs:

```
// помещаем в массив:
var id := (y * WIDTH + x)* 4;
```

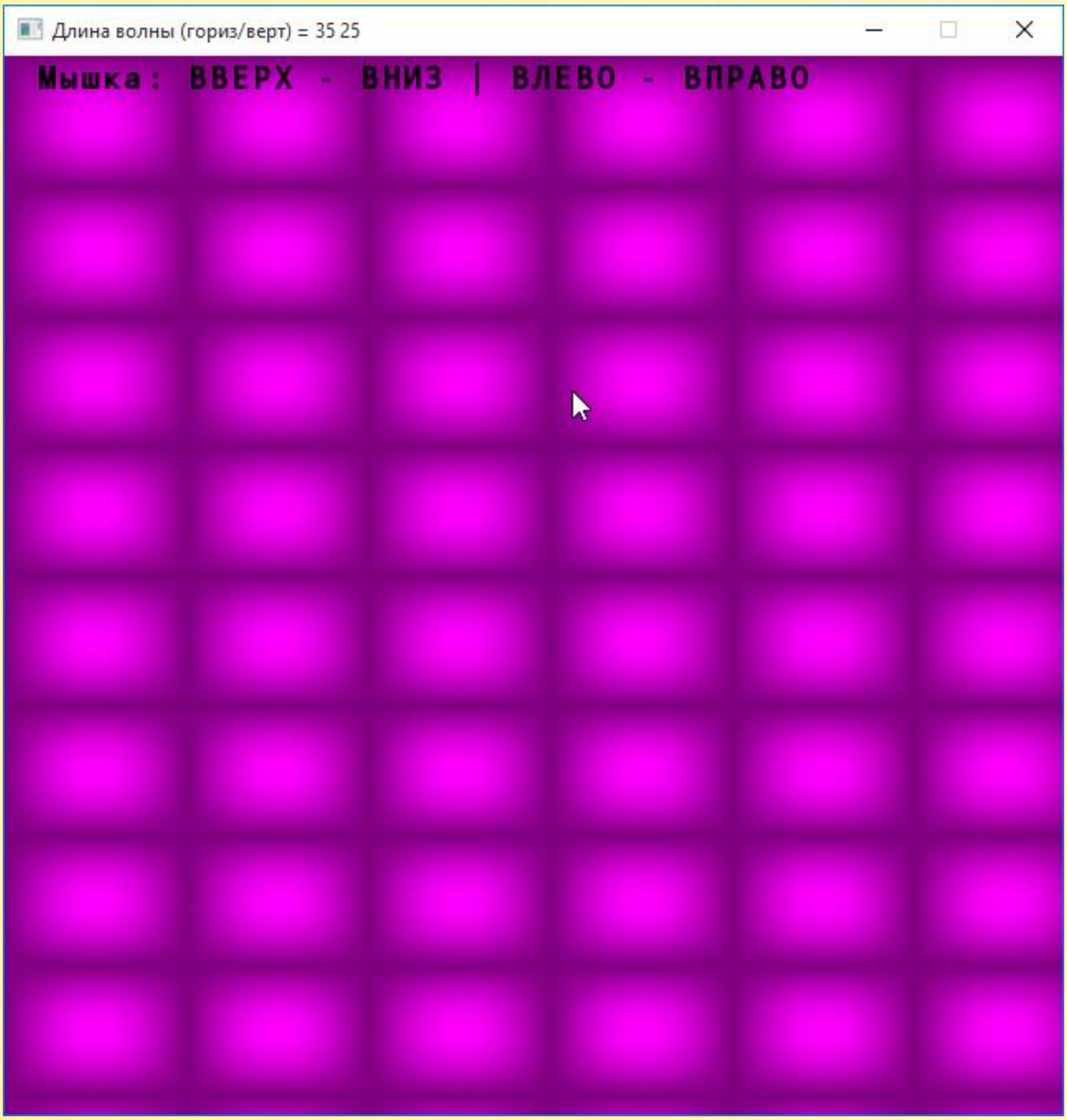
И **записываем** в 4 соседних ячейки памяти байты – цветовые составляющие:

```
           clrs[id] := iclr;     // R
           clrs[id + 1] := 0;    // G
           clrs[id + 2] := iclr; // B
           clrs[id +3] := 255;   // A
        end;
```

Из этого массива создаём новую картинку:

```
    img := new Image(WIDTH, HEIGHT, clrs);
    // обновляем текстуру:
    tex.Update(img);
    // рисуем текстурированный прямоугольник:
    wind.Draw(rect);
end;
```

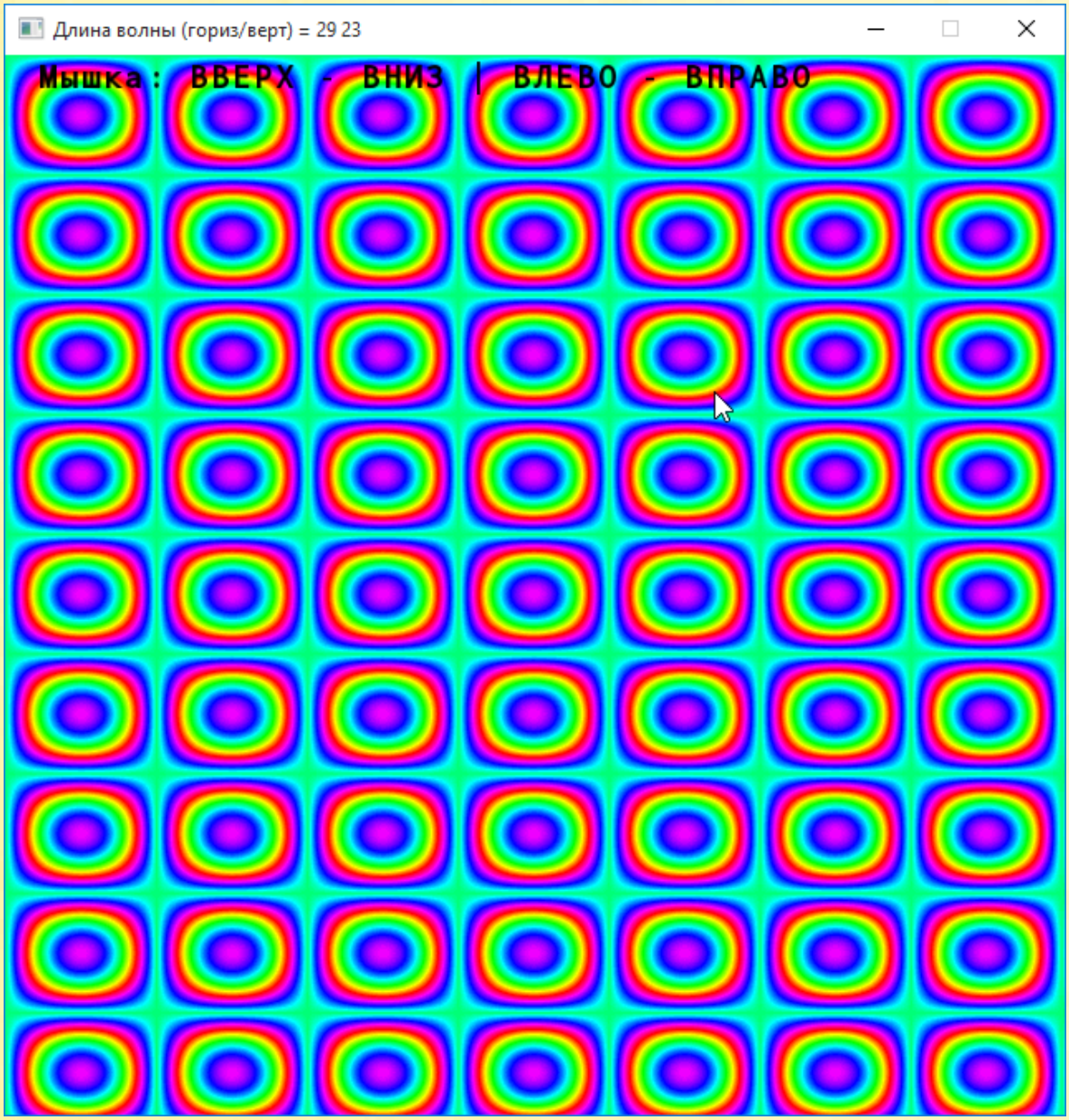
Лунки готовы:



Получаем цвет в режиме HSV:

```
var clr := HsvToRgb(iclr * 4);
clrs[id] := clr.R;
clrs[id+1] := clr.G;
clrs[id+2] := clr.B;
clrs[id+3] := 255;
```

И лунки окрашиваются во все цвета радуги:



Класс Sprite (Спрайты)

Класс Sprite описывает *спрайты* – основные объекты компьютерных игр.

Спрайт - это текстурированный прямоугольник. Текстура дополнительно может быть тонирована любым цветом. Размеры спрайта определяются текстурой или её частью.

Дальше мы рассмотрим применение спрайтов в программах.

Класс Shader (Шейдеры)

Шейдер – это небольшая программа, написанная на языке *GLSL* (*OpenGL Shading Language*). Шейдеры служат для добавления различных визуальных эффектов к изображению.

Проект *Класс Shader*

В этом проекте мы воспользуемся пиксельным шейдером из демонстрационной программы.

Картинка будет большая, поэтому размеры окна должны ей соответствовать:

```
const
   // размеры окна:
   WIDTH = 800;
   HEIGHT = 533;
```

В главном блоке создаём окно:

```
begin
   // создаём главное окно:
   CreateWindow(WIDTH, HEIGHT, 'Класс Shader');
```

И проверяем, поддерживает ли видеокарта шейдеры:

```
if (not Shader.IsAvailable) then
begin
   wind.SetTitle('ШЕЙДЕРЫ НЕ ПОДДЕРЖИВАЮТСЯ!');
end;
```

На всех новых компьютерах это так, но порядок есть порядок.

Загружаем шейдер из папки ресурсов:

```
// загружаем шейдер:
var _shader := new Shader(nil, 'Media/pixelate.frag');
```

Для применения шейдера необходим текстурированный графический объект. Например, спрайт:

```
// загружаем текстуру:
var texture := new Texture('Media/тюльпаны.jpg');
// создаём спрайт из текстуры:
var sprite := new Sprite(texture);
```

Шейдер будет применяться к спрайту, поэтому его текущей текстурой будет текстура спрайта:

```
_shader.SetParameter('texture', Shader.CurrentTexture);
```

Можно более прозрачно указать, какая именно текстура будет использоваться шейдером:

```
_shader.SetParameter('texture', sprite.Texture);
```

При визулизации графических объектов можно указывать различные **режимы**. В данном случае – **шейдерный**:

```
// создаём шейдерный режим:
var states := new RenderStates(_shader);
```

Запускаем **игровой цикл**:

```
// игровой цикл:
while (wind.IsOpen) do
begin
   // вызываем все обработчики событий:
   wind.DispatchEvents();
```

Мы хотим изменять степень пикселизации в работающнй программе с помощью **мышки**, поэтому внутри игрового цикла необходимо обновлять значение параметра **pixel_threshold**:

```
// обновляем параметр pixel_threshold:
_shader.SetParameter('pixel_threshold', threshold);
```

Чтобы применить шейдер к текстуре спрайта, указываем режим визуализации в методе окна **Draw**:

```
// печатаем спрайт с шейдером:
wind.Draw(sprite, states);
```

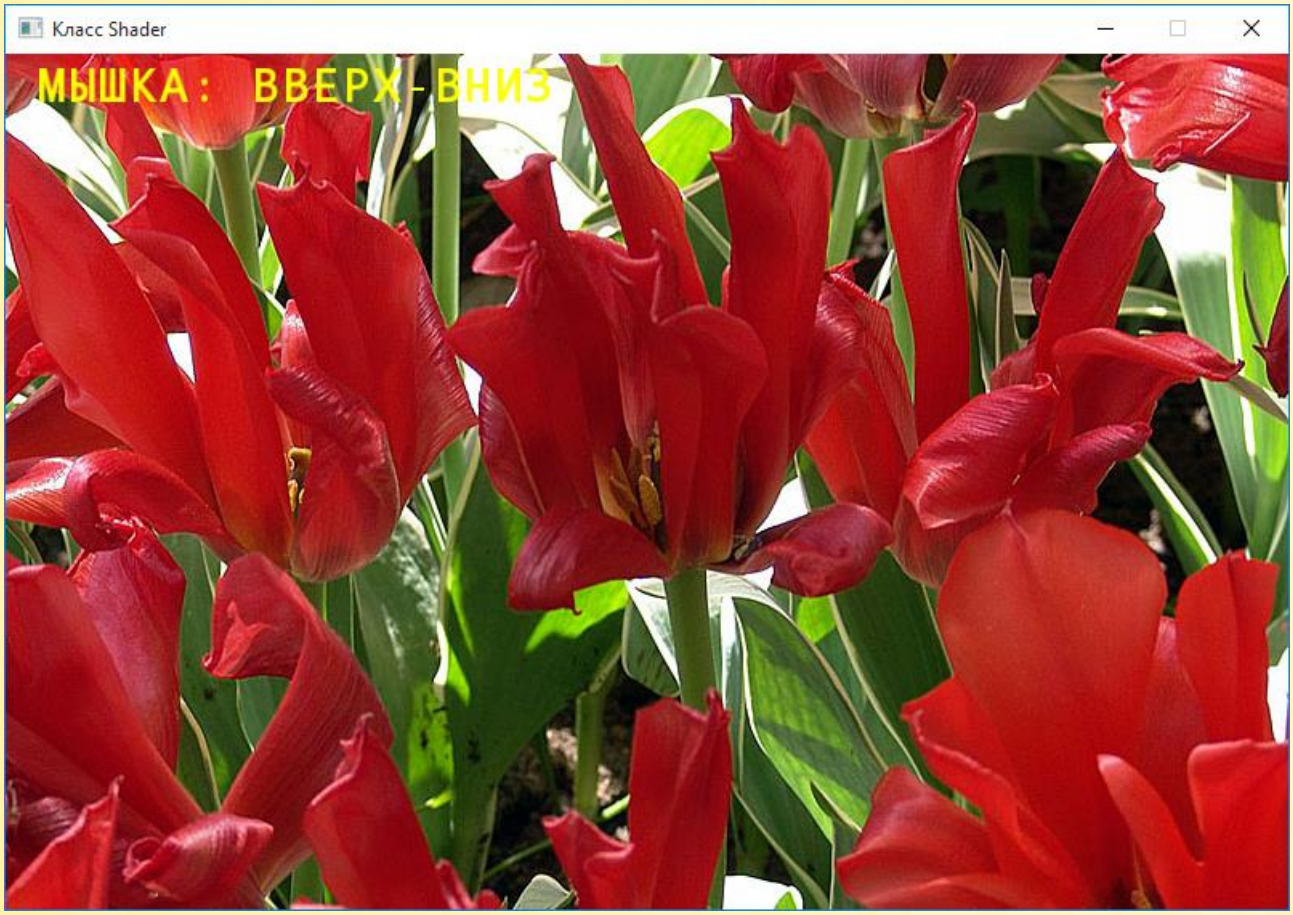
Поверх картинки печатаем поясняющий **текст**:

```
   // печатаем надпись:
   wind.Draw(txt);

   // показываем на экране:
   wind.Display();
end;
```

Начальное значение глобальной переменной threshold равно нулю:

```
var
   // окно приложения:
   wind: RenderWindow := nil;
   rand := new Random();
   // шрифт:
   fnt: Font := nil;
   // текст:
   txt: Text := nil;
   // координаты мышки:
   mousePos: Vector2i;

   threshold := 0.0;
```

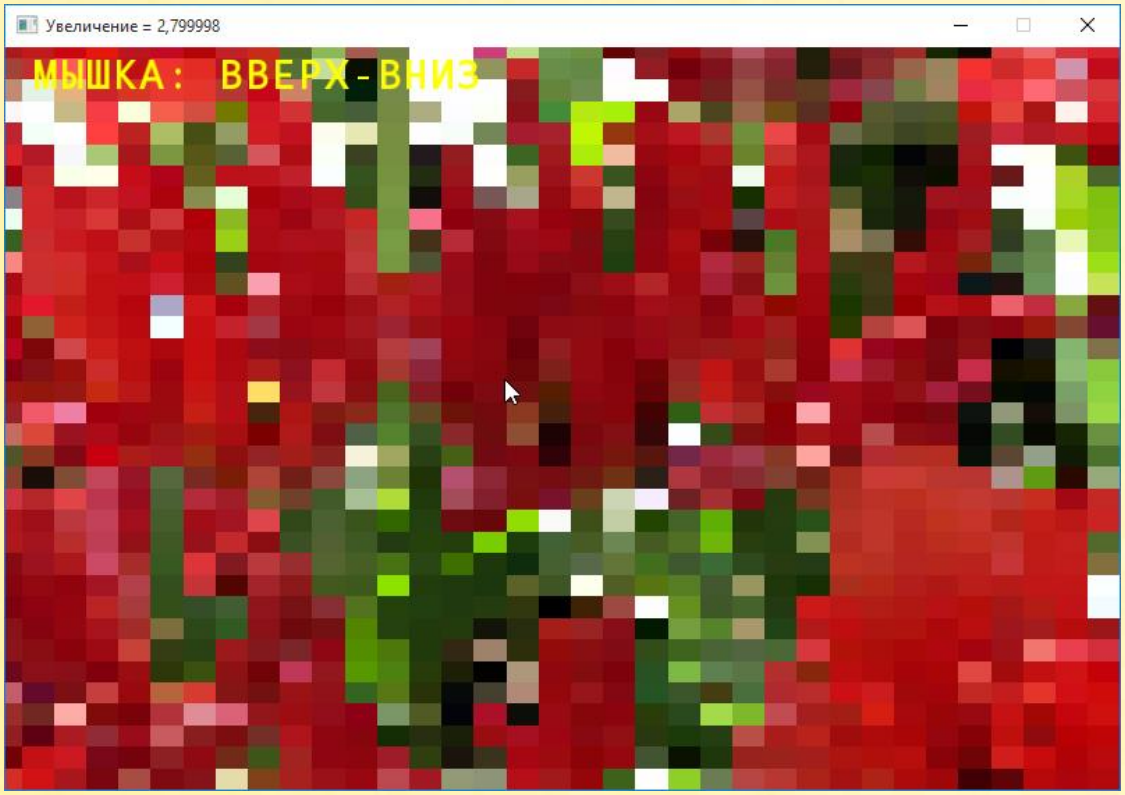
Поэтому картинка (текстура спрайта) выводится на экран без изменений:



Но при **перемещении мышки** с нажатой кнопкой значение этой переменной изменяется:

```
// ПЕРЕМЕЩАЕМ МЫШКУ
procedure Window_MouseMoved(sender: Object; e: MouseMoveEventArgs);
begin
   if (not Mouse.IsButtonPressed(Mouse.Button.Left)) then
       exit;
   // текущие координаты мышки:
   var x := e.X;
   var y := e.Y;
   var dy := y - mousePos.Y;
   if (dy < 0) then
       threshold += 0.0005
   else threshold -= 0.0005;
   wind.SetTitle('Увеличение = ' + Abs(threshold)*100);
   // запоминаем новые координаты мышки:
   mousePos := new Vector2i(x, y);
end;
```

Что приводит к **пикселизации** изображения шейдером:



Такую картинку уже трудно узнать!

Обратите внимание, что пропорции пикселей такие же, как у картинки, поэтому только для квадратных картинок шейдер нарисует квадратные пиксели.

Проект *Шейдеры*

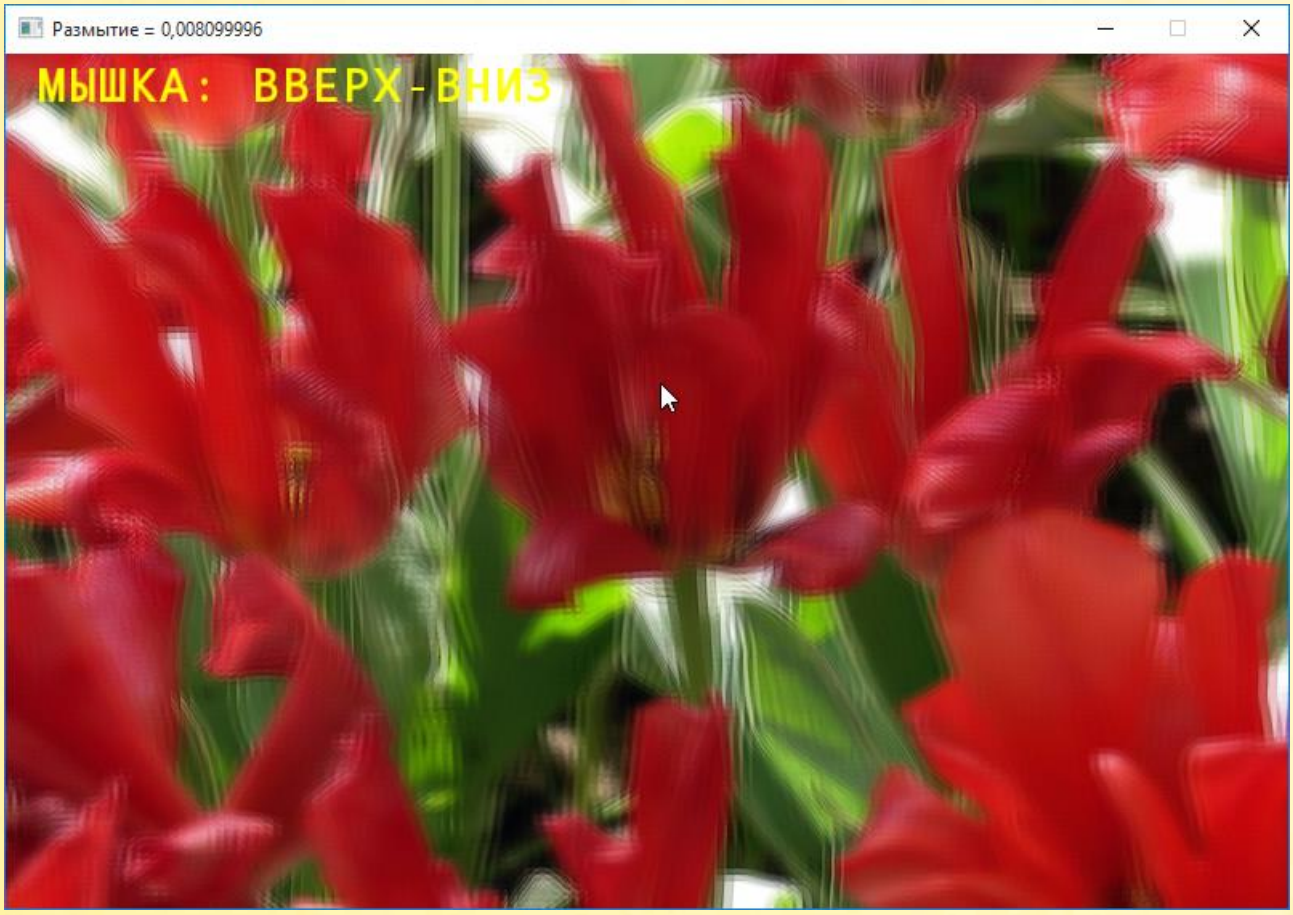
Очень часто нужно **размыть** изображение, сделать его нерезким. Для этого у нас имеется шейдер *blur.frag*. Применим его к картинке с тюльпанами:

```
blur_radius := 0.0;
```

```
begin
   // создаём главное окно:
   CreateWindow(WIDTH, HEIGHT, 'Шейдеры');

   if (not Shader.IsAvailable) then
   begin
      wind.SetTitle('ШЕЙДЕРЫ НЕ ПОДДЕРЖИВАЮТСЯ!');
   end;
   // загружаем шейдер:
   var _shader := new Shader(nil, 'Resources/blur.frag');

   // загружаем текстуру:
   var texture := new Texture('Media/тюльпаны.jpg');
   // создаём спрайт из текстуры:
   var sprite := new Sprite(texture);

   _shader.SetParameter('texture', Shader.CurrentTexture);
   //_shader.SetParameter('texture', sprite.Texture);

   // создаём шейдерный режим:
   var states := new RenderStates(_shader);

   // игровой цикл:
   while (wind.IsOpen) do
   begin
      // вызываем все обработчики событий:
```

```
        wind.DispatchEvents();

        // обновляем параметр blur_radius:
        _shader.SetParameter('blur_radius', blur_radius);

        // печатаем спрайт с шейдером:
        wind.Draw(sprite, states);

        // печатаем надпись:
        wind.Draw(txt);

        // показываем на экране:
        wind.Display();
    end;

end.
```

Шейдер, конечно, не лучшего качества, но и он может быть полезен:



Проект *Шейдеры 2*

В *Интернете* вы без труда найдёте множество шейдеров для своих программ. В этом проекте мы познакомимся с двумя шейдерами Виктора Корсуна с сайта *glslsandbox.com*.

Для их работы нужна пустая **текстура** без картинки, потому что они сами заполняют текстуру цветными пикселями, которые рассчитываются по весьма замысловатым формулам.

В **главном блоке** загружаем шейдер из файла:

```
begin
   // создаём главное окно:
   CreateWindow(WIDTH, HEIGHT, 'Шейдеры 2');

   if (not Shader.IsAvailable) then
   begin
      wind.SetTitle('ШЕЙДЕРЫ НЕ ПОДДЕРЖИВАЮТСЯ!');
   end;
   // загружаем шейдер:
   var _shader := new Shader(nil, 'Media/sandbox.frag');
```

Создаём **текстуру** по размерам окна:

```
// создаём текстуру:
var texture := new Texture(WIDTH, HEIGHT);
```

Передаём её **спрайту**, чтобы напечать в окне:

```
// создаём спрайт из текстуры:
var sprite := new Sprite(texture);
```

Шейдеры обычно имеют **параметры**, которые записываются в начале программы. У шейдера *sandbox.frag* 2 параметра – **разрешение** и **время**:

```
uniform vec2 resolution;
uniform float time;

void main(void)
{
    float x = gl_FragCoord.x;
    float y = gl_FragCoord.y;
    float mov0 = x + y + cos(sin(time) * 2.) * 100.+ sin(x / 100.) * 1000.;
    float mov1 = y / resolution.y / 0.2 + time;
    float mov2 = x / resolution.x / 0.2;
    float c1 = abs(cos(mov1 + time) / 2.+ mov2 / 2.- mov1 - mov2 + time);
    float c2 = abs(sin(c1 + sin(mov0 / 1000.+ time) +
               tan(y / 40.+ time) + tan((x + y) / 100.) * 3.));
    float c3 = abs(tan(c2 + cos(mov1 + mov2 + c2) - tan(mov2) - cos(x / 1000.)));
    gl_FragColor = vec4(c1, c2, c3, 1.0);
}
```

Первый – resolution – это размеры текстуры, поэтому нужно передать этому параметру вектор с размерами текстуры:

```
// устанавливаем значение переменной resolution:
_shader.SetParameter('resolution', new Vector2f(WIDTH, HEIGHT));
```

При печати спрайта необходимо использовать режим с шейдером:

```
// создаём шейдерный режим:
var states := new RenderStates(_shader);
```

И для изменения картинки в игровом цикле создаём переменную time:

```
// условное время для игрового цикла:
var time := 0.0;
```

А тут как раз и начался игровой цикл:

```
// игровой цикл:
while (wind.IsOpen) do
begin
```

```
    // вызываем все обработчики событий:
    wind.DispatchEvents();
```

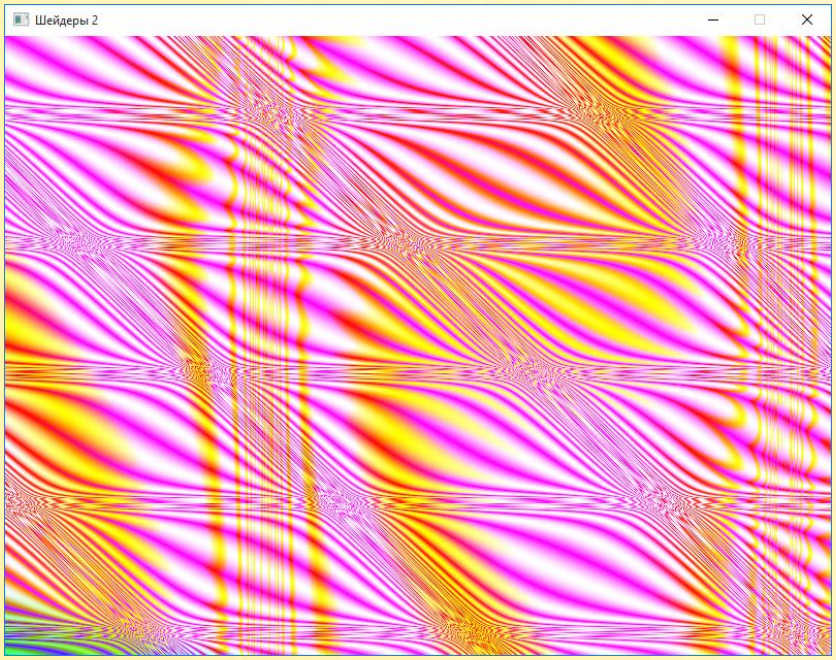
На каждой итерации изменяем значение параметра time:

```
// обновляем параметр time:
_shader.SetParameter('time', time);
```

И печатаем спрайт с шейдером:

```
      // печатаем спрайт с шейдером:
      wind.Draw(sprite, states);

      // увеличиваем время:
      time += 0.1;

      // показываем на экране:
      wind.Display();
   end;
end.
```

И вот такая у нас получилась картинка:



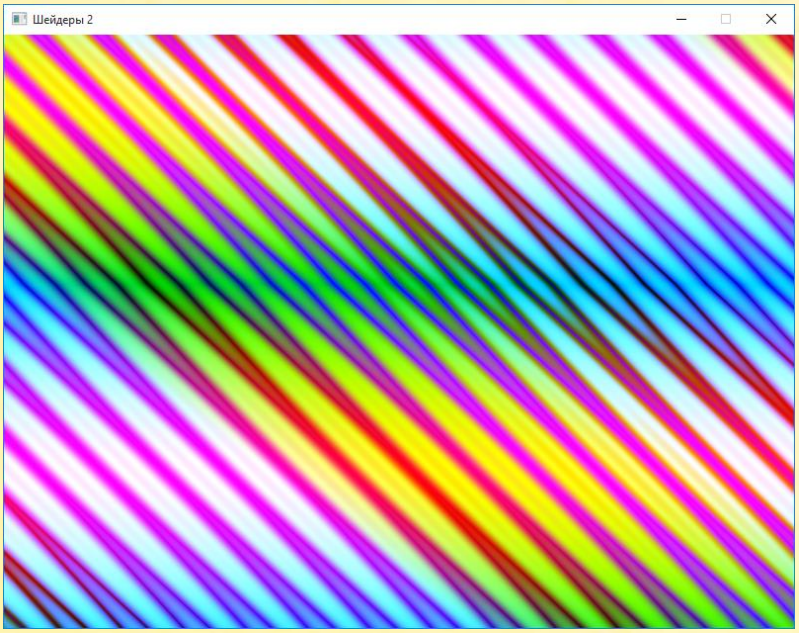
На экране волны двигаются, но в книге движение не передашь.

К сожалению, на картинке много лишних линий, которые только портят её. Давайте **упростим** шейдер:

```
var _shader := new Shader(nil, 'Media/sandbox1.frag');
```

```
uniform vec2 resolution;
uniform float time;

void main(void)
{
    float x = gl_FragCoord.x;
    float y = gl_FragCoord.y;
    float mov0 = x + y + cos(sin(time) * 2.);
    float mov1 = y / resolution.y / 0.2 + time;
    float mov2 = x / resolution.x / 0.2;
    float c1 = abs(cos(mov1 + time) / 1.);
    float c2 = abs(sin(c1 + sin(mov0 / 10.+ time)));
    float c3 = abs(tan(c2 + cos(mov1 + mov2 + c2)));
    gl_FragColor = vec4(c1, c2, c3, 1.0);
}
```

Волны получились не столь изящными, как в оригинале, но зато без всяких посторонних шумов:



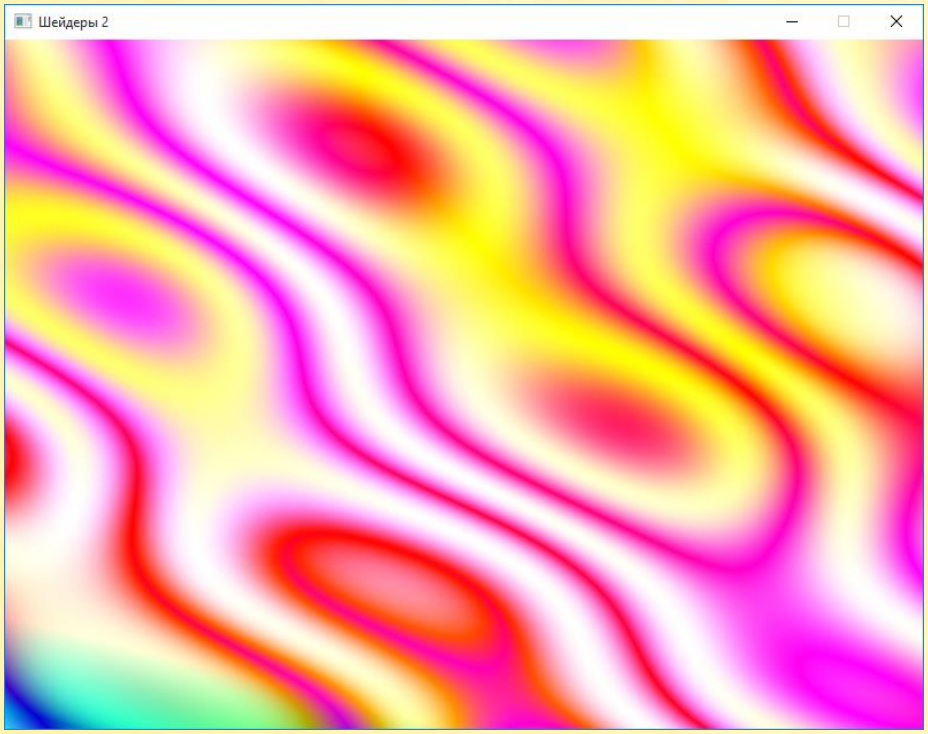
Продолжайте эксперименты, и у вас наверняка получатся новые красивые шейдеры.

А мы переходим ко **второму** шейдеру Виктора Корсуна. Он имеет те же самые параметры, что и первый, но менее «навороченный»:

```
var _shader := new Shader(nil, 'Media/sandboxA.frag');
```

```
uniform vec2 resolution;
uniform float time;

void main(void)
{
    float x = gl_FragCoord.x;
    float y = gl_FragCoord.y;
    float mov0 = x + y + cos(sin(time) * 2.) * 100.+ sin(x / 100.) * 1000.;
    float mov1 = y / resolution.y / 0.2 + time;
    float mov2 = x / resolution.x / 0.2;
    float c1 = abs(sin(mov1 + time) / 2.+ mov2 / 2.- mov1 - mov2 + time);
    float c2 = abs(sin(c1 + sin(mov0 / 1000.+ time) + sin(y / 40.+ time) + sin((x + y) / 100.) * 3.));
    float c3 = abs(sin(c2 + cos(mov1 + mov2 + c2) + cos(mov2) + sin(x / 1000.)));
    gl_FragColor = vec4(c1, c2, c3, 1.0);
}
```

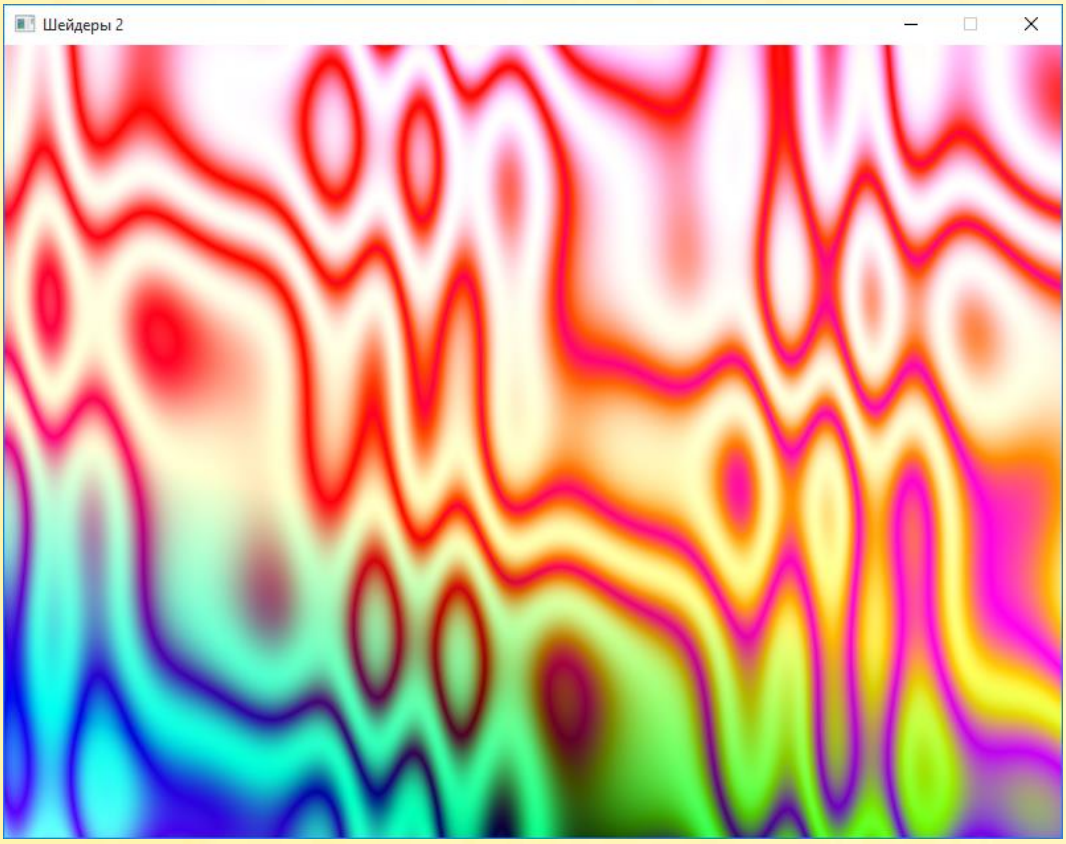
Здесь уже посторонних линий нет, так что плазма получилась на заглядение:



Но при желании вы и здесь можете поиграть с параметрами:

```
var _shader := new Shader(nil, 'Media/sandboxA1.frag');
```

```
uniform vec2 resolution;
uniform float time;

void main(void)
{
    float x = gl_FragCoord.x;
    float y = gl_FragCoord.y;
    float mov0 = x + y + cos(sin(time) * 2.) * 100.+ sin(x / 100.) * 10000.;
    float mov1 = y / resolution.y / 0.5 + time;
    float mov2 = x / resolution.x / 0.5;
    float c1 = abs(sin(mov1 + time) / 2.+ mov2 / 2.- mov1 - mov2 + time);
    float c2 = abs(sin(c1 + sin(mov0 / 1000.+ time) + sin(y / 40.+ time) +
                           sin((x + y) / 100.) * 3.));
    float c3 = abs(sin(c2 + cos(mov1 + mov2 + c2) + cos(mov2) + sin(x / 10000.)));
    gl_FragColor = vec4(c1, c2, c3, 1.0);
}
```



Проект *Шейдеры 3*

Шейдер не позволяет масштабировать картинку, поэтому в этом проекте мы будем увеличивать изображение самостоятельно.

Введём константу ZOOM для кратности увеличения изображения:

```
// увеличение:
ZOOM = 2.1;
```

Для печати картинки нам потребуется прямоугольник по размерам окна:

```
// прямоугольник для шейдера:
var rect := new RectangleShape(new Vector2f(WIDTH, HEIGHT));
```

Но рисовать его мы будем не в окне, а на промежуточной текстуре rtex:

```
// вспомогательная текстура:
var w := Ceil(WIDTH / ZOOM);
var h := Ceil(HEIGHT / ZOOM);
var rtex := new RenderTexture(w, h);
rtex.Smooth := true;
```

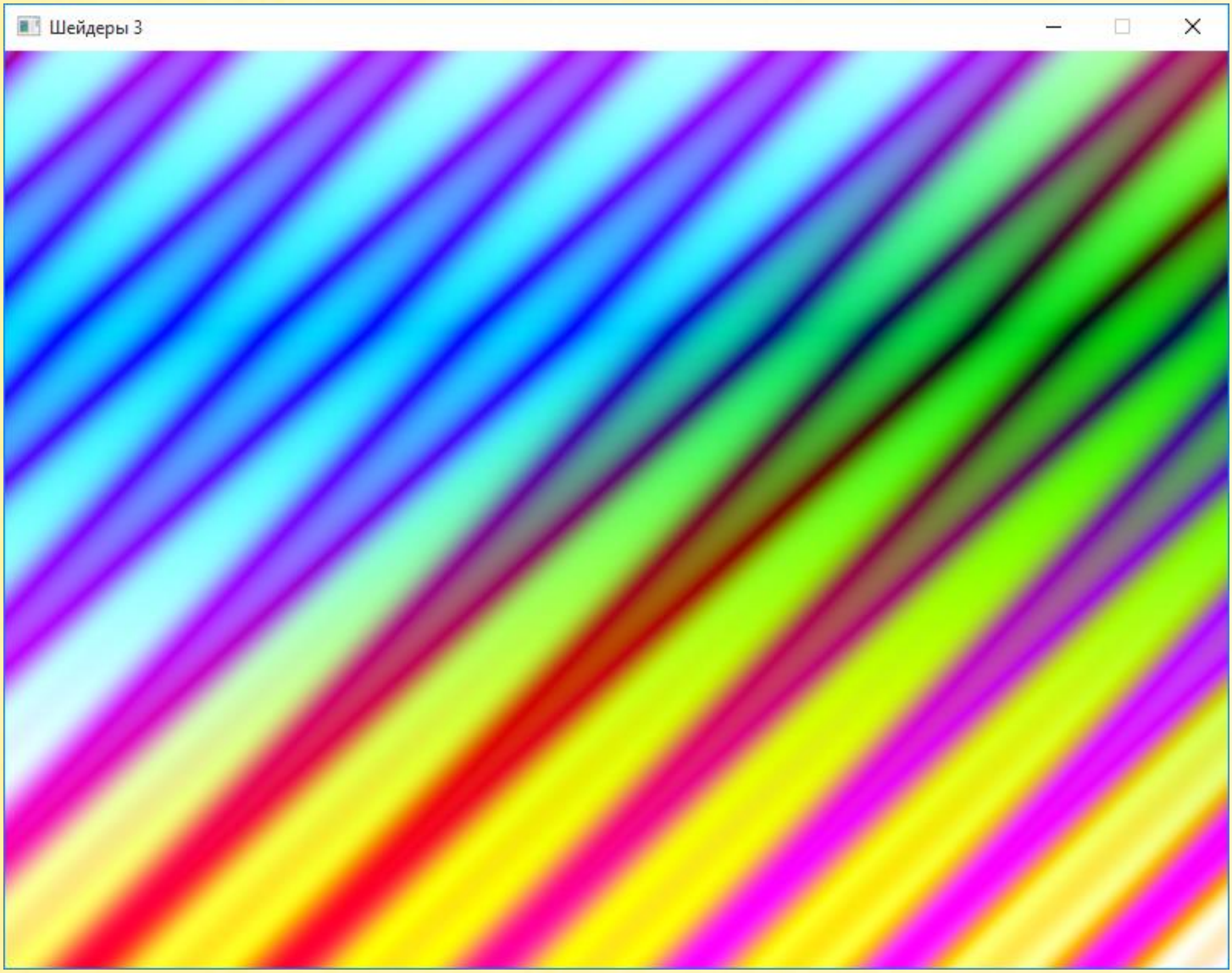
В игровом цикле рисуем шейдерную картинку из прямоугольника rect на текстуре rtex:

```
// рисуем картинку на вспомогательной текстуре:
rtex.Draw(rect, states);
```

Затем растягиваем текстуру до размеров прямоугольника rect, который и показываем на экране:

```
// печатаем её на экране:
rect.Texture := rtex.Texture;
wind.Draw(rect);
```

Запускаем программу – диагонали заметно **утолстились**:



Проект *Шейдеры 3.1*

Более гибкий способ масштабирования изображения заключается в вырезании прямоугольника из шейдерной текстуры.

Тогда можно изменять увеличение по горизонтали и по вертикали отдельно:

```
// увеличение по осям:
zoomX := 1.0;
zoomY := 1.0;
```

Прямоугольнику rect мы будем передавать прямоугольник intrect из текстуры:

```
// прямоугольник для шейдера:
rect: RectangleShape;
// часть текстуры:
_intrect: IntRect;
```

В игровом цикле мы передаём заданный прямоугольник *intrect* свойству TextureRect:

```
// игровой цикл:
while (wind.IsOpen) do
begin
   // вызываем все обработчики событий:
   wind.DispatchEvents();

   // обновляем параметр time:
   _shader.SetParameter('time', time);

   // рисуем картинку на вспомогательной текстуре:
   rtex.Draw(rect, states);

   // печатаем её на экране:
   rect.Texture := rtex.Texture;
   rect.TextureRect := _intrect;
   wind.Draw(rect);
```

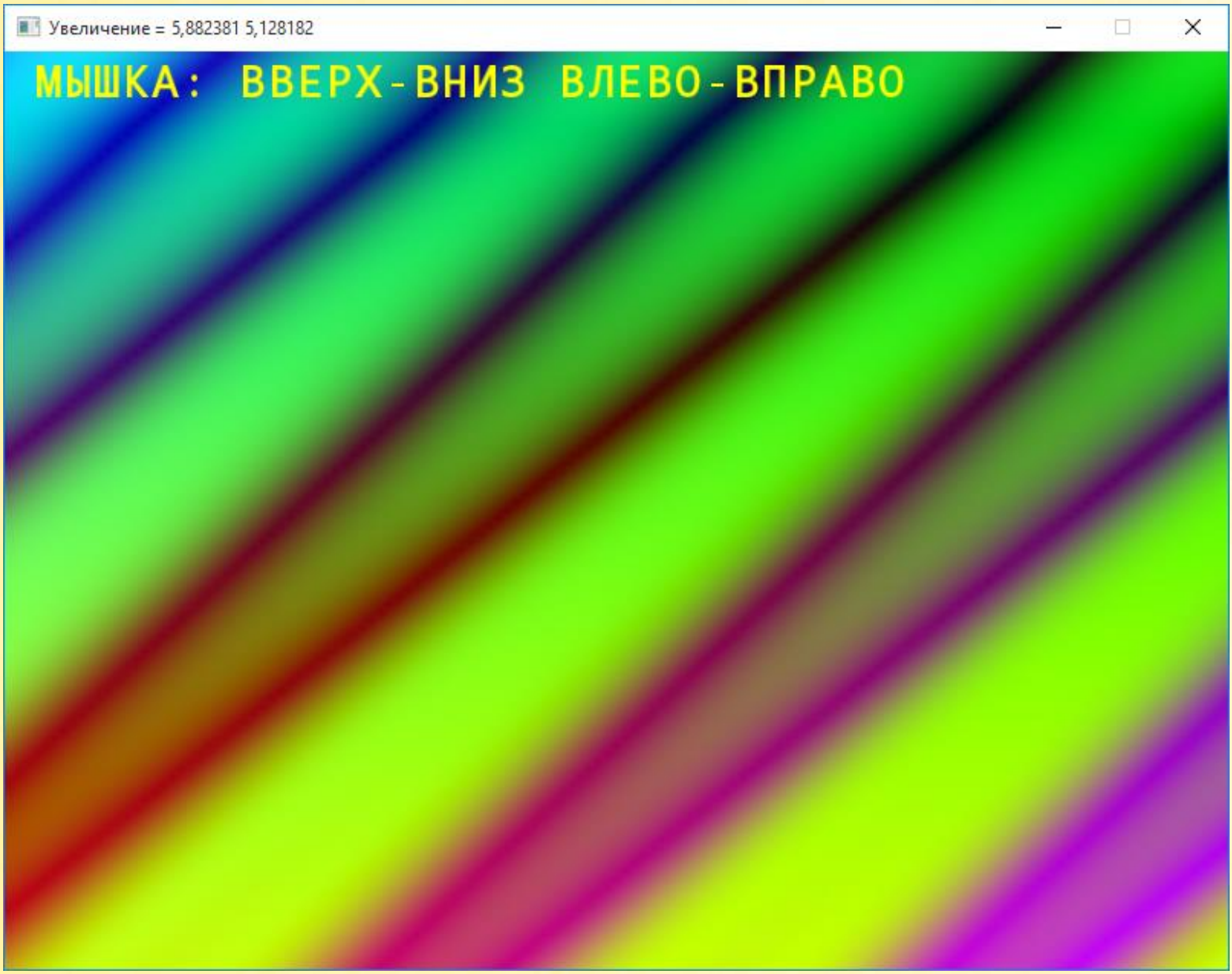
```
    // увеличиваем время:
    time += 0.05;

    // печатаем надпись:
    wind.Draw(txt);

    // показываем на экране:
    wind.Display();
end;
```

А размеры прямоугольника _intrect можно изменять в работающей программе с помощью мышки:

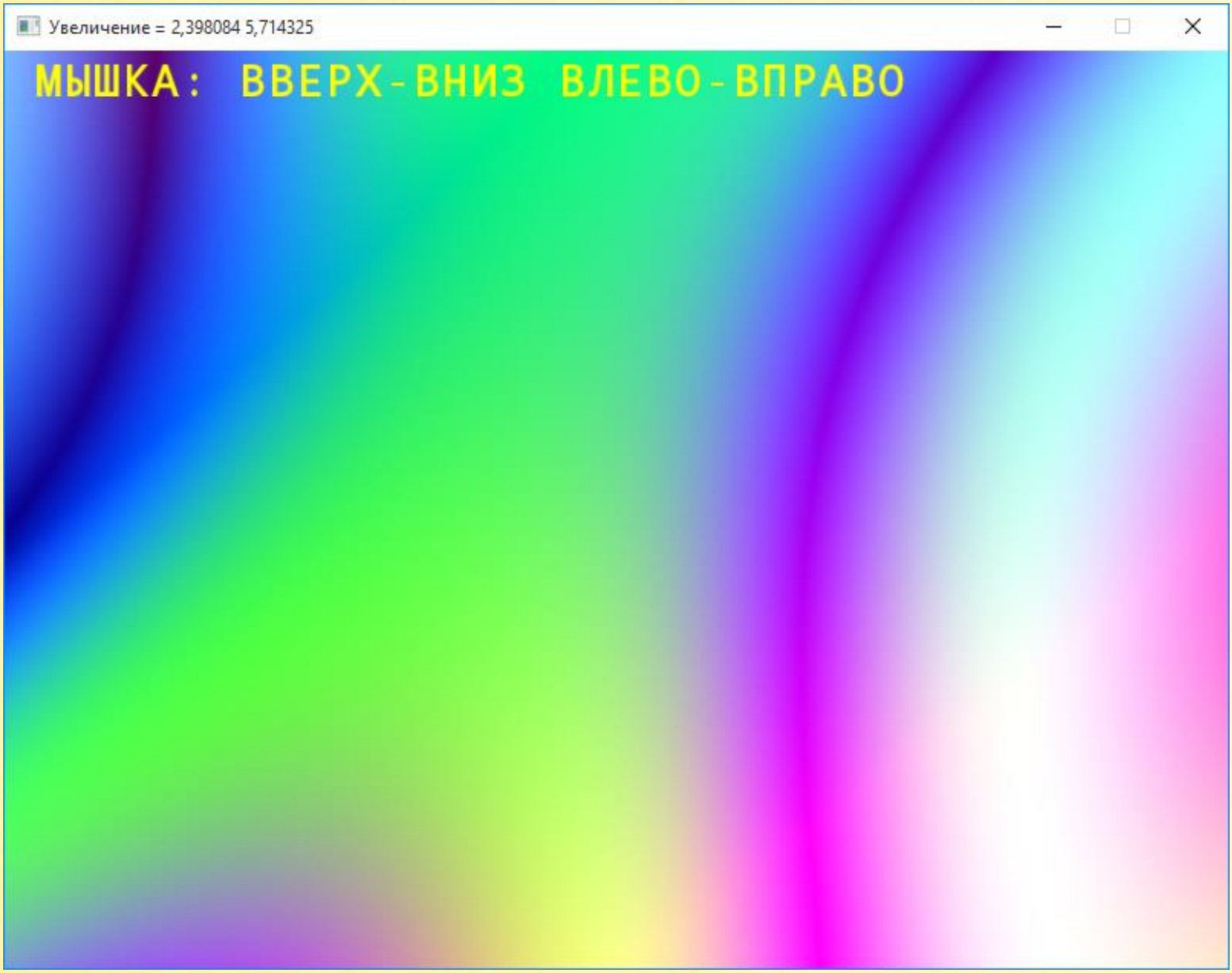
```
// ПЕРЕМЕЩАЕМ МЫШКУ
procedure Window_MouseMoved(sender: Object; e: MouseMoveEventArgs);
begin
    if (not Mouse.IsButtonPressed(Mouse.Button.Left)) then
        exit;

    // текущие координаты мышки:
    var x := e.X;
    var y := e.Y;

    var dx := x - mousePos.X;
    var dy := y - mousePos.Y;

    zoomX -= dx / 1000.0;
    zoomX := Math.Abs(zoomX);
    if (zoomX > 1) then
        zoomX := 1;

    zoomY += dy / 1000.0;
    if (zoomY > 1) then
        zoomY := 1;

    var w := Ceil(WIDTH * zoomX);
    var h := Ceil(HEIGHT * zoomY);
    _intrect.Width := w;
    _intrect.Height := h;

    wind.SetTitle('Увеличение = ' + 1/zoomX + ' ' + 1/zoomY);

    // запоминаем новые координаты мышки:
```

```
    mousePos := new Vector2i(x, y);
end;
```

Вот УВЕЛИЧЕННАЯ картинка шейдера *sandbox1.frag*:



А это УВЕЛИЧЕННАЯ картинка шейдера *sandboxA.frag*:


Увеличение = 2,398084 5,714325
МЫШКА: ВВЕРХ-ВНИЗ ВЛЕВО-ВПРАВО

Проект *Шейдеры 4*

Другой способ увеличения размеров картинки на экране – изменение размеров текущего **вида**. Сначала мы создаём вид **view** по размерам окна приложения:

```
// создаём вид по размерам окна:
var _view := new View(new FloatRect(new Vector2f(0, 0),
                                    new Vector2f(WIDTH, HEIGHT)));
```

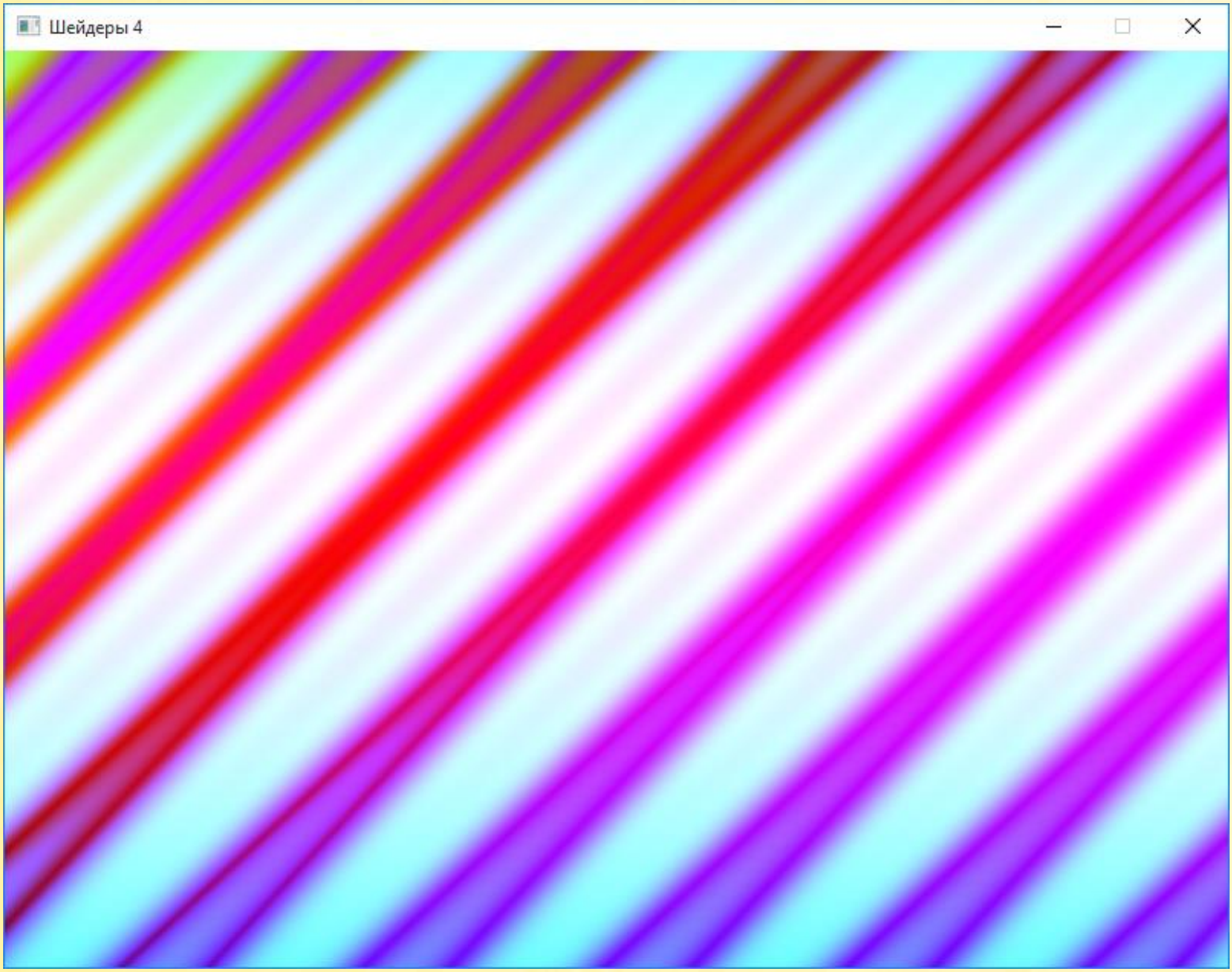
После запуска программы вы увидите картинку исходного размера.

В **игровом цикле** мы каждую итерацию немного уменьшаем размеры вида и тем самым **увеличиваем** масштаб изображения на экране:

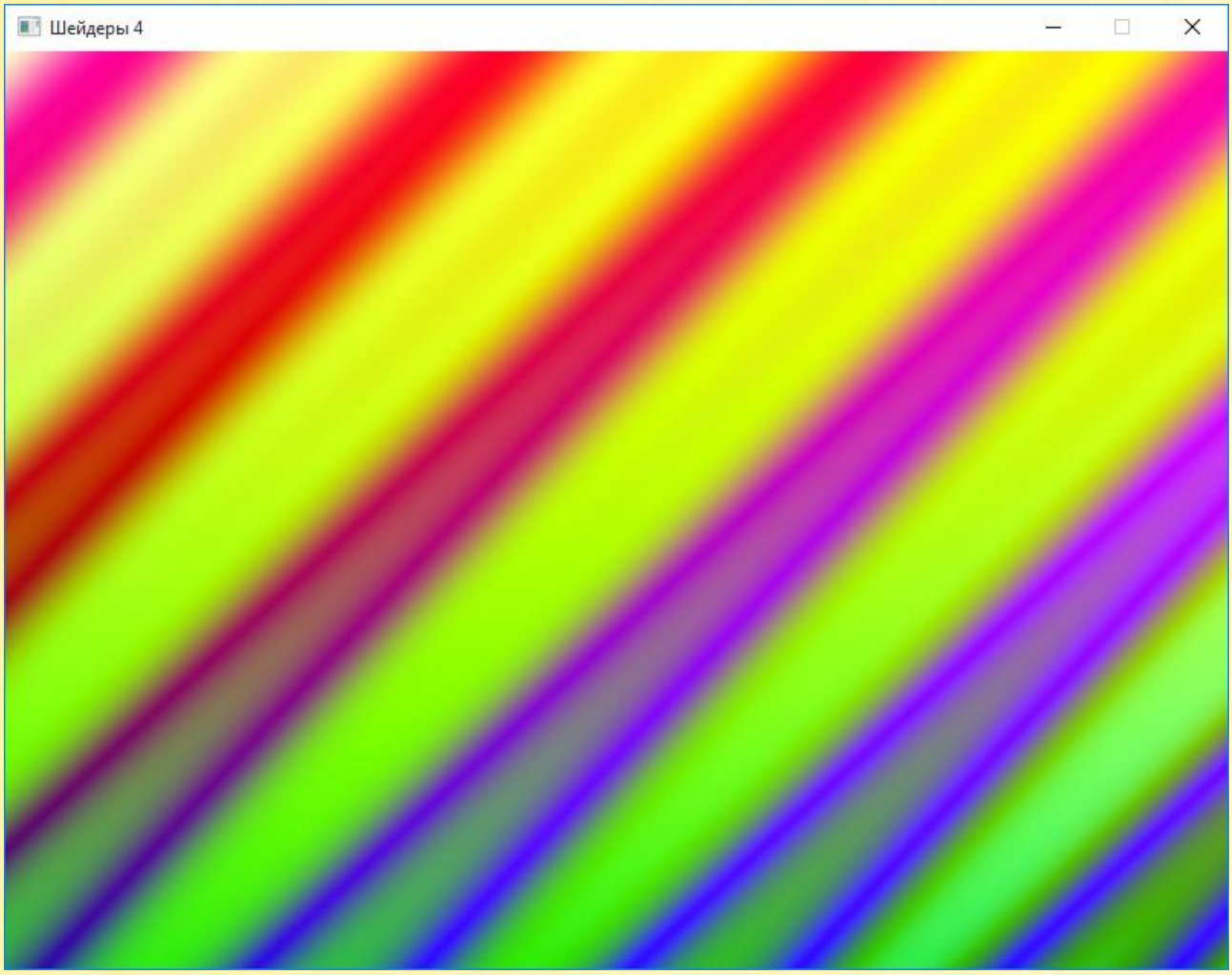
```
// игровой цикл:
while (wind.IsOpen) do
begin
   // вызываем все обработчики событий:
   wind.DispatchEvents();

   // обновляем параметр time:
   _shader.SetParameter('time', time);

   // рисуем картинку на вспомогательной текстуре:
   rtex.Draw(rect, states);

   // печатаем её на экране:
   rect.Texture := rtex.Texture;
   wind.Draw(rect);

   // увеличиваем время:
   time += 0.05;
   if (time < 40) then
   begin
       // уменьшаем размеры вида:
       _view.Zoom(0.999);
       wind.SetView(_view);

       // обновляем сцену:
```

```
        wind.Display();
    end;
end;
```

Полоски становятся всё шире и шире:



Пока не истечёт заданное время:



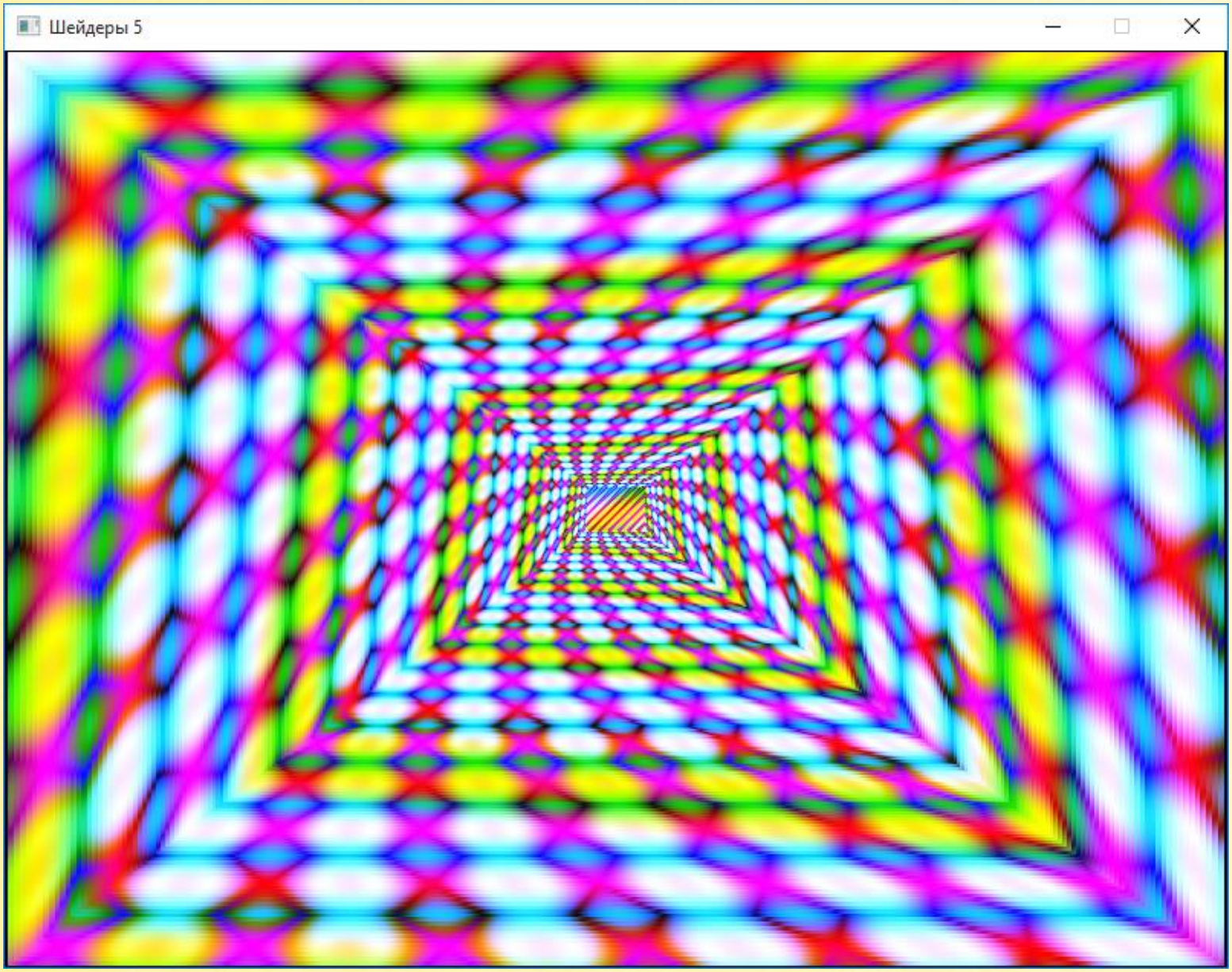
Проект *Шейдеры 5*

Интересный эффект получается, если размеры вида в игровом цикле не уменьшать, а увеличивать:

```
if (time < 40) then
begin
    // уменьшаем размеры вида:
    _view.Zoom(1.005);
    wind.SetView(_view);

    // обновляем сцену:
```

```
    wind.Display();
end;
```

Так как мы не очищаем окно, а картинка уменьшается, то возникает своеобразный туннель из картинок в разных масштабах:



Проект *Шейдеры своими руками*

Язык шейдеров довольно сложный и требует отдельного изучения, но самые простые шейдеры вы можете написать уже сейчас.

Фрагментный шейдер может окрасить текстуру в цвет, который задаётся цветовыми составляющими. Их значения изменяются от 0.0 до 1.0. Цветовые составляющие удобно хранить в переменных типа *float*.

Установив значения всех составляющих, вы создаёте вектор vec4 и присваиваете встроенной переменной gl_FragColor:

```
void main(void)
{
    float r = 1.0;
    float g = 0.0;
    float b = 0.0;
    float a = 1.0;
    gl_FragColor = vec4(r, g, b, a);
}
```

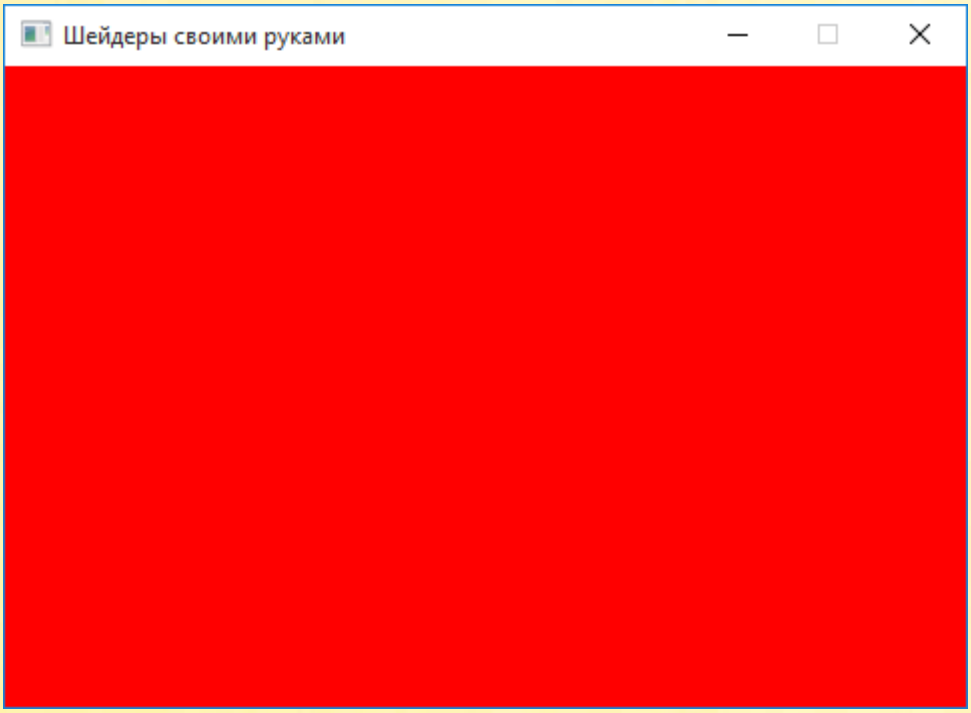
В этом шейдере мы задали красный непрозрачный цвет для всей текстуры.

В главном блоке программы загружаем шейдер, создаём текстуру нужных размеров и передаём её спрайту для печати на экране:

```
begin
   // создаём главное окно:
   CreateWindow(WIDTH, HEIGHT, 'Шейдеры своими руками');

   if (not Shader.IsAvailable) then
   begin
      wind.SetTitle('ШЕЙДЕРЫ НЕ ПОДДЕРЖИВАЮТСЯ!');
   end;
   // загружаем шейдер:
   var _shader := new Shader(nil, 'Media/color.frag');


   // создаём текстуру:
```

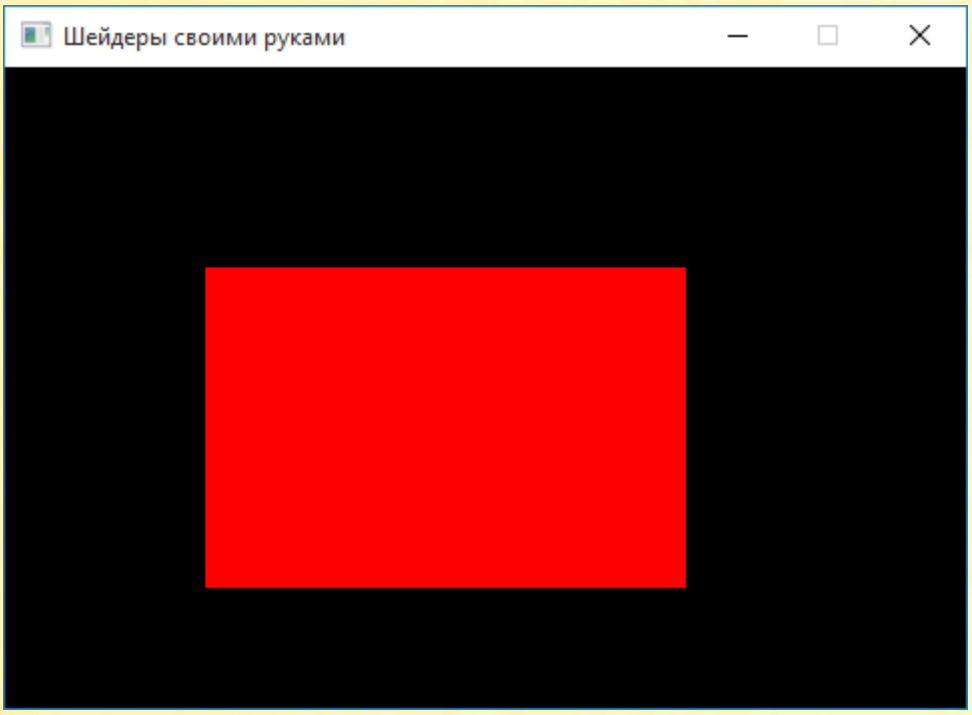
```
    var texture := new Texture(WIDTH, HEIGHT);

    // создаём спрайт из текстуры:
    var sprite := new Sprite(texture);

    // создаём шейдерный режим:
    var states := new RenderStates(_shader);

    // игровой цикл:
    while (wind.IsOpen) do
    begin
      // вызываем все обработчики событий:
      wind.DispatchEvents();

      // печатаем спрайт с шейдером:
      wind.Draw(sprite, states);

      // показываем на экране:
      wind.Display();
    end;
end.
```

Запускаем программу – вся клиентская часть окна окрашена в красный цвет:



Вы можете создать текстуру любых размеров, а спрайт перенести в любое место экрана:

```
// создаём текстуру:
//var texture := new Texture(WIDTH, HEIGHT);
var texture := new Texture(WIDTH div 2, HEIGHT div 2);
// создаём спрайт из текстуры:
var sprite := new Sprite(texture);
sprite.Position := new Vector2f(100,100);
```



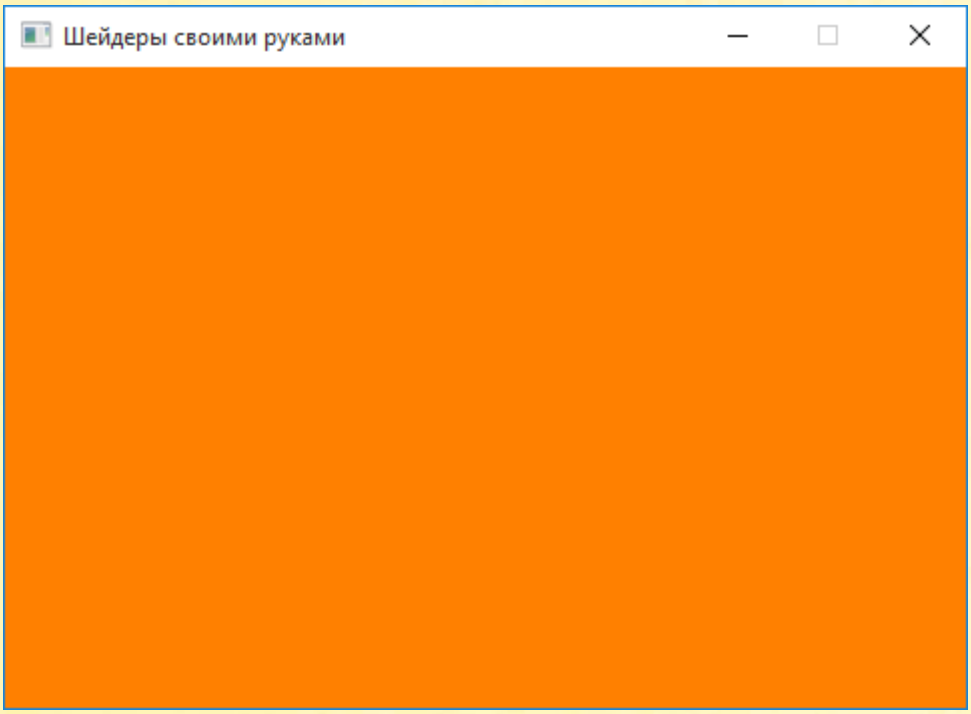
Чтобы сделать шейдер более гибким, заменим переменные параметрами для всех составляющих цвета:

```
uniform float r;
uniform float g;
uniform float b;
uniform float a;

void main(void)
{
    gl_FragColor = vec4(r, g, b, a);
}
```

Теперь вы можете в программе задавать значения этих параметров:

```
var _shader := new Shader(nil, 'Media/color2.frag');
```

```
_shader.SetParameter('r', 1.0);
_shader.SetParameter('g', 0.5);
_shader.SetParameter('b', 0.0);
_shader.SetParameter('a', 1.0);
```



Создадим **мигающее** окно.

В процедуре **CreateWindow** задаём смену кадров 1 раз в секунду:

```
// СОЗДАЁМ ОКНО
procedure CreateWindow(width, height: uint64;
                       title: string := 'SFML Window';
                       style: Styles := Styles.Close);
begin
   // создаём окно приложения:
   wind := new RenderWindow(new VideoMode(width, height), title, style);
```

```
    wind.SetVerticalSyncEnabled(true);
    wind.SetFramerateLimit(1);
```

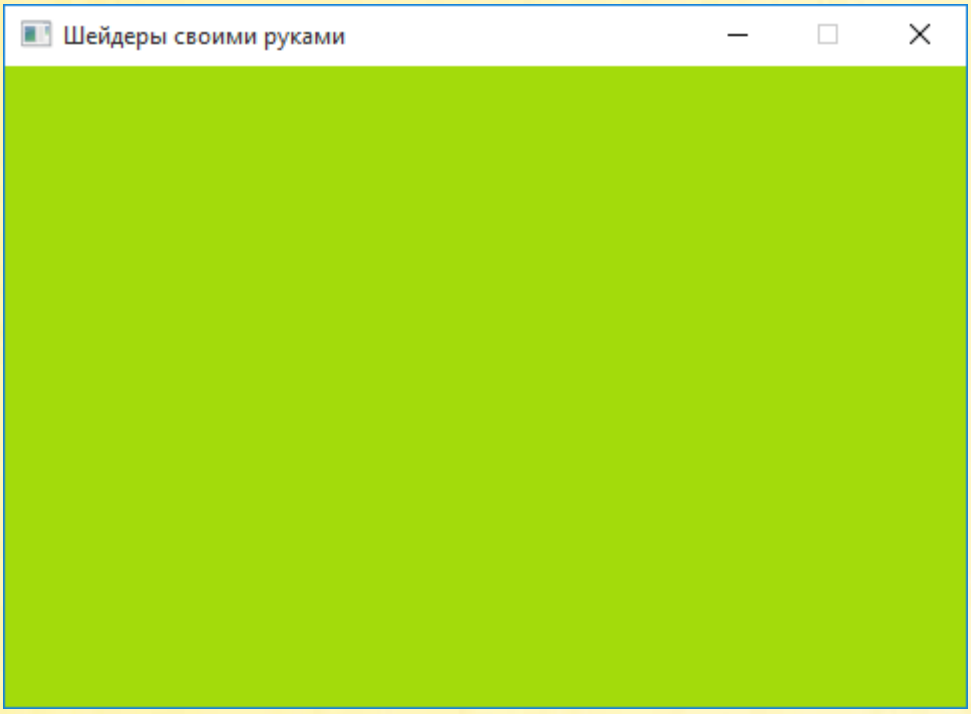
А главном блоке прозрачность устанавливаем в 1, потому что она не изменяется:

```
_shader.SetParameter('a', 1.0);
```

В игровом цикле каждую итерацию задаём случайные значения цветовым составляющим:

```
// игровой цикл:
while (wind.IsOpen) do
begin
    // вызываем все обработчики событий:
    wind.DispatchEvents();

    // задаём случайные составляющие цвета:
    var r := rand.NextDouble();
    _shader.SetParameter('r', r);
    var g := rand.NextDouble();
    _shader.SetParameter('g', g);
    var b := rand.NextDouble();
    _shader.SetParameter('b', b);

    // печатаем спрайт с шейдером:
    wind.Draw(sprite, states);

    // показываем на экране:
    wind.Display();
end;
```

Каждую секунду экран окрашивается в новый цвет:



Чтобы не мучить шейдер многочисленными изменениями параметров, перепишем шейдер:

```
uniform vec3 color;

void main(void)
{
    gl_FragColor = vec4(color, 1.0);
}
```

Пользоваться этим шейдером удобнее:

```
// загружаем шейдер:
var _shader := new Shader(nil, 'Media/color3.frag');

// задаём случайные составляющие цвета:
var r := rand.NextDouble();
//_shader.SetParameter('r', r);
var g := rand.NextDouble();
//_shader.SetParameter('g', g);
var b := rand.NextDouble();
//_shader.SetParameter('b', b);
```

```
_shader.SetParameter('color', r,g,b);
```

Если вы хотите изменять и прозрачность, то пользуйтесь таким шейдером:

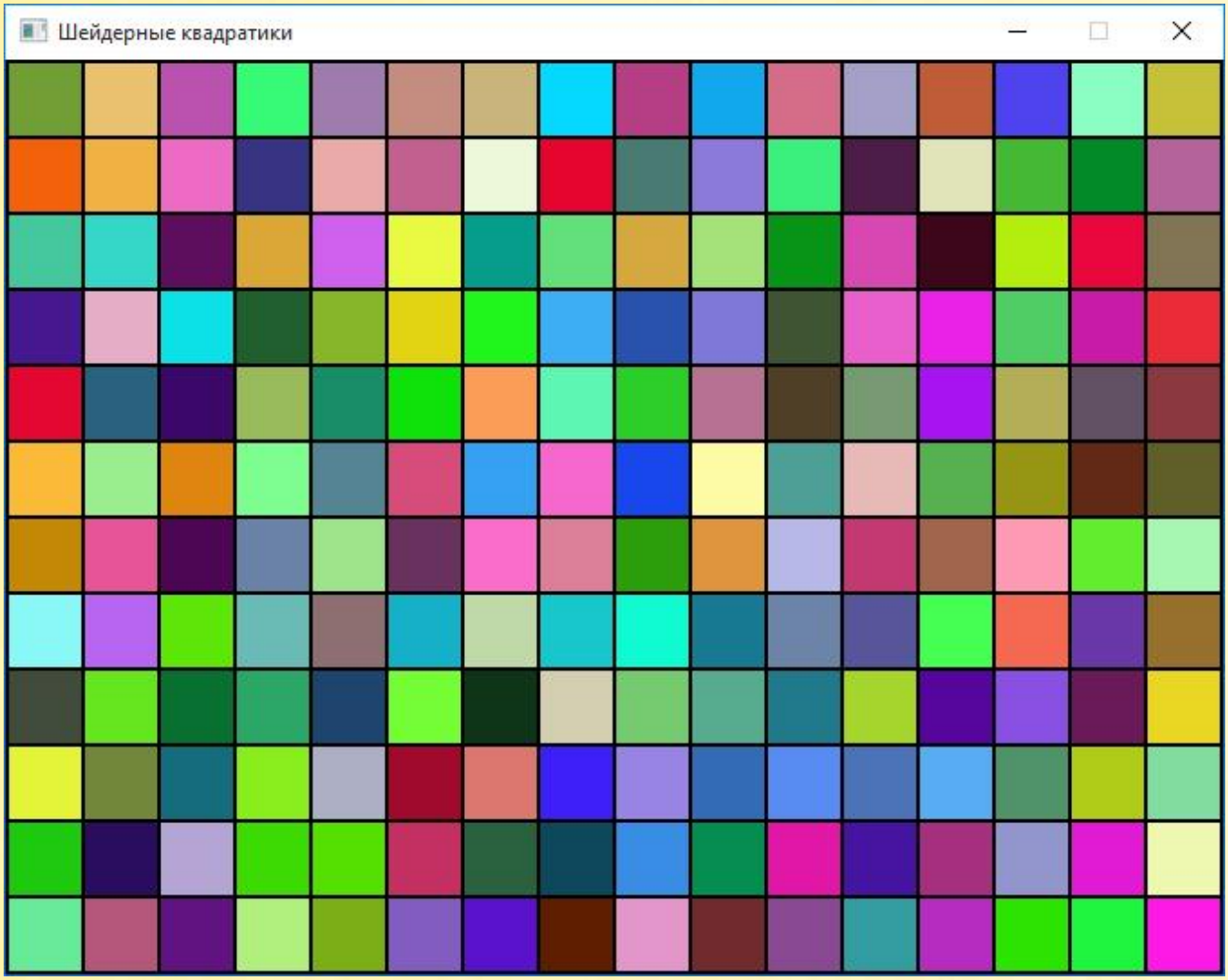
```
uniform vec4 rgba;

void main(void)
{
    gl_FragColor = vec4(rgba);
}
```

В этом примере прозрачность равна 1 (полная непрозрачность), но вы можете передать в шейдер любое значение от 0 до 1:

```
// загружаем шейдер:
var _shader := new Shader(nil, 'Media/color4.frag');

_shader.SetParameter('rgba', r,g,b, 1);
```

Проект *Шейдерные квадратики*

В этом проекте мы нарисуем целую сетку из цветных квадратиков:



Нам для этого нужен единственный квадратик quadr:

```
const
   // квадратик:
   Q_SIZE = 40;

  quadr := new RectangleShape(new Vector2f(Q_SIZE, Q_SIZE));
```

Чтобы регулировать расстояние между квадратиками, объявим константу OFFSET:

```
// зазоры между квадратиками:
OFFSET = 2;
```

Задаём число клеток в сетке и вычисляем размеры окна:

```
// размер сетки в клетках:
COLS = 16;
ROWS = 12;
// размеры окна:
WIDTH = (Q_SIZE + OFFSET) * COLS + OFFSET;
HEIGHT = (Q_SIZE + OFFSET) * ROWS + OFFSET;
```

В главном блоке загружаем шейдер *color3.frag*:

```
begin
   // создаём главное окно:
   CreateWindow(WIDTH, HEIGHT, 'Шейдерные квадратики');

   if (not Shader.IsAvailable) then
   begin
      wind.SetTitle('ШЕЙДЕРЫ НЕ ПОДДЕРЖИВАЮТСЯ!');
   end;
   // загружаем шейдер:
   var _shader := new Shader(nil, 'Media/color3.frag');

    // создаём шейдерный режим:
   var states := new RenderStates(_shader);
```

И начинаем игровой цикл:

```
// игровой цикл:
while (wind.IsOpen) do
begin
   // вызываем все обработчики событий:
   wind.DispatchEvents();
```

Цветные составляющие для каждого квадратика выбираем случайно, а его место в сетке вычисляем по несложной формуле:

```
      // рисуем сетку из квадратиков:
      for var row := 0 to ROWS-1 do
         for var col := 0 to COLS-1 do
         begin
            // задаём случайные составляющие цвета:
            var r := rand.NextDouble();
            var g := rand.NextDouble();
            var b := rand.NextDouble();
            _shader.SetParameter('color', r, g, b);
            // устанавливаем квадратик в сетке:
            quadr.Position := new Vector2f(OFFSET + col *
                                           (Q_SIZE + OFFSET),
                                            OFFSET + row *
                                           (Q_SIZE+OFFSET));
            // рисуем квадратик:
            wind.Draw(quadr, states);
         end;
         // обновляем сцену:
        wind.Display();
   end;

end.
```

Мы нарисовали цветную сетку, которая будет обновляться 20 раз в секунду:

```
wind.SetFramerateLimit(20);
```

Квадратики можно раскрашивать и шейдерами!

Проект *Шейдерный градиент*

Мы научились раскрашивать поверхности одним цветом. Следующий этап в освоении шейдеров – рисование градиентов.

В шейдерах это можно сделать несколькими способами. Самый простой из них – переписать паскалевский код из проекта *Градиент* на язык шейдеров.

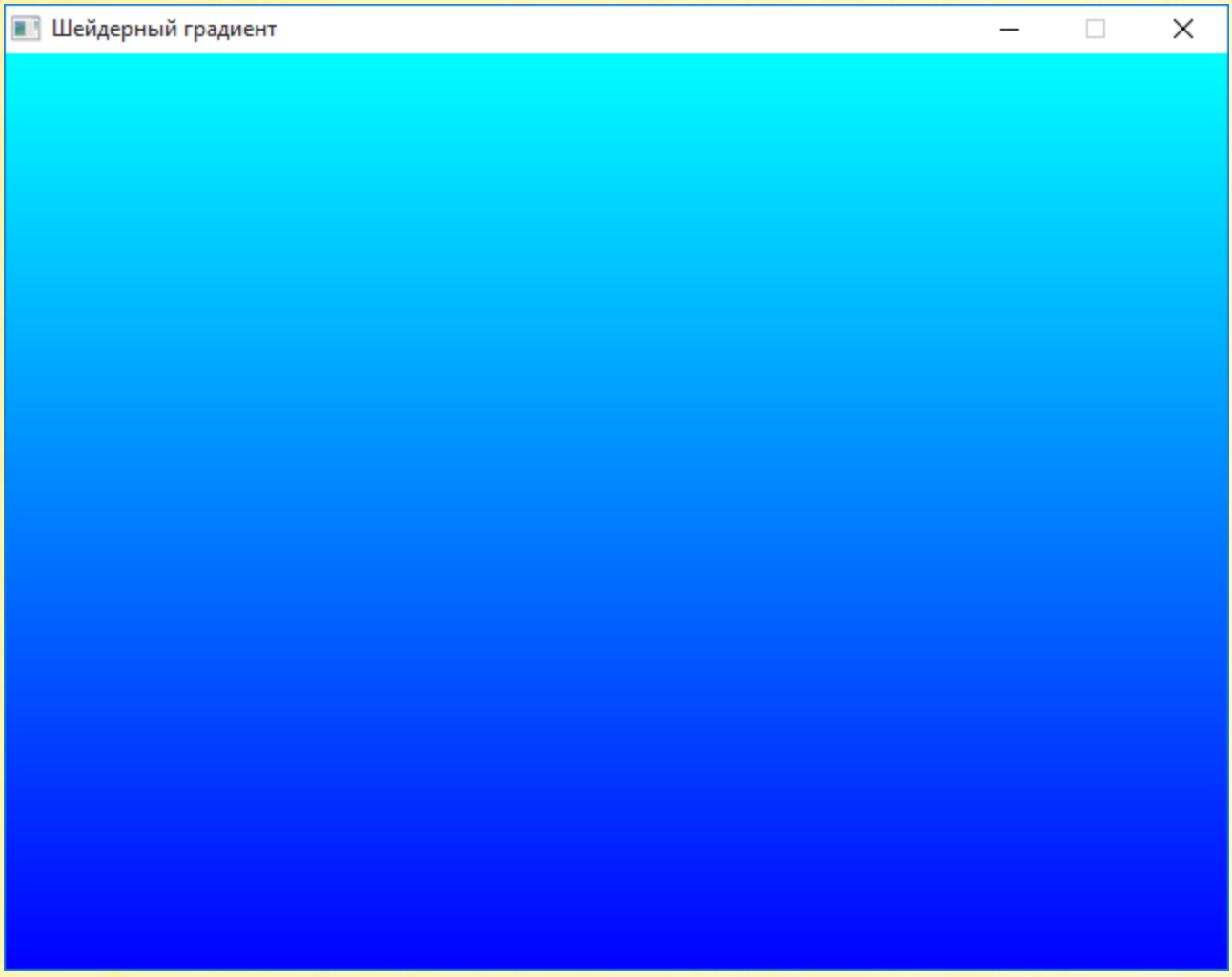
Текущие координаты пикселя в заданном фрагменте (у нас это оконные координаты) мы получаем от переменной *gl_FragCoord*. Значения цветовых составляющих изменяются в диапазоне 0.0..1.0, а не 0..255, как это принято в компьютерной графике. Значения двух составляющих не изменяются, а значение третьей составляющей изменяется от 1 до 0, в результате чего мы и получаем градиентный переход:

```
uniform vec2 resolution;

void main(void)
{
    // pixel coords:
    float x = gl_FragCoord.x;
    float y = gl_FragCoord.y;
    // blue:
    float b = 1;
    // red:
    float r = 0;
    // green:
    float g = y / resolution.y;
    // pixel color (rgb) --> (x,y):
    gl_FragColor = vec4(r, g, b, 1.0);
}
```

```
begin
   // создаём главное окно:
   CreateWindow(WIDTH, HEIGHT, 'Шейдерный градиент');

   if (not Shader.IsAvailable) then
   begin
```

```
      wind.SetTitle('ШЕЙДЕРЫ НЕ ПОДДЕРЖИВАЮТСЯ!');
   end;
   // загружаем шейдер:
   var _shader := new Shader(nil, 'Media/grad.frag');


   // создаём текстуру:
   var texture := new Texture(WIDTH, HEIGHT);
   // создаём спрайт из текстуры:
   var sprite := new Sprite(texture);

   // устанавливаем значение переменной resolution:
   _shader.SetParameter('resolution', new Vector2f(WIDTH, HEIGHT));

   // создаём шейдерный режим:
   var states := new RenderStates(_shader);

   // игровой цикл:
   while (wind.IsOpen) do
   begin
      // вызываем все обработчики событий:
      wind.DispatchEvents();

      // печатаем спрайт с шейдером:
      wind.Draw(sprite, states);

      // показываем на экране:
      wind.Display();
   end;

end.
```

Сам **градиент** ничуть не пострадал при переводе:

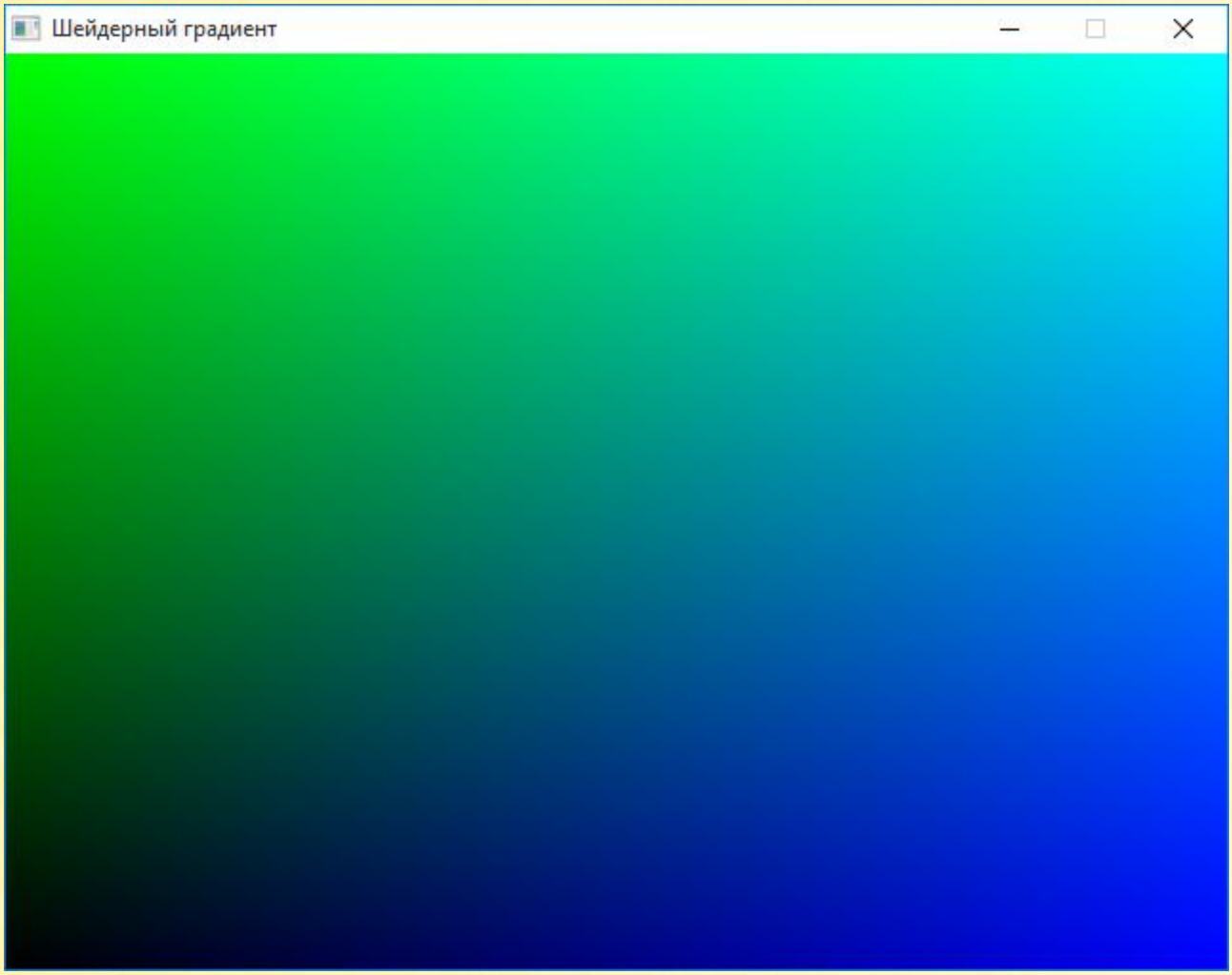


Если изменять ещё одну составляющую цвета, то получится более сложный градиент:

```
var _shader := new Shader(nil, 'Media/grad2.frag');
```

```
uniform vec2 resolution;

void main(void)
{
    // pixel coords:
    float x = gl_FragCoord.x;
    float y = gl_FragCoord.y;
    // blue:
    float b = x / resolution.x;
```

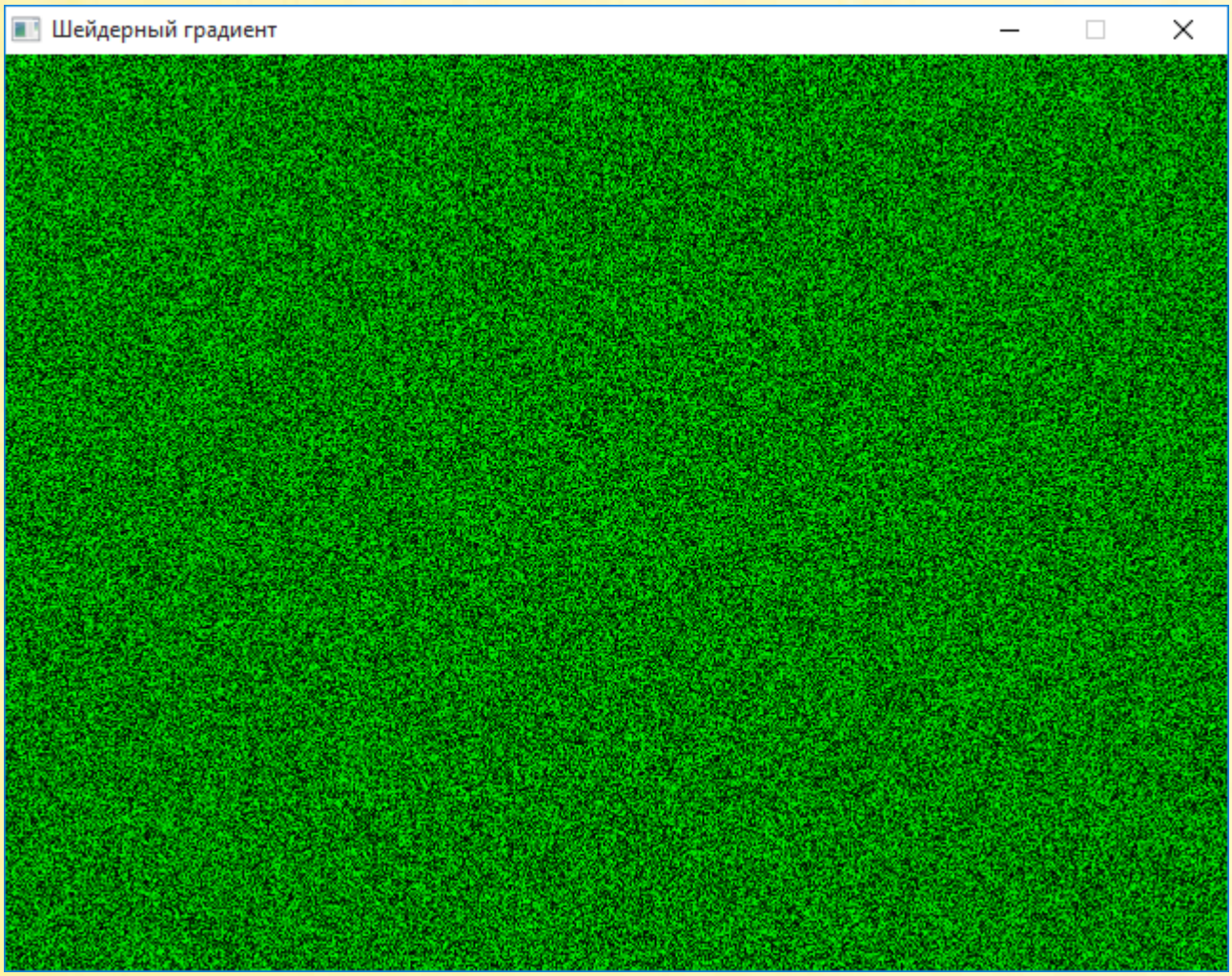
```
    // red:
    float r = 0;
    // green:
    float g = y / resolution.y;
    // pixel color (rgb) --> (x,y):
    gl_FragColor = vec4(r, g, b, 1.0);
}
```



Вы можете поэкспериментировать со случайными значениями:

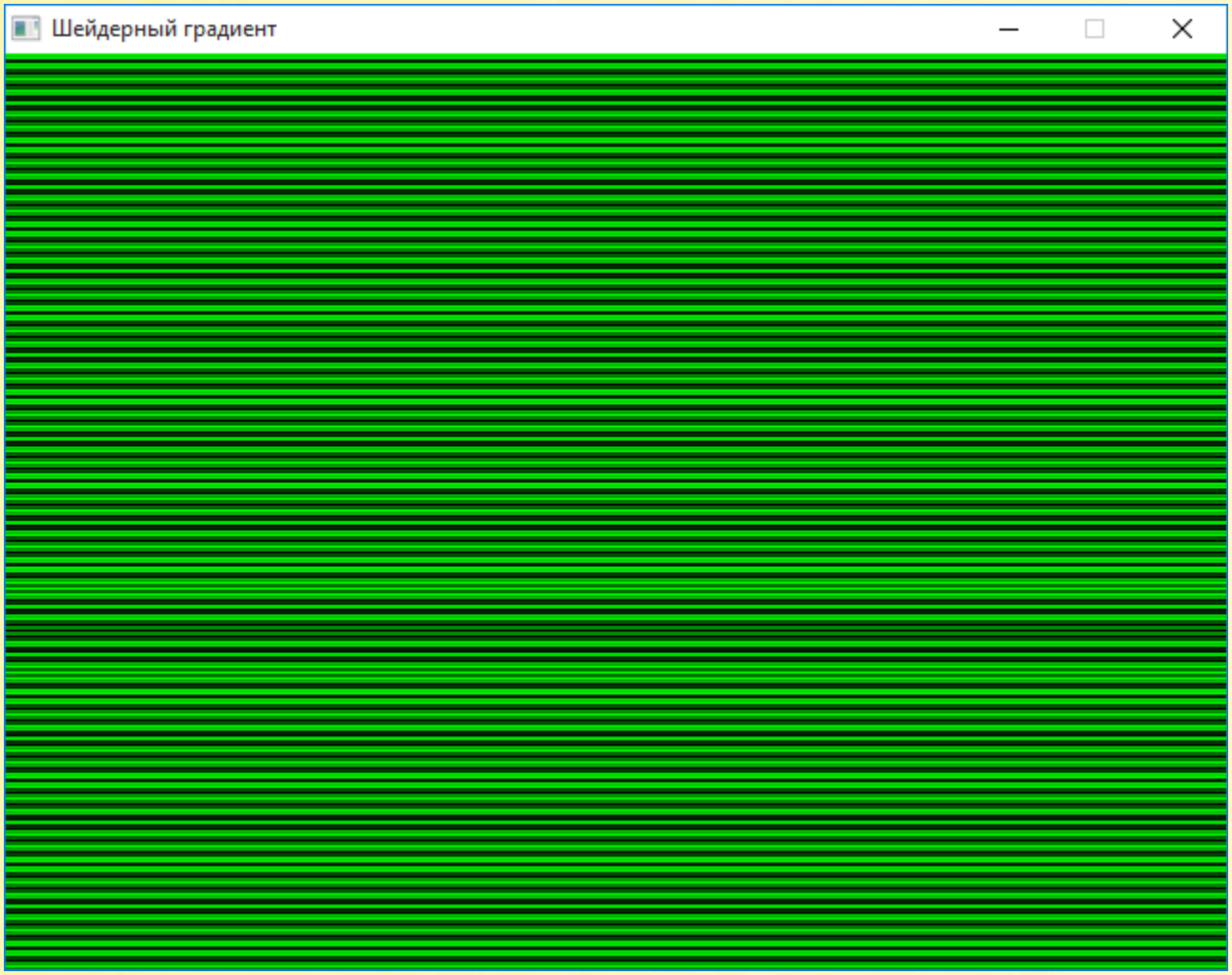
```
var _shader := new Shader(nil, 'Media/grad3.frag');
```

```
uniform vec2 resolution;

void main(void)
{
    // pixel coords:
    float x = gl_FragCoord.x;
    float y = gl_FragCoord.y;
    // blue:
    float b = 0;
    // red:
    float r = 0;
    // green:
    float g = fract(sin(dot(gl_FragCoord.xy, vec2(12.9898, 78.233))) * 43758.5453);
    // pixel color (rgb) --> (x,y):
    gl_FragColor = vec4(r, g, b, 1.0);
}
```



И с синусами:

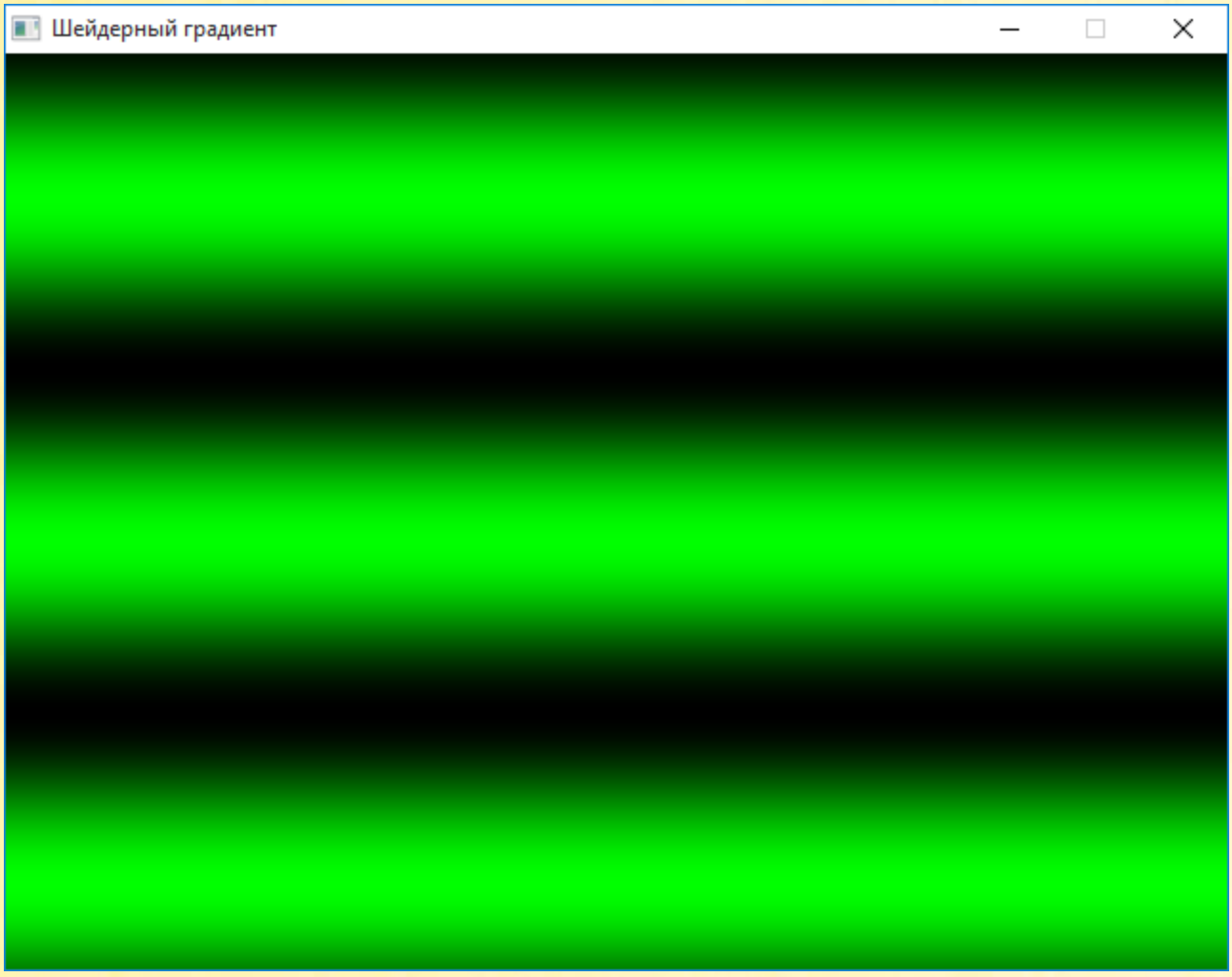
```
float g = fract(sin(gl_FragCoord.y));
```




```
float g = sin(gl_FragCoord.y);
```

А вот и наши синусоидные полоски:

```
float g = (sin(y * 0.035) + 1.0) / 2.0;
```



Проект *Шейдерный градиент 2*

Простенький шейдер из книги *The Book of Shaders*:

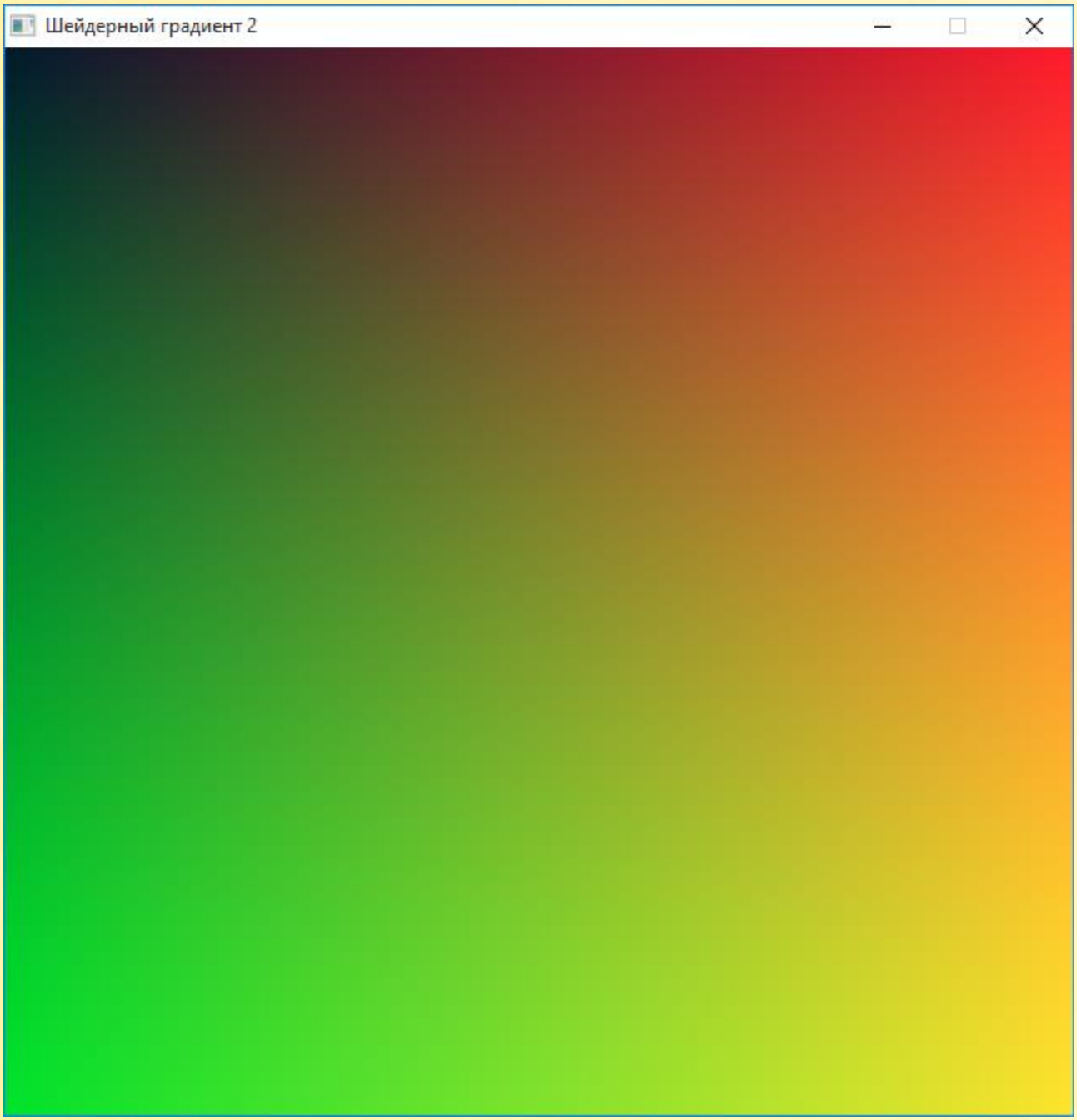
```
uniform vec2 resolution;
uniform float time;

void main() {
    vec2 st = gl_FragCoord.xy / resolution.xy;
    vec3 color = vec3(st.x, abs(cos(st.y + time * sqrt(2))), abs(sin(time)));

    gl_FragColor = vec4(color, 1.0);
}
```

Я слегка обработал этот шейдер, что ничуть его не ухудшило:

```
begin
   // создаём главное окно:
   CreateWindow(WIDTH, HEIGHT, 'Шейдерный градиент 2');

   if (not Shader.IsAvailable) then
   begin
      wind.SetTitle('ШЕЙДЕРЫ НЕ ПОДДЕРЖИВАЮТСЯ!');
   end;
   // загружаем шейдер:
   var _shader := new Shader(nil, 'Media/bookofsh.frag');

   // создаём текстуру:
   var texture := new Texture(WIDTH, HEIGHT);
   // создаём спрайт из текстуры:
   var sprite := new Sprite(texture);

   // устанавливаем значение переменной resolution:
   _shader.SetParameter('resolution', new Vector2f(WIDTH, HEIGHT));

   // создаём шейдерный режим:
   var states := new RenderStates(_shader);

   var time := 0.0;

   // игровой цикл:
   while (wind.IsOpen) do
   begin
      // вызываем все обработчики событий:
      wind.DispatchEvents();

     _shader.SetParameter('time', time);

      // печатаем спрайт с шейдером:
      wind.Draw(sprite, states);

      time += 0.01;

      // показываем на экране:
      wind.Display();
   end;
```

```
end.
```

Цвета со временем изменяются, поэтому за их игрой наблюдать весьма любопытно:



Проект *Таблица Пифагора*

В номере 8 журнала *Наука и жизнь* за 2004 год, на страницах 99-100 напечатана любопытная статья *Узоры таблицы Пифагора*. Таблица Пифагора – это всем известная со школьных времён таблица умножения, которую якобы придумал сам Пифагор:

	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7
2	2	4	6	8	10	12	14
3	3	6	9	12	15	18	21
4	4	8	12	16	20	24	28
5	5	10	15	20	25	30	35
6	6	12	18	24	30	36	42
7	7	14	21	28	35	42	49

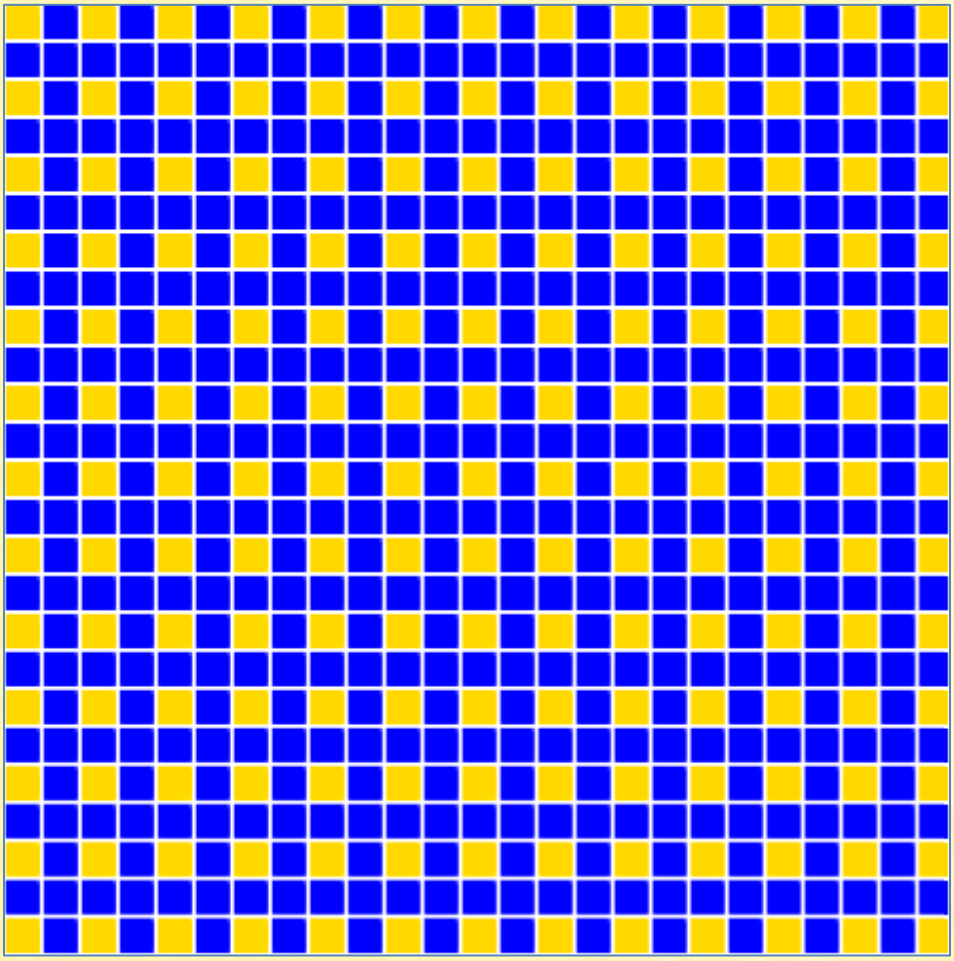
Казалось бы, таблица ничем не примечательна, однако пытливые умы нашли в ней немало интересного! С некоторыми свойствами таблицы Пифагора мы познакомимся в этом проекте.

В журнале *Мир информатики* были предложены задачи по журнальной статье:

“Мир информатики”
Журнал для тех, у кого информатика – любимый школьный предмет
Выпуск № 6, ноябрь 2016 г.

Нас интересует такая задача:

7. Разработайте программу, в результате выполнения которой на экран будет выведено следующее изображение:



Она не имеет непосредственного отношения к таблице Пифагора, но зато мы хорошенько подготовимся к решению более сложных задач.

Неизменяемые параметры программы сохраним в **константах**:

```
const
    // размер клеток в пикселях:
    QSIZE = 25;
    // число клеток по горизонтали:
    NCOL = 25;
    // число клеток по вертикали:
    NROW = 25;
    //толщина линий:
    PEN_WIDTH = 1;
    // цвет фона:
    BACKCOLOR = Color.Black;
    // размеры окна:
    WIDTH = (QSIZE + PEN_WIDTH) * NCOL + PEN_WIDTH;
    HEIGHT = (QSIZE + PEN_WIDTH) * NROW + PEN_WIDTH;
```

Также нам потребуются **переменные**:

```
var
    // окно приложения:
    wind: RenderWindow := nil;

    // квадратик:
    quadr := new RectangleShape(new Vector2f(QSIZE, QSIZE));
    _shader : Shader;
    states : RenderStates;

    // цвета клеток:
    colors := new Color[2] (Color.Blue, ColorEx.Gold);
```

На картинке хорошо видно, что все клетки окрашены в 2 цвета – синий и золотой. Мы поместили цвета в массив **colors**.

В **главном блоке** мы создаём окно, загружаем цветной шейдер и вызываем процедуру *Draw* для рисования таблицы:

```
begin
    // создаём главное окно:
```

```
   CreateWindow(WIDTH, HEIGHT, 'Таблица Пифагора');

   if (not Shader.IsAvailable) then
   begin
      wind.SetTitle('ШЕЙДЕРЫ НЕ ПОДДЕРЖИВАЮТСЯ!');
   end;
   // загружаем шейдер:
   _shader := new Shader(nil, 'Media/color3.frag');

   // создаём шейдерный режим:
   states := new RenderStates(_shader);

   // рисуем таблицу:
   Draw();

   // игровой цикл:
   while (wind.IsOpen) do
   begin
      // вызываем все обработчики событий:
      wind.DispatchEvents();
   end;

end.
```

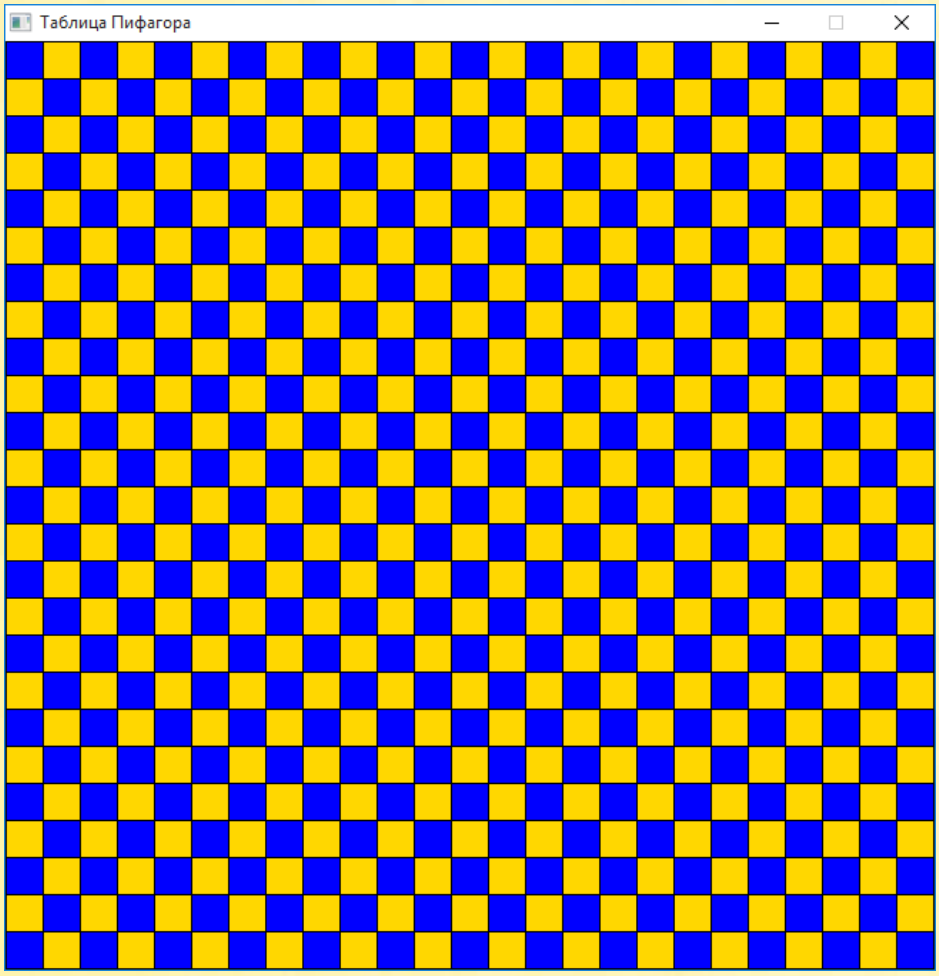
Процедура Draw очень простая.

Мы перебираем все клетки таблицы и определяем для каждой номер цвета в массиве colors – 0 или 1. Составляющие этого цвета мы передаём в шейдер, а затем печатаем окрашенный квадратик в соответствующую клетку таблицы:

```
// РИСУЕМ ТАБЛИЦУ
procedure Draw();
begin
   // рисуем сетку из квадратиков:
   for var row := 1 to NROW do
      for var col := 1 to NCOL do
      begin
         // шахматная раскраска:
         var c := (row + col) mod 2;
         var clr := colors[c];
        var r := clr.R/255;
```

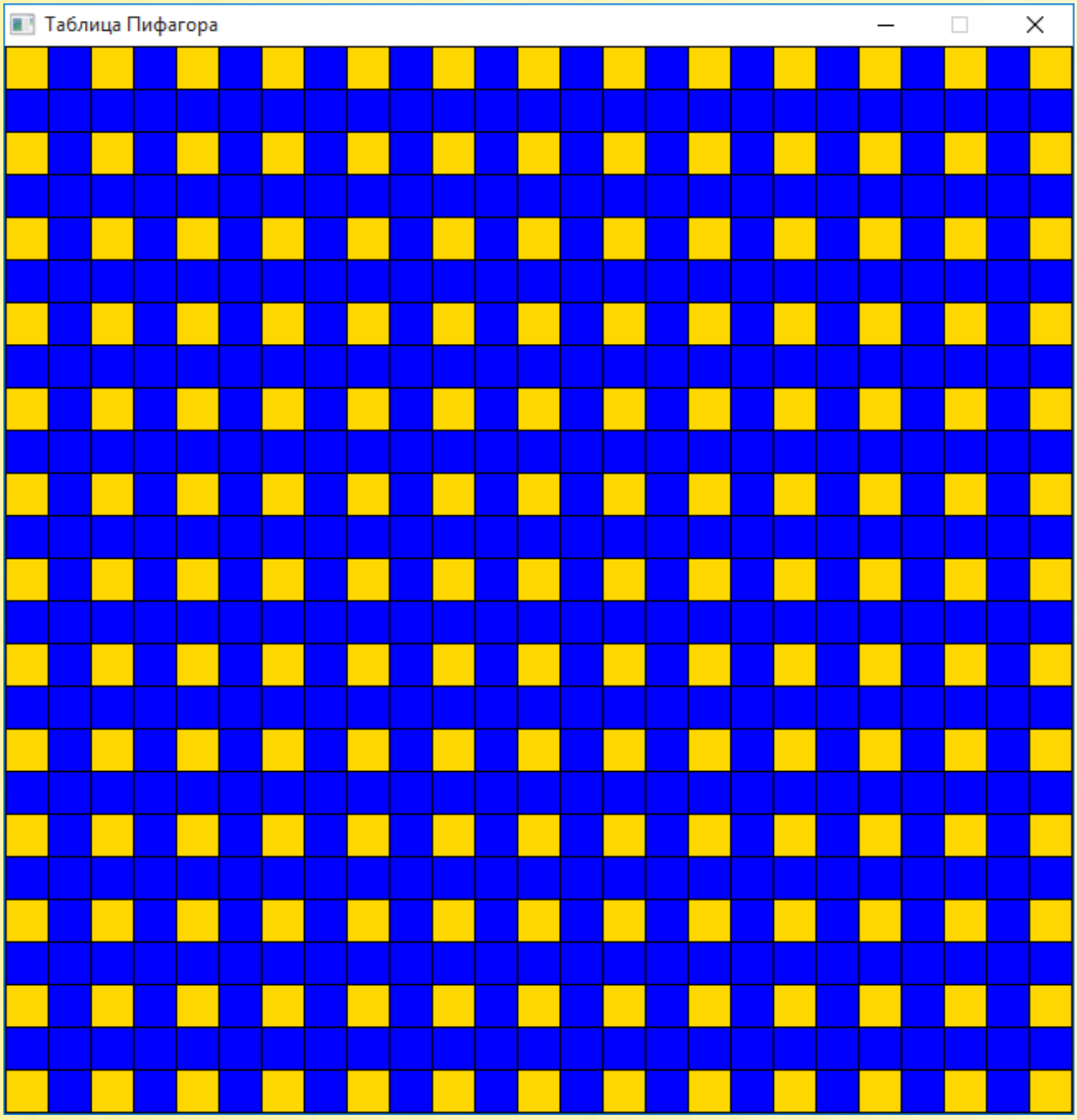
```
            var g := clr.G/255;
            var b := clr.B/255;
            _shader.SetParameter('color', r, g, b);
            // устанавливаем квадратик в сетке:
            quadr.Position := new Vector2f(PEN_WIDTH + (col-1) *
                                           (QSIZE + PEN_WIDTH),
                                            PEN_WIDTH + (row-1) *
                                           (QSIZE+PEN_WIDTH));
            // рисуем квадратик:
            wind.Draw(quadr, states);
        end;
        // обновляем сцену:
       wind.Display();
end;
```

В данном случае будет напечатана не заданная в журнале таблица, а **шахматная доска**, которая тоже по-своему красива:



Чтобы решить журнальную задачу, достаточно изменить одну строку в процедуре *Draw*:

```
// шахматная раскраска:
//var c := (row + col) mod 2;
// журнальная раскраска:
var c := row * col mod 2;
```



Проект *Таблица Пифагора 2*

Так как цвет клетки определяется её координатами, то можно написать *лямбда-выражение* для вычисления цвета клетки. Тогда выбор цвета клеток в циклах упростится:

```
// РИСУЕМ ТАБЛИЦУ
procedure Draw();
begin
   // шахматная раскраска:
   var fclr: (integer,integer) -> integer := (c,r) -> (r + c) mod 2;

      // рисуем сетку из квадратиков:
      for var row := 0 to NROW-1 do
         for var col := 0 to NCOL-1 do
         begin
            // шахматная раскраска:
            var clr := colors[fclr(col, row)];
```

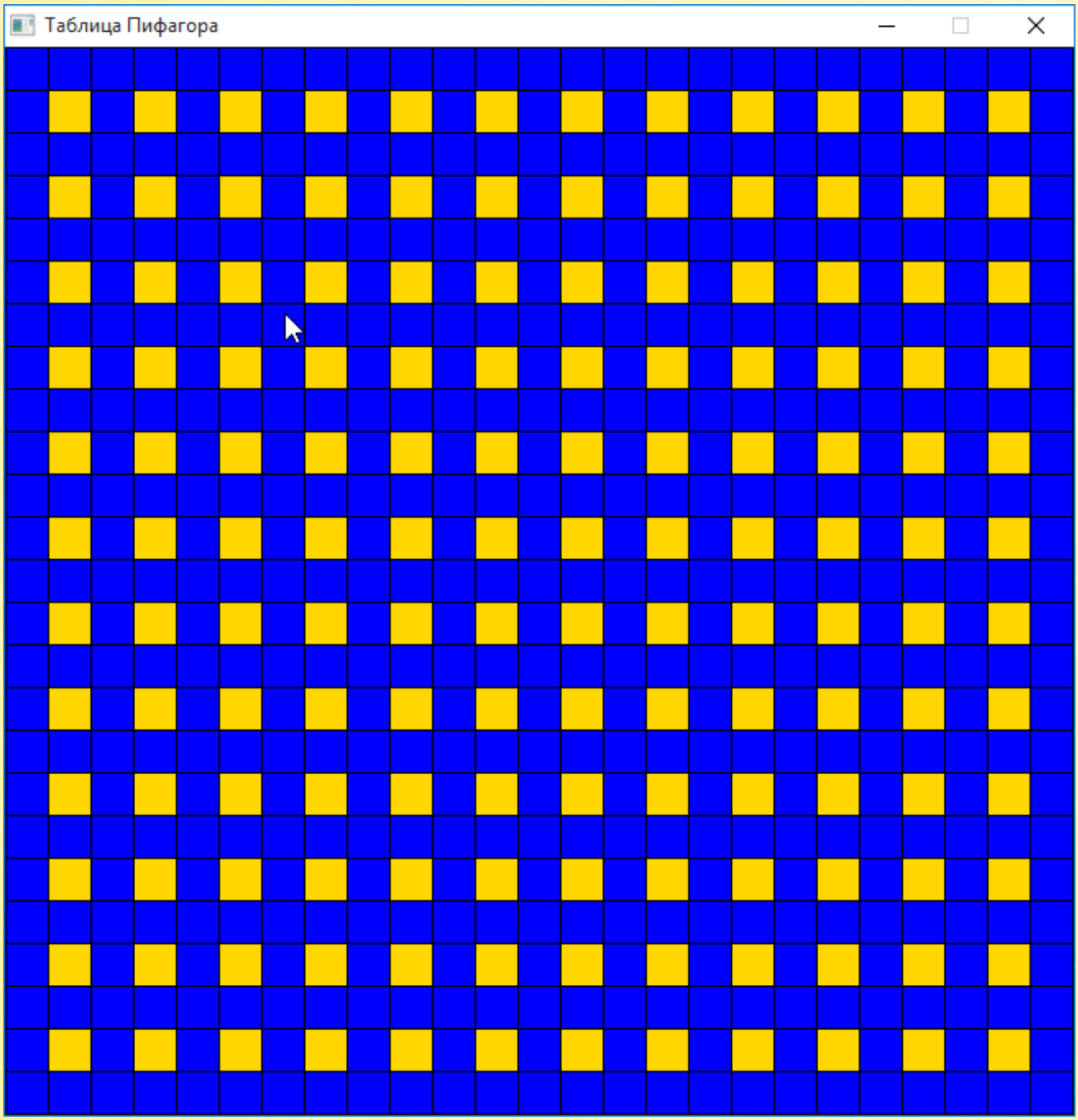
Аналогичная функция для журнальной раскраски клеток:

```
// журнальная раскраска:
var fclr2: (integer,integer) -> integer :=
           (c,r) -> (r + 1) mod 2 * (c + 1) mod 2;

// шахматная раскраска:
//var clr := colors[fclr(col, row)];
// журнальная раскраска:
var clr := colors[fclr2(col, row)];
```

Если немного поэкспериментировать с лямбдами, то можно найти и другие варианты раскрашивания таблицы:

```
// почти журнальная раскраска:
var fclr3: (integer,integer) -> integer := (c,r) -> (r * c) mod 2;
```

```
// шахматная раскраска:
//var clr := colors[fclr(col, row)];
// журнальная раскраска:
//var clr := colors[fclr2(col, row)];
// почти журнальная раскраска:
var clr := colors[fclr3(col, row)];
```



Проект *Таблица Пифагора 3*

Следующая задача из журнала *Мир информатики* – это **вторая задача** из журнала *Наука и жизнь*:

Чтобы получить представление о том, как в таблице Пифагора расположены числа, дающие одинаковые остатки при делении, например на 5, закрасим числа, дающие остатки 0, 1,2, 3, 4, каждое своим цветом. Как это ни удивительно, но таблица Пифагора оказывается расчлененной на совершенно одинаковые по раскраске квадраты (Рис. слева). В журнале *Мир информатики* клетки раскрашены по-другому (Рис. справа).

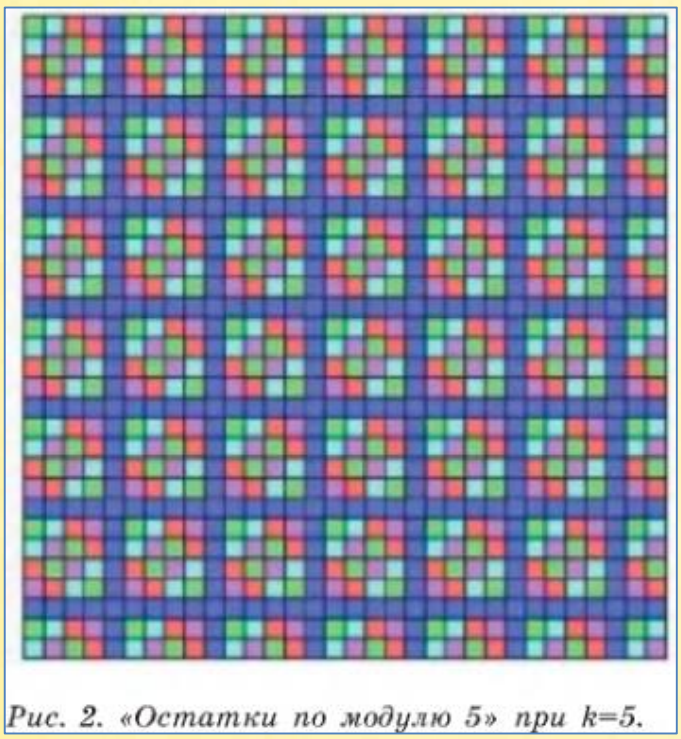
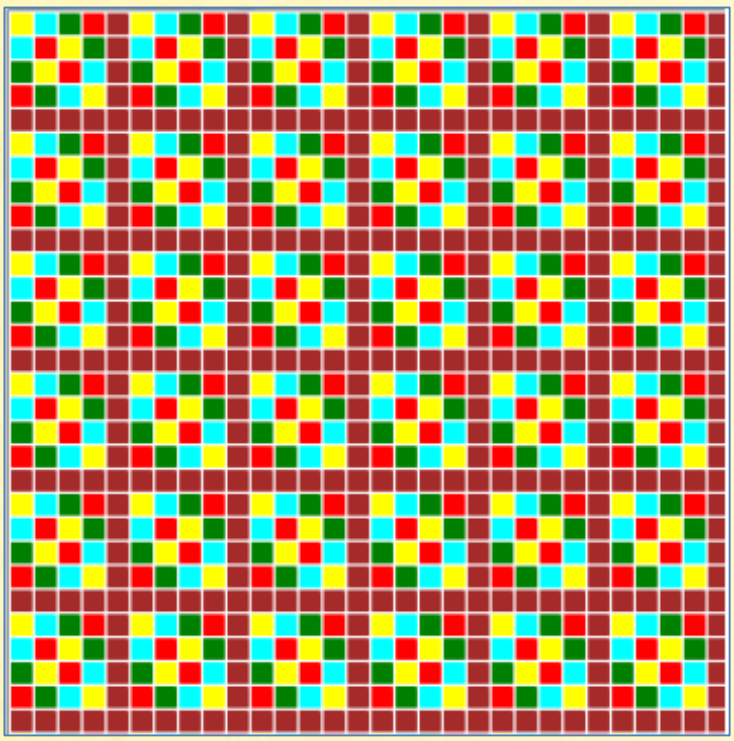


Рис. 2. «Остатки по модулю 5» при k=5.



Мы раскрасим квадратики по последнему образцу.

Но сначала изменим значения **констант**:

```
const
   // размер клеток в пикселях:
   QSIZE = 20;
```

```
    // число клеток по горизонтали:
    NCOL = 30;
    // число клеток по вертикали:
    NROW = 30;
```

Как следует из условия задачи, нам потребуется 5 цветов:

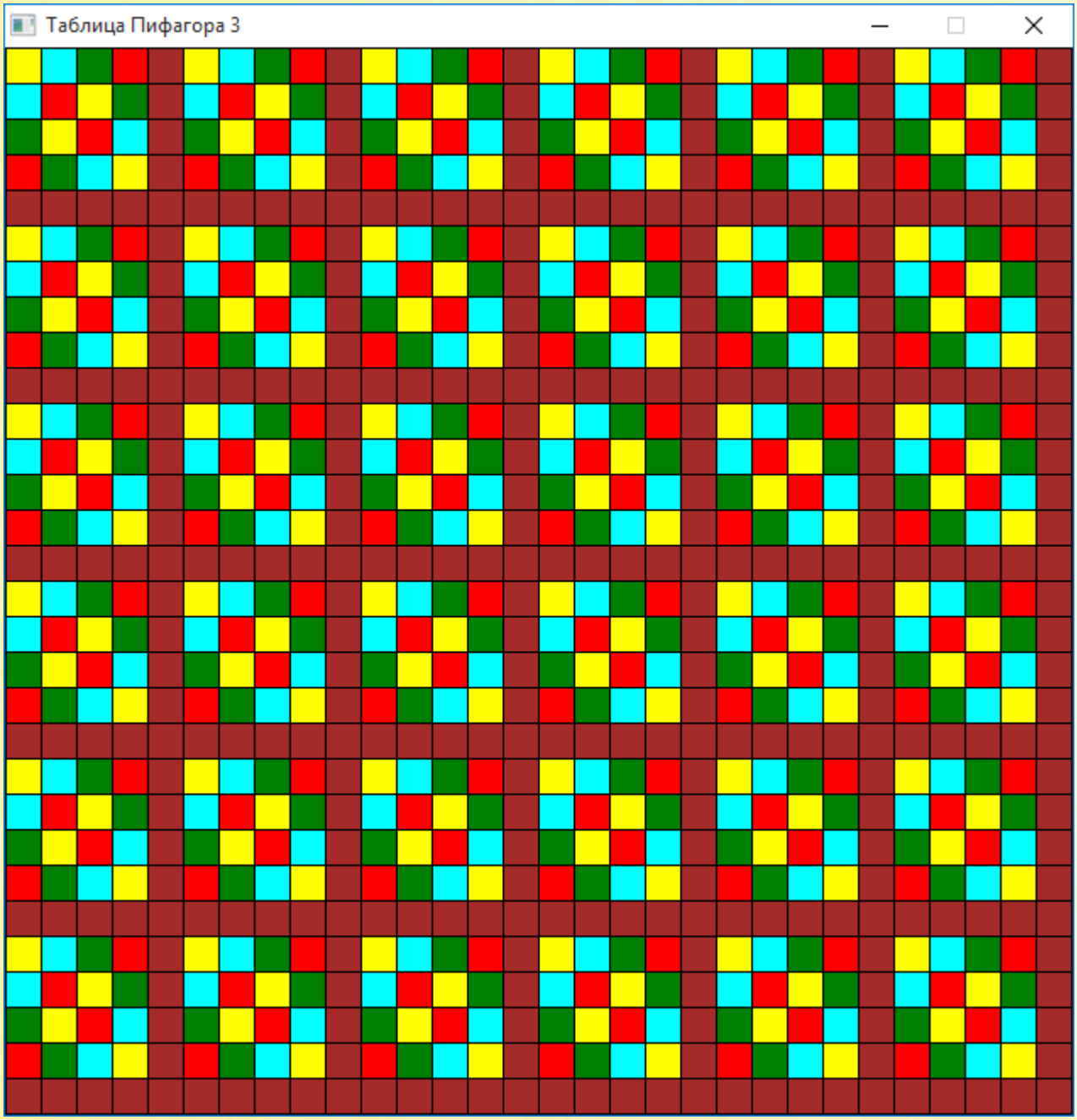
```
// цвета клеток:
 colors := new Color[] (ColorEx.Brown,
                        ColorEx.Yellow,
                        ColorEx.Cyan,
                        ColorEx.Green,
                        ColorEx.Red);
```

Всё, что нам нужно сделать в процедуре Draw, - написать новую функцию для вычисления цвета квадратиков:

```
// РИСУЕМ ТАБЛИЦУ
procedure Draw();
begin
   // журнальная раскраска:
   var fclr: (integer,integer) -> integer :=
             (c,r) -> (r + 1) mod 5 * (c + 1) mod 5;

  // рисуем сетку из квадратиков:
  for var row := 0 to NROW-1 do
      for var col := 0 to NCOL-1 do
      begin
        // цвет клетки:
        var clr := colors[fclr(col, row)];
        var r := clr.R/255;
        var g := clr.G/255;
        var b := clr.B/255;
        _shader.SetParameter('color', r, g, b);
        // устанавливаем квадратик в сетке:
        quadr.Position := new Vector2f(PEN_WIDTH + col *
                                       (QSIZE + PEN_WIDTH),
                                        PEN_WIDTH + row *
                                       (QSIZE+PEN_WIDTH));
        // рисуем квадратик:
        wind.Draw(quadr, states);
```

```
        end;
        // обновляем сцену:
        wind.Display();
end;
```

Таблица получилась один в один:



Проект *Таблица Пифагора 4*

Мы сделаем процедуру Draw более универсальной, если добавим параметр k, то есть число, для которого мы находим остатки от деления:

```
// РИСУЕМ ТАБЛИЦУ
procedure Draw(k: integer);
begin
   // журнальная раскраска:
   var fclr: (integer,integer,integer) -> integer := (c,r,k) -> (r +
1) * (c + 1) mod k;

  // рисуем сетку из квадратиков:
  for var row := 0 to NROW-1 do
      for var col := 0 to NCOL-1 do
      begin
        // цвет клетки:
        var clr := colors[fclr(col, row, k)];
        var r := clr.R/255;
        var g := clr.G/255;
        var b := clr.B/255;
        _shader.SetParameter('color', r, g, b);
        // устанавливаем квадратик в сетке:
        quadr.Position := new Vector2f(PEN_WIDTH + col *
                                       (QSIZE + PEN_WIDTH),
                                        PEN_WIDTH + row *
                                       (QSIZE+PEN_WIDTH));
        // рисуем квадратик:
        wind.Draw(quadr, states);
     end;
     // обновляем сцену:
     wind.Display();
end;
```

Вызов процедуры в главном блоке следует исправить:

```
// чертим таблицу:
Draw(5);
```

Для k = 5 мы получим ту же самую таблицу, что и в предыдущем примере.
Для k < 5 получаются аналогичные, но более простые узоры.
Если k > 5, то нужно дополнить массив цветов новыми элементами!

Проект *Таблица Пифагора 5*

Третья задача из журнала *Наука и жизнь* (Задача 8 из журнала *Мир информатики*):

Благодаря свойству периодичности таблицы Пифагора по остаткам на экране возникают разнообразные мозаики. Очевидно, чем больше k, тем больше будет остатков r, тем больше потребуется цветов. Чтобы узоры не были слишком пёстрыми, ограничимся, например, тремя цветами. Для этого остатки сгруппируем по модулю 3, то есть первым цветом закрасим числа таблицы с остатками 1, 4, 7, 10.., вторым - числа с остатками 2, 5, 8, 11.., а третьим - числа, кратные 3.

Работы в этом проекте опять будет немного.

Подправляем значения константа:

```
const
   // размер клеток в пикселях:
   QSIZE = 10;
   // число клеток по горизонтали:
   NCOL = 65;
   // число клеток по вертикали:
   NROW = 65;
```

Переставляем цвет а массиве colors:

```
// цвета клеток:
colors := new Color[] (Color.Blue,
                       ColorEx.Brown,
                       ColorEx.Yellow,
```

```
                    ColorEx.Cyan,
                    ColorEx.Green,
                    ColorEx.Red);
```

В процедуру Draw добавляем ещё 1 параметр – n, - который отвечает за число цветов мозаике, и переписываем функцию раскраски:

```
// РИСУЕМ ТАБЛИЦУ
procedure Draw(k, n: integer);
begin
   // функция раскраски:
   var fclr: (integer,integer,integer,integer) -> integer :=
             (c,r,k,n) -> ((r + 1) * (c + 1) mod k) mod n;

  // рисуем сетку из квадратиков:
  for var row := 0 to NROW-1 do
      for var col := 0 to NCOL-1 do
      begin
        // цвет клетки:
        var clr := colors[fclr(col, row, k, n)];
      . . .
```

И наконец, в главном блоке вызываем эту процедуру с соответствующими аргументами:

```
// чертим таблицу:
Draw(13, 3);
```

Вот такая сложная мозаика получилась в этом проекте:

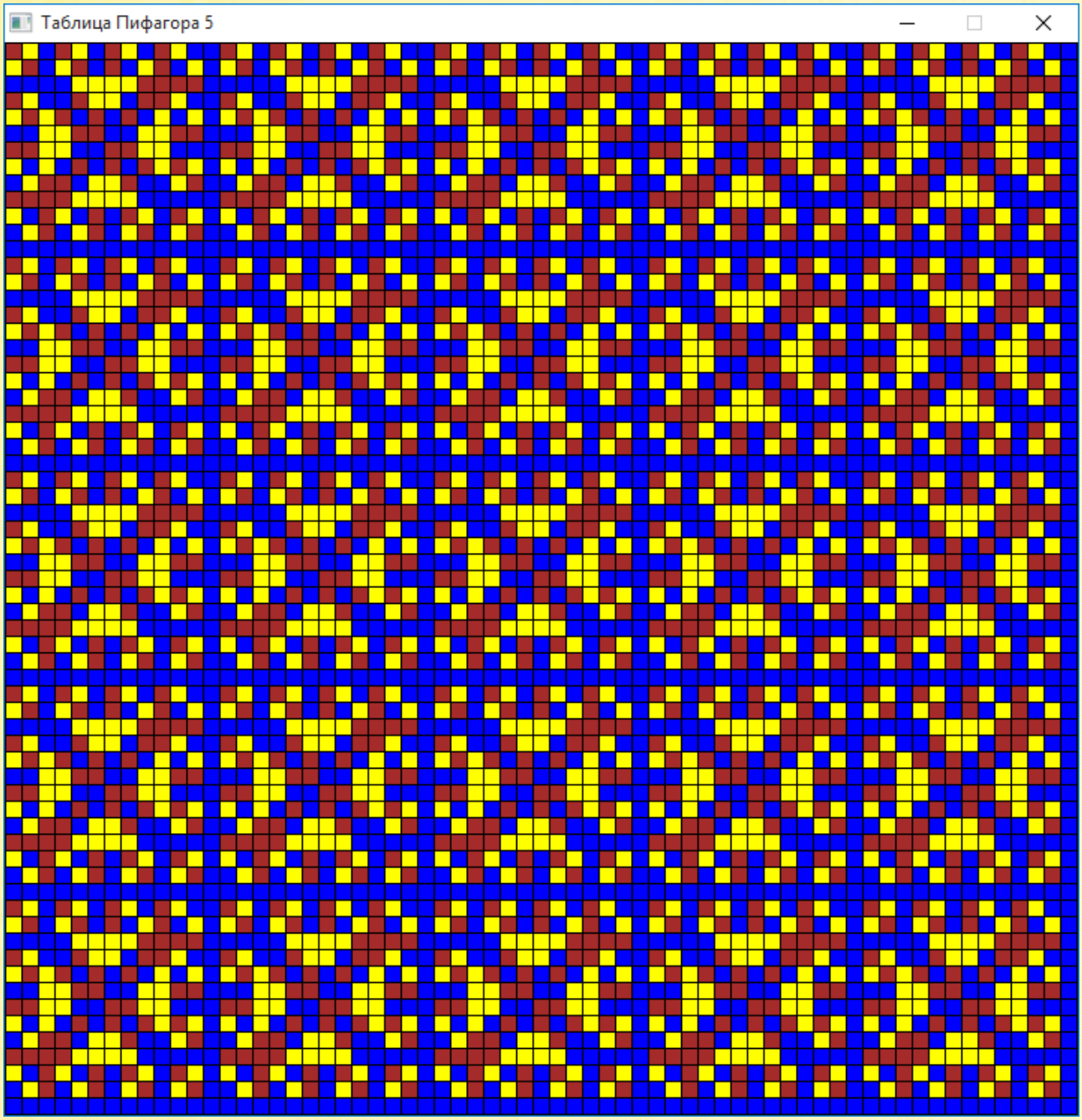
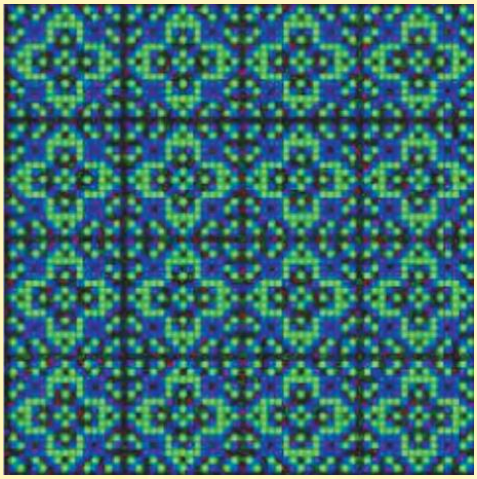


Таблица Пифагора 5

Проект *Таблица Пифагора 6*

Пятая задача из *Науки и жизни* (Задача 9, *Рис. 13* из журнала *Мир информатики*):

Еще один вариант трехцветных мозаик приведен на рис. 5. Здесь для большей симметрии одинаковым цветом закрашены не только числа с одинаковым остатком r, но и числа с остатком, дополняющим r до k.

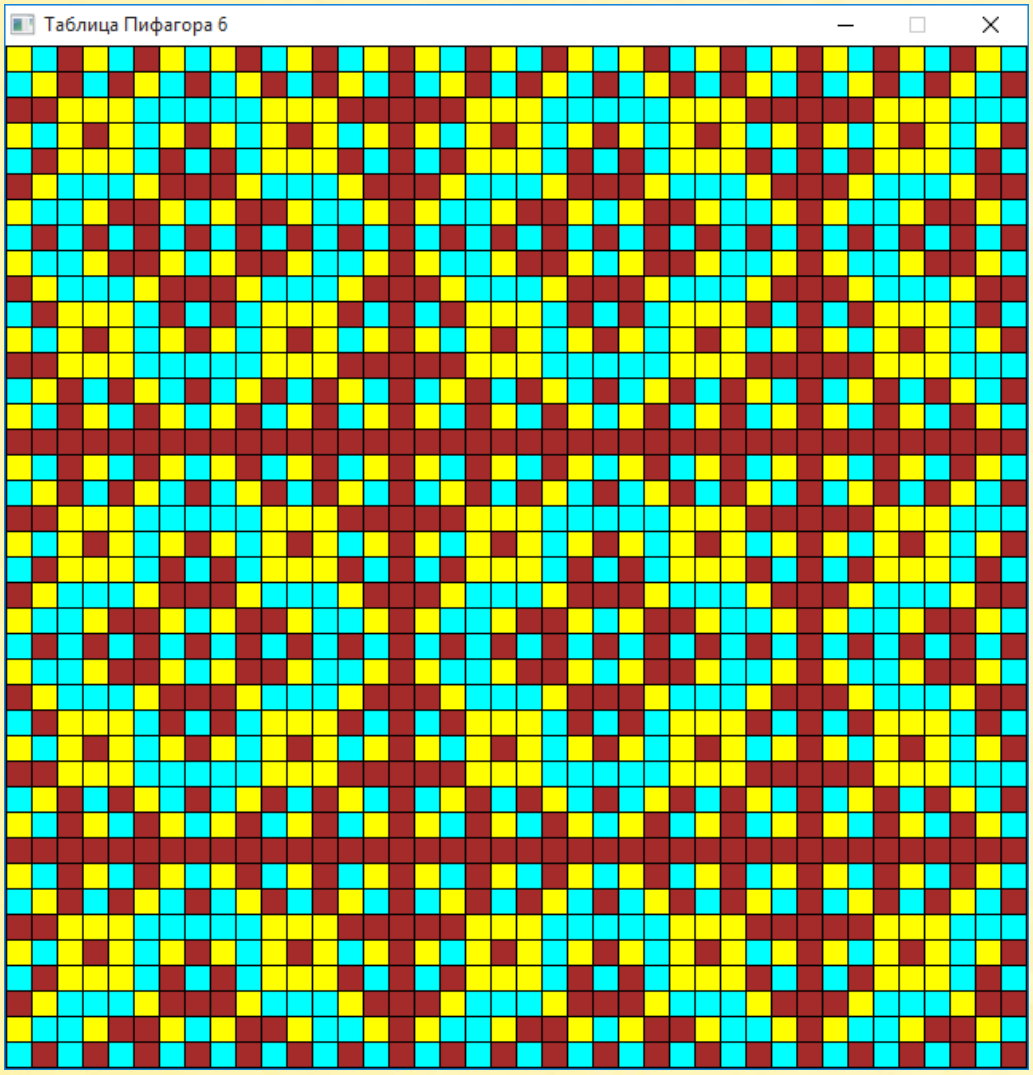


Делаем традиционные изменения в предыдущей программе:

```
const
   // размер клеток в пикселях:
   QSIZE = 15;
   // число клеток по горизонтали:
   NCOL = 40;
   // число клеток по вертикали:
   NROW = 40;

  // цвета клеток:
   colors := new Color[] (ColorEx.Brown,
                          ColorEx.Yellow,
                          ColorEx.Cyan,
                          Color.Blue,
                          ColorEx.Green,
                          ColorEx.Red);

// РИСУЕМ ТАБЛИЦУ
procedure Draw(k, n: integer);
```

```
begin
   // функция раскраски:
   var f: (integer,integer,integer) -> integer :=
          (c,r,k) -> (r + 1) * (c + 1) mod k;
   var fclr: (integer,integer,integer,integer) -> integer :=
             (c,r,k,n) -> f(c,r,k) < k div 2 ? f(c,r,k) mod n :
             (k - f(c,r,k)) mod n;

   // чертим таблицу:
   Draw(16, 3);
```

Мозаика готова:

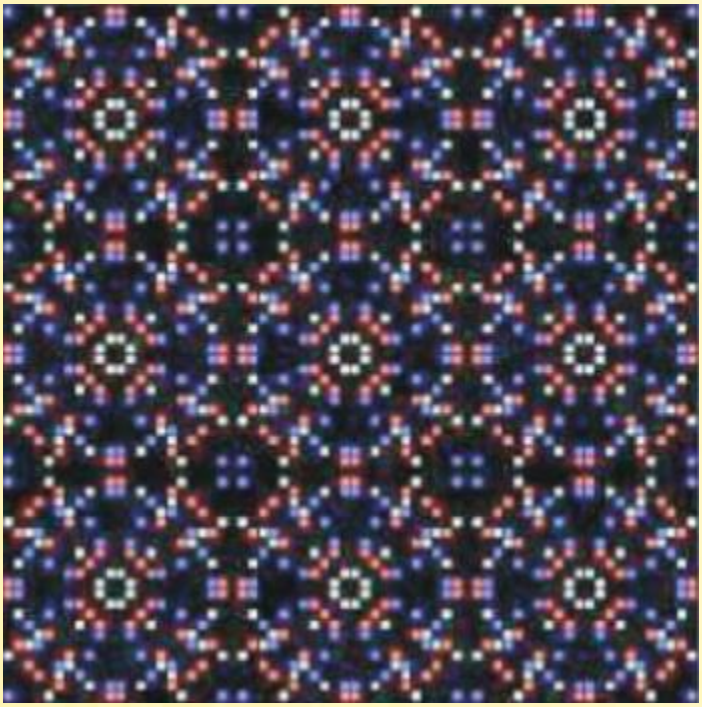


В журнале *Мир информатики* она имеет меньше клеток и другие цвета.

Проект *Таблица Пифагора 7*

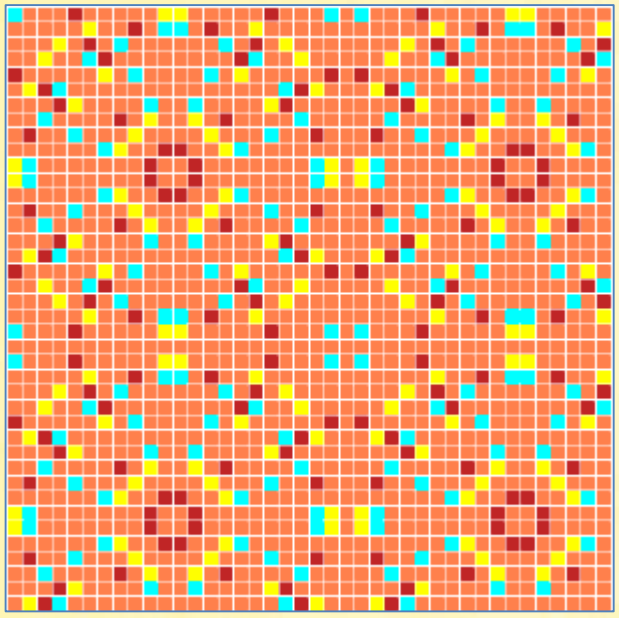
Шестая задача из *Науки и жизни* (Задача 10, *Рис. 14* из журнала *Мир информатики*):

Интересные мозаики возникают и тогда, когда красят не все числа, а выборочно. Например, трехцветный узор на рис. 6.



В журнале *Мир информатики* условие задачи дополнено:

Например, узор на рис. 14 получен при k = 23 в случае закраски разными цветами только трех групп чисел:
1) для которых остаток от деления произведения на 23 равен 1 или 22;
2) для которых остаток от деления произведения на 23 равен 11 или 12;
3) для которых остаток от деления произведения на 23 равен 5 или 18.
Остальные числа (соответствующие им квадраты) закрашиваются неким четвёртым цветом.



Константы остаются без изменений, а порядок цветов необходимо поменять:

```
// цвета клеток:
colors := new Color[] (ColorEx.Red,
                       ColorEx.Cyan,
                       ColorEx.Yellow,
                       ColorEx.Brown,
                       Color.Blue,
                       ColorEx.Green
                       );
```

Сильнее всего пострадала процедура Draw:

```
// РИСУЕМ ТАБЛИЦУ
procedure Draw();
begin
   // функция раскраски:
   var f: (integer,integer) -> integer :=
          (c, r) -> (r + 1) * (c + 1) mod 23;

  // рисуем сетку из квадратиков:
  for var row := 0 to NROW-1 do
      for var col := 0 to NCOL-1 do
      begin
```

```
            // цвет клетки:
             var clr := f(col, row);
             if (clr = 1) or (clr = 22)
                then clr := 1
             else if (clr = 11) or (clr = 12)
                then clr := 2
             else if (clr = 5) or (clr = 18)
                then clr := 3
             else
                clr := 0;

            var cl := colors[clr];
            var r := cl.R/255;
            var g := cl.G/255;
            var b := cl.B/255;
            _shader.SetParameter('color', r, g, b);
            // устанавливаем квадратик в сетке:
            quadr.Position := new Vector2f(PEN_WIDTH + col *
                                           (QSIZE + PEN_WIDTH),
                                            PEN_WIDTH + row *
                                           (QSIZE+PEN_WIDTH));
            // рисуем квадратик:
            wind.Draw(quadr, states);
        end;
        // обновляем сцену:
        wind.Display();
end;
```

Зато вызов процедуры в **главном блоке** сократился до предела:

```
// чертим таблицу:
Draw();
```

А **мозаика** получилась не очень выразительная:

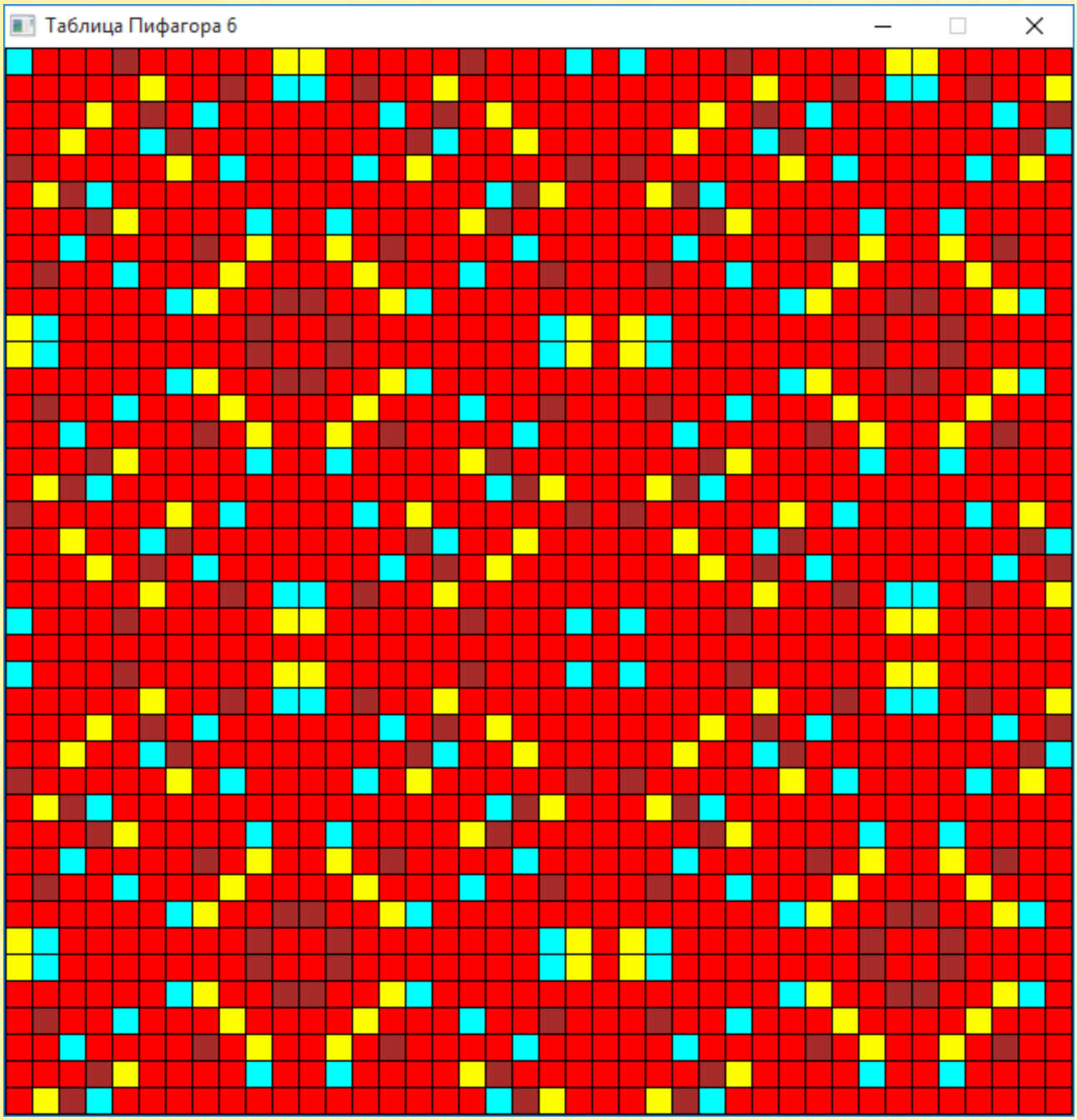


Таблица Пифагора 6

Проект *Таблица Пифагора 8*

Седьмая задача из *Науки и жизни* (Задача 11, *Рис. 15* из журнала *Мир информатики*):

Кружевной монохромный узор (рис.7) возникает, если во всей таблице закрасить одинаковым цветом только числа, дающие остатки, сравнимые с одним и тем же натуральным числом.

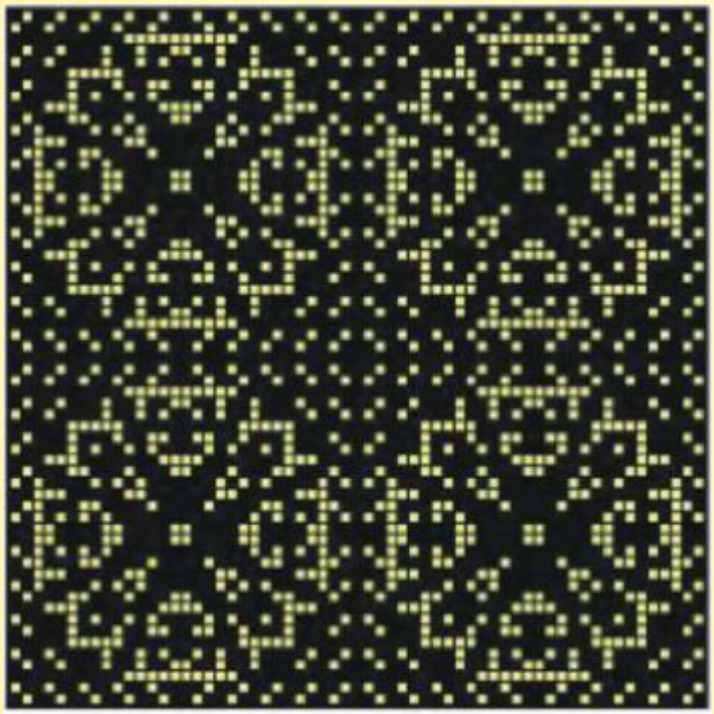


Рис. 7. «Монохромный узор» при k=31.

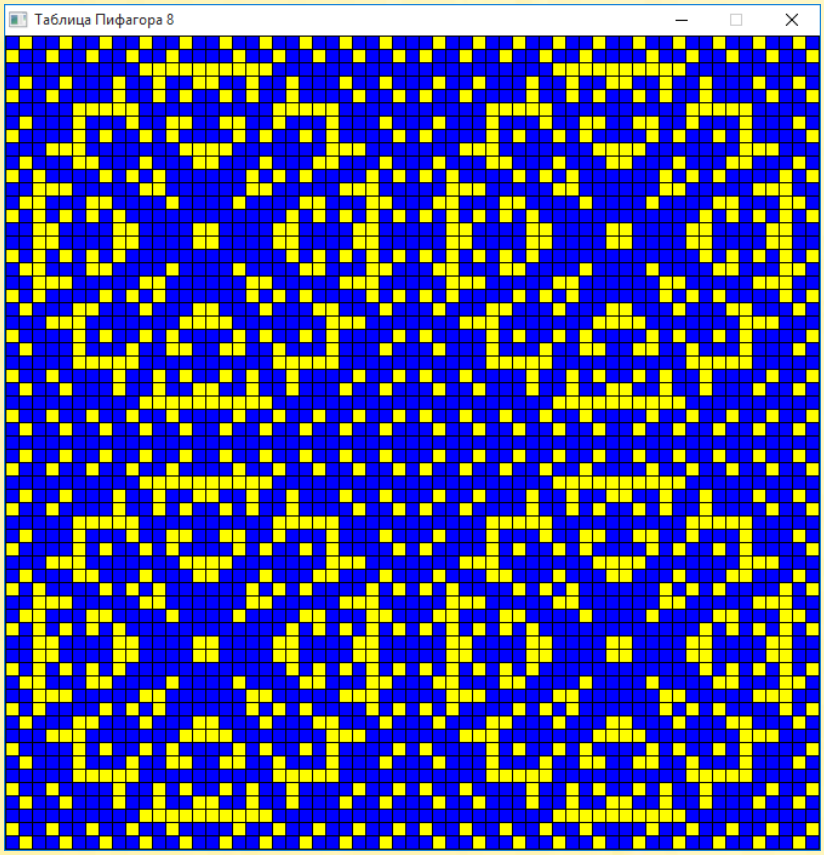
В данном случае k = 31, а закрашиваются только числа, удовлетворяющие условию: i * j mod 31 mod 3 = 2.

```
const
   // размер клеток в пикселях:
   QSIZE = 10;
   // число клеток по горизонтали:
   NCOL = 61;
   // число клеток по вертикали:
   NROW = 61;
```

```
// цвета клеток:
colors := new Color[] (ColorEx.Yellow,
                       Color.Blue,
                       ColorEx.Brown,
                       ColorEx.Cyan,
                       ColorEx.Green,
                       ColorEx.Red);
```

```
// РИСУЕМ ТАБЛИЦУ
procedure Draw(k, n: integer);
begin
   // функция раскраски:
   var fclr: (integer,integer,integer,integer) -> integer :=
             (c,r,k,n) -> (r + 1) * (c + 1) mod k mod n = 2 ? 0 : 1;
   // чертим таблицу:
   Draw(31, 3
```

На этот раз мозаика удалась на славу:



Проект *Шейдерная* чёрная *дыра*

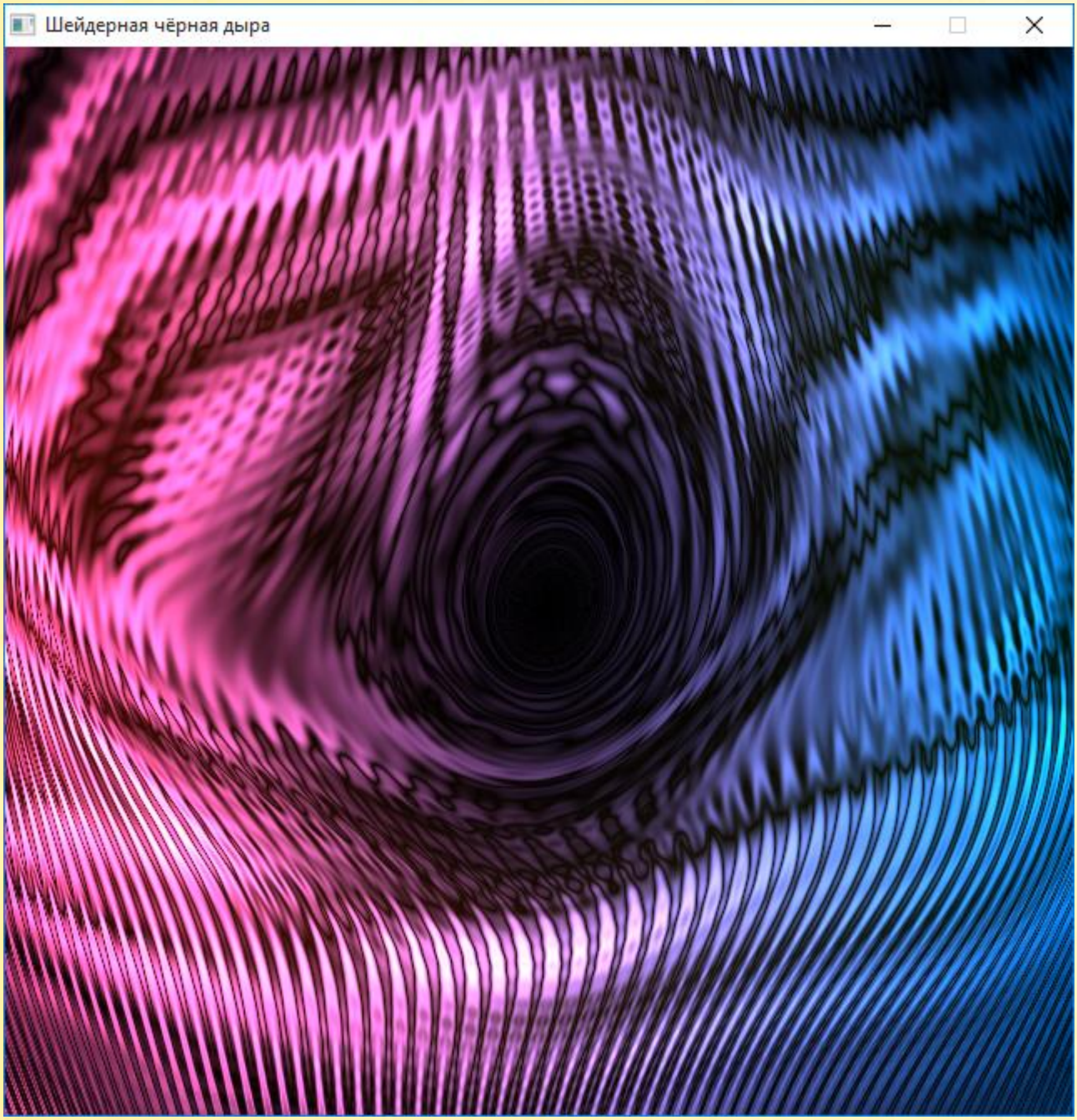
Интересные шейдеры можно найти повсюду! Например в программе Processing. Мы рассмотрим только три из них.

Главный блок программы одинаков для всех шейдеров. Нужно только изменить заголовок окна и название шейдера:

```
begin
   // создаём главное окно:
   CreateWindow(WIDTH, HEIGHT, 'Шейдерная чёрная дыра');

   if (not Shader.IsAvailable) then
   begin
      wind.SetTitle('ШЕЙДЕРЫ НЕ ПОДДЕРЖИВАЮТСЯ!');
   end;
   // загружаем шейдер:
   var _shader := new Shader(nil, 'Media/monjori.frag');

   // создаём текстуру:
   var texture := new Texture(WIDTH, HEIGHT);
   // создаём спрайт из текстуры:
   var sprite := new Sprite(texture);

   // устанавливаем значение переменной resolution:
   _shader.SetParameter('resolution', new Vector2f(WIDTH, HEIGHT));

   // создаём шейдерный режим:
   var states := new RenderStates(_shader);

   var time := 0.0;
   // игровой цикл:
   while (wind.IsOpen) do
   begin
      // вызываем все обработчики событий:
      wind.DispatchEvents();
      _shader.SetParameter('time', time);
      // печатаем спрайт с шейдером:
      wind.Draw(sprite, states);

      time += 0.1;
```

```
      // показываем на экране:
      wind.Display();
   end;
end.
```

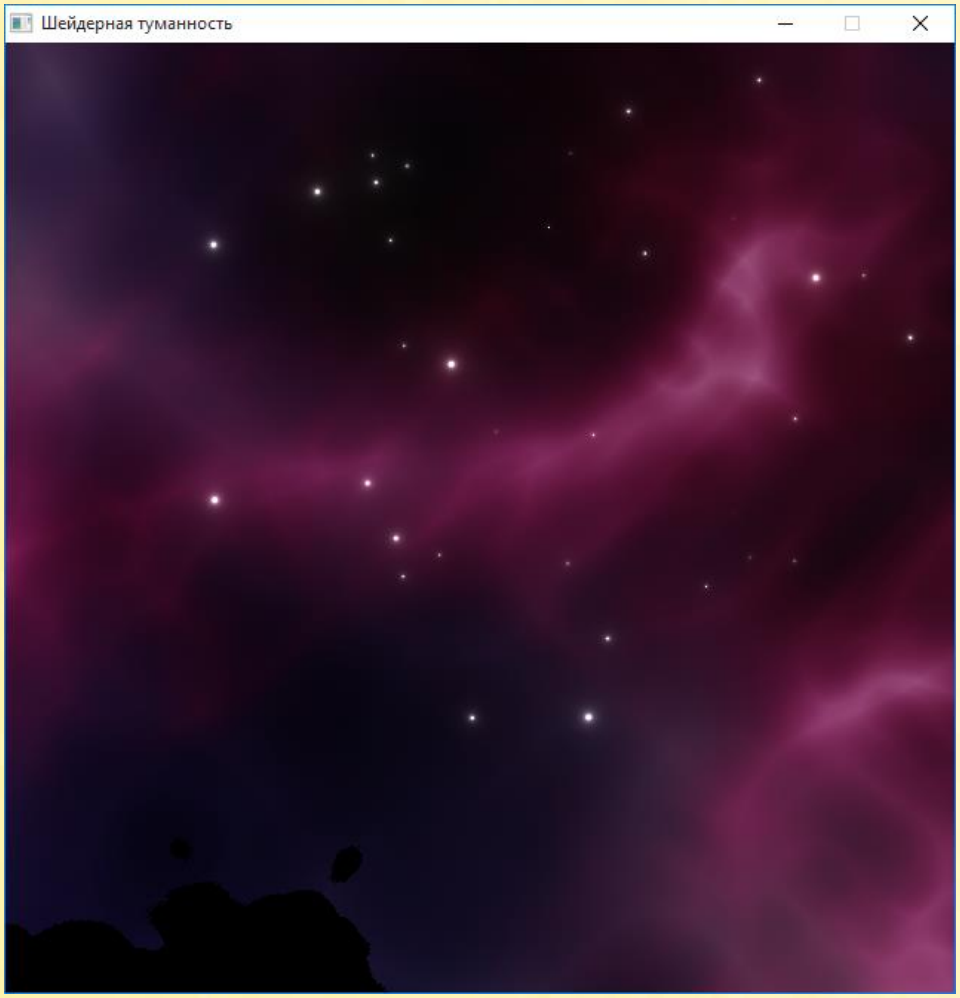
Все шейдерные картинки изменяются со временем, поэтому оценить их в полной мере можно только в работающей программе.



Проект *Шейдерная туманность*

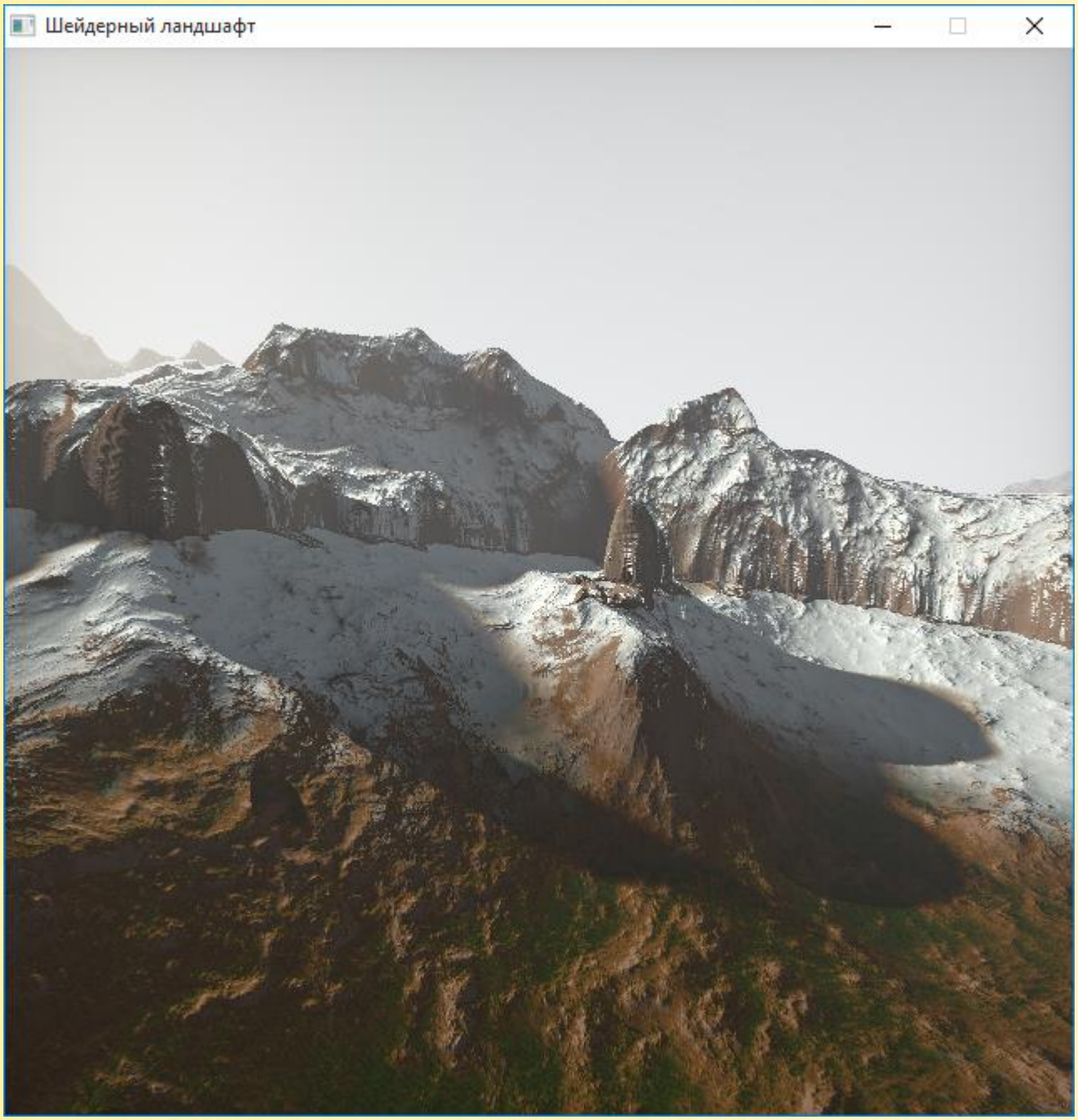
А этот шейдер создаёт вполне реалистическую картину космического путешествия:

```
begin
   // создаём главное окно:
   CreateWindow(WIDTH, HEIGHT, 'Шейдерная туманность');
   if (not Shader.IsAvailable) then
   begin
      wind.SetTitle('ШЕЙДЕРЫ НЕ ПОДДЕРЖИВАЮТСЯ!');
   end;
   // загружаем шейдер:
   var _shader := new Shader(nil, 'Media/nebula.frag');
   ...
```



Проект *Шейдерный ландшафт*

Самый сложный шейдер. Он умеет создавать горный ландшафт! И даже с тенями:



Класс RenderTexture (Текстура)

RenderTexture – это класс, описывающий текстуру для внеэкранной визуализации.

Проект *Класс RenderTexture*

Разберём на конкретном примере процесс визуализации на текстуре типа *RenderTexture*.

Создаём внеэкранную текстуру по размерам окна приложения:

```
begin
   // создаём главное окно:
   CreateWindow(WIDTH, HEIGHT, 'Класс RenderTexture');

   // создаём внеэкранную текстуру:
   var rtex := new RenderTexture(WIDTH, HEIGHT);
```

Закрашиваем её жёлтым цветом:

```
// закрашиваем жёлтым цветом:
rtex.Clear(Color.Yellow);
```

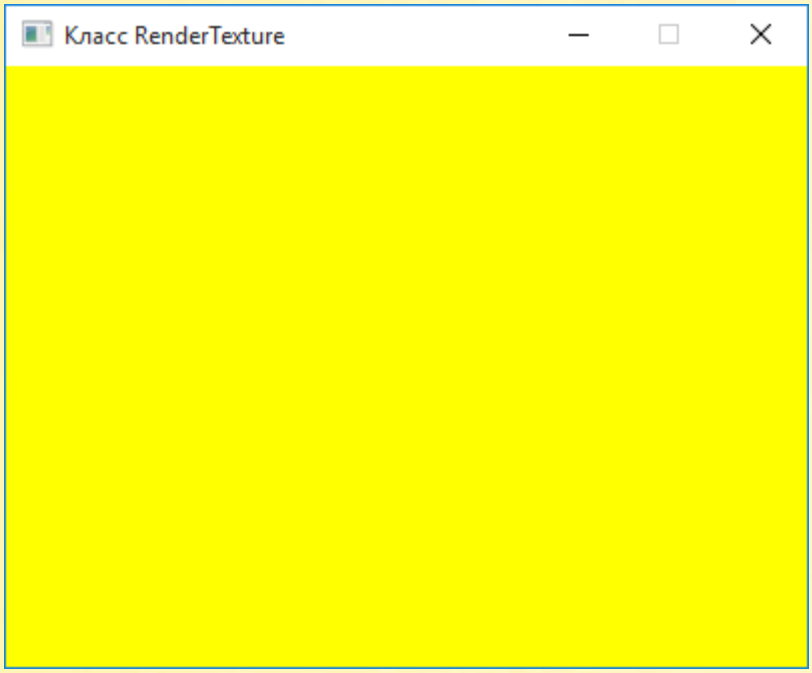
Из текстуры внеэкранной поверхности создаём спрайт:

```
// создаём спрайт из текстуры:
var sprite := new Sprite(rtex.Texture);
```

Рисуем спрайт на экране:

```
// показываем спрайт на экране:
wind.Draw(sprite);
wind.Display();
```

Жёлтая текстура занимает всю клиентскую часть окна приложения:



Давайте нарисуем на ней красный прямоугольник.

```
// создаём прямоугольник:
var rect := new RectangleShape(new Vector2f(200, 100));
// задаём его координаты:
rect.Position := new Vector2f(0, 0);
// и цвет:
rect.FillColor := Color.Red;
// рисуем прямоугольник на внеэкранной поверхности:
rtex.Draw(rect);
```

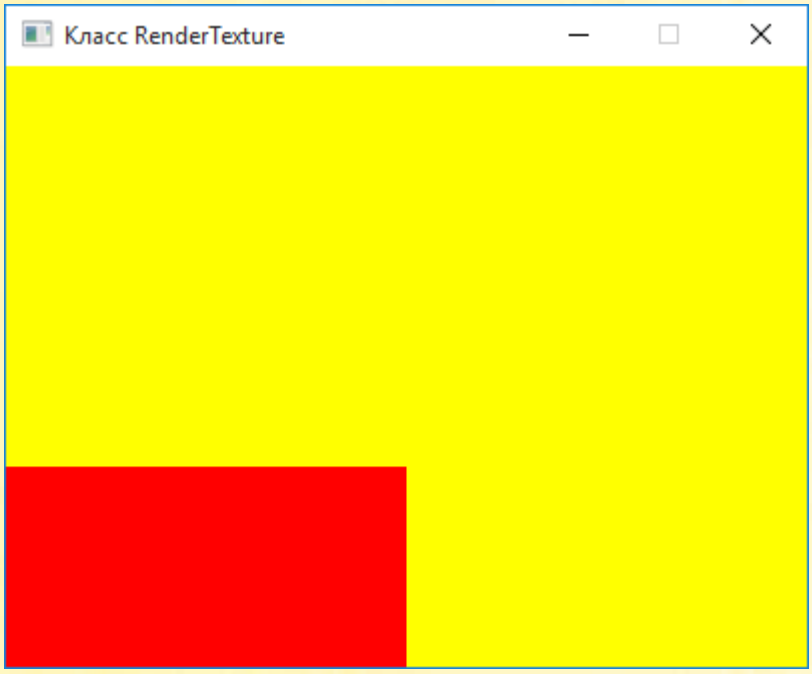
У каждого графического объекта также имеется метод Draw для «саморисования» на заданной поверхности типа *RenderWindow* или *RenderTexture*, поэтому прямоугольник можно нарисовать на внеэкранной поверхности и так:

```
rect.Draw(rtex, RenderStates.Default);
```

Или так:

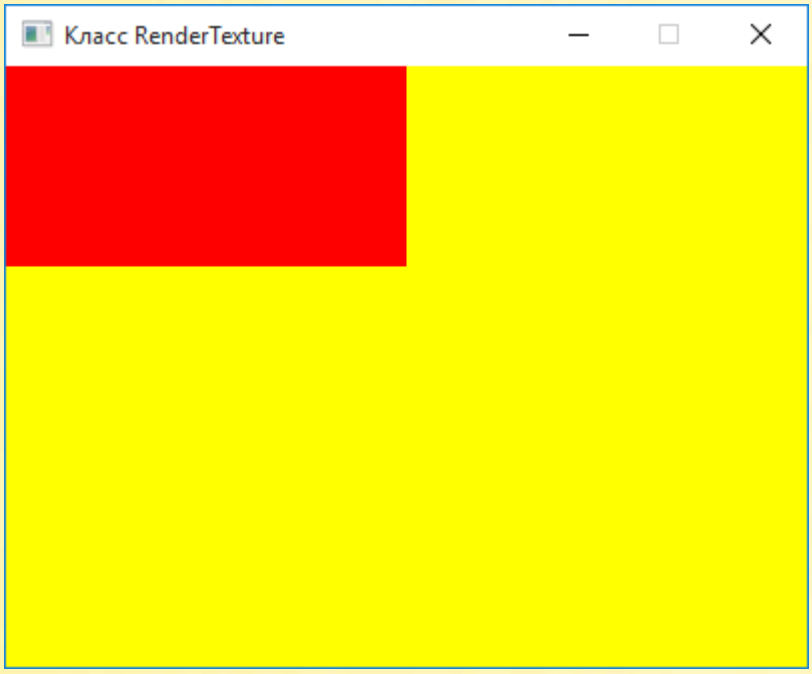
```
var state := new RenderStates(BlendMode.None);
rect.Draw(rtex, state);
```

В любом случае на экране появится вот такая **картинка**:



В отличие от окна, на текстуре начало координат находится в левом **нижнем** углу, а ось **Y** направлена **вверх**. Точно так же располагается и локальная система координат самого прямоугольника, поэтому, чтобы нарисовать прямоугольник в левом верхнем углу, нужно задать его координаты так:

```
rect.Position := new Vector2f(0, HEIGHT - 100);
```



Создадим также текст:

```
// загружаем шрифт с диска:
fnt := new Font('Media/segmono.ttf');
// создаём текст:
txt := new Text('Класс RenderTexture', fnt, 30);
//цвет текста:
txt.Color := Color.Black;
// координаты текста:
txt.Position := new Vector2f(20, 0);
// стиль текста:
txt.Style := Text.Styles.Bold;
```

И напечатаем его на внеэкранной поверхности:

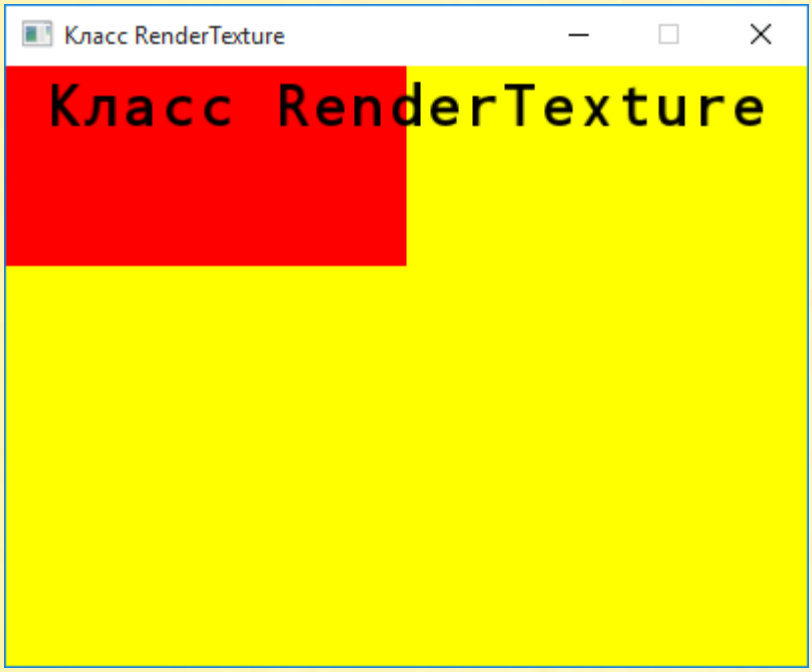
```
// печатаем текст:
rtex.Draw(txt);
```

Хорошо видно, что текст напечатан вверх ногами:

Нужно перевернуть текстуру, чтобы она была напечатана верно:

```
// с переворотом -->
// рисуем прямоугольник на внеэкранной поверхности:
rtex.Draw(rect);
// печатаем текст:
rtex.Draw(text);
```

```
// создаём спрайт из текстуры:
var img := rtex.Texture.CopyToImage();
img.FlipVertically();
var tmp := new Texture(img);
var sprite := new Sprite(tmp);

// показываем спрайт на экране:
wind.Draw(sprite);
wind.Display();
```



Класс Clock (Часы)

Класс Clock описывает системные часы для отсчёта прошедшего времени.

Класс *Clock* и структура *Time* используются для вычисления промежутков времени. В новом проекте часы послужат нам **секундомером** в незатейливой игре.

Проект *Быстрота реакции*

В глобальной переменной _record будем запоминать *лучшее время* игрока:

```
// часы:
_clock: Clock;
// время:
time := 0;
// рекордное время:
_record := 1000000;
```

Через случайный промежуток времени на чёрном-чёрном экране:

```
const
   // размеры окна:
   WIDTH = 600;
   HEIGHT = 400;
   // цвет фона:
   BACKCOLOR = Color.Black;

var
   // окно приложения:
   wind: RenderWindow := nil;
```

появляется в случайном месте красный квадрат:

```
// квадрат:
WRECT = 100;
```

```
HRECT = 100;

rand := new Random();
rect := new RectangleShape(new Vector2f(WRECT, HRECT));
```

Для печати информации на экране нам потребуются глобальные переменные fnt и txt:

```
// шрифт:
fnt: Font := nil;
// текст:
txt: Text := nil;
```

В процедуре CreateWindow загружаем шрифт и создаём часы:

```
// СОЗДАЁМ ОКНО
procedure CreateWindow(width, height: uint64;
                       title: string := 'SFML Window';
                       style: Styles := Styles.Close);
begin
   // создаём окно приложения:
   wind := new RenderWindow(new VideoMode(width, height),
                            title, style);
   wind.SetVerticalSyncEnabled(true);
   wind.SetFramerateLimit(60);


   // загружаем шрифт с диска:
   fnt := new Font('Media/verdana.ttf');

   // создаём часы:
   _clock := new Clock();

   // добавляем метод для закрывания окна:
   wind.Closed += OnClosed;
   // обработчик нажатия кнопки мышки:
   wind.MouseButtonPressed += Window_MouseButtonPressed;
end;
```

Когда на экране появится квадрат, игрок должен МОЛНИЕНОСНО нажать кнопку мышки, чтобы остановить время.

Переходим в **главный блок**.

Понятно, что квадрат должен появляться на экране **неожиданно**. Для этого в игровом цикле мы запишем в переменную **delay** случайную задержку в миллисекундах.

```
begin
   // создаём главное окно:
   CreateWindow(WIDTH, HEIGHT, 'Быстрота реакции');

   // случайная задержка перед появлением квадрата:
   var delay := 100;
```

Весь игровой процесс можно разбить на ситуации, или положения, в которых программа должна действовать по-разному. Мы записали их в **перечисление GameStatus**:

```
type
   Text = SFML.Graphics.Text;

   // игровая ситуация:
   GameStatus = ( GS_NEWGAME, GS_START, GS_PRESS, GS_MESSAGE,
                  GS_SHOW, GS_WAIT );

   gs := GameStatus.GS_NEWGAME;
```

Перед началом игрового цикла игровой статус устанавливаем в **GS_NEWGAME**, то есть начинаем новую игру:

```
// начинаем новую игру:
gs := GameStatus.GS_NEWGAME;
```

Запускаем **игровой цикл**!

```
// игровой цикл:
while (wind.IsOpen) do
begin
   // вызываем все обработчики событий:
   wind.DispatchEvents();

   // цвет фона окна:
   wind.Clear(BACKCOLOR);
```

Начало – традиционное, а затем программа в операторе *switch* выбирает текущую игровую ситуацию. После старта программы это `GameStatus.GS_NEWGAME`:

```
case (gs) of
   GameStatus.GS_NEWGAME: begin
      CreateRect();
      delay := rand.Next(1000, 4000);
      _clock.Restart();
      gs := GameStatus.GS_START;
   end;
```

Здесь мы должны создать **квадрат**:

```
// СОЗДАЁМ КВАДРАТ В СЛУЧАЙНОМ МЕСТЕ ОКНА
procedure CreateRect();
begin
   rect.FillColor := Color.Red;
   var x := rand.Next(WIDTH - WRECT);
   var y := rand.Next(HEIGHT - HRECT);
   rect.Position := new Vector2f(x, y);
end;
```

Его **координаты** выбираем так, чтобы он *целиком* был виден на экране. Но квадрат появится позже, поэтому пока мы его не увидим.

Задаём случайную задержку **delay** и переводим программу в режим `GameStatus.GS_START`.

Здесь мы ждём, пока истечёт время *delay*, снова запускаем отсчёт времени и переходим в режим GameStatus.GS_PRESS:

```
GameStatus.GS_START: begin
   wind.Clear(BACKCOLOR);
   if (_clock.ElapsedTime.AsMilliseconds() > delay) then
   begin
        _clock.Restart();
        gs := GameStatus.GS_PRESS;
   end;
end;
```

Пришло время показать квадрат на экране:

```
// показываем квадрат:
GameStatus.GS_PRESS: begin
   wind.Draw(rect);
   _clock.Restart();
   gs := GameStatus.GS_WAIT;
end;
```

Мы снова сбрасываем показания часов, чтобы теперь они отсчитывали время реакции игрока.

В режиме GameStatus.GS_WAIT показываем квадрат:

```
GameStatus.GS_WAIT: begin
   wind.Draw(rect);
end;
```

Пока игрок не нажмёт кнопку мышки:

```
// НАЖИМАЕМ КНОПКУ МЫШКИ
procedure Window_MouseButtonPressed(sender: Object; e: MouseButtonEven-
tArgs);
begin
    // игра закончена:
```

```
    if (gs = GameStatus.GS_SHOW) then
        exit;

    // кнопка нажата раньше:
    if (gs <> GameStatus.GS_WAIT) then
    begin
       // задаём большое время:
       time := 1000000;
       // создаём текст:
       txt := new Text('ФАЛЬСТАРТ!', fnt, 48);
    end
    else
    begin
       // время реакции:
       time := _clock.ElapsedTime.AsMilliseconds() div 2;
       var message := time.ToString();
       // создаём текст:
       txt := new Text('ВРЕМЯ (мс) = ' + message, fnt, 48);
    end;

    // цвет текста:
    txt.Color := Color.Red;
    // координаты текста:
    txt.Position := new Vector2f((WIDTH - txt.GetLocalBounds().Width) / 2,
                                 (HEIGHT - txt.GetLocalBounds().Height) / 2);
    // стиль текста:
    txt.Style := Text.Styles.Bold;
    // небольшой поворот:
    txt.Rotation := -1.1;

    // показываем сообщение на экране:
    _clock.Restart();
    gs := GameStatus.GS_SHOW;
end;
```

В режиме GameStatus.GS_SHOW показываем сообщение о фальстарте или время реакции:

```
GameStatus.GS_SHOW: begin
    wind.Draw(txt);
```

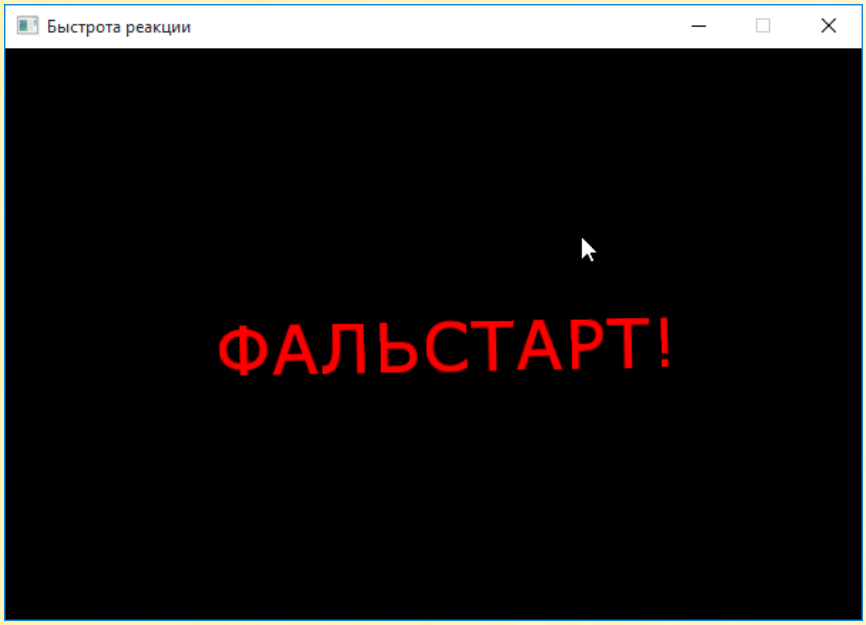
Если установлен новый рекорд, то сохраняем снимок с экрана на диске:

```
GameStatus.GS_SHOW: begin
   wind.Draw(txt);
   if (time < _record) then
   begin
      _record := time;
      delay := 5000;
      wind.Capture().SaveToFile('record.jpg');
   end
   else
      delay := 2000;
```

Когда закончится время показа сообщения, стираем квадрат и надпись и начинаем новую игру:

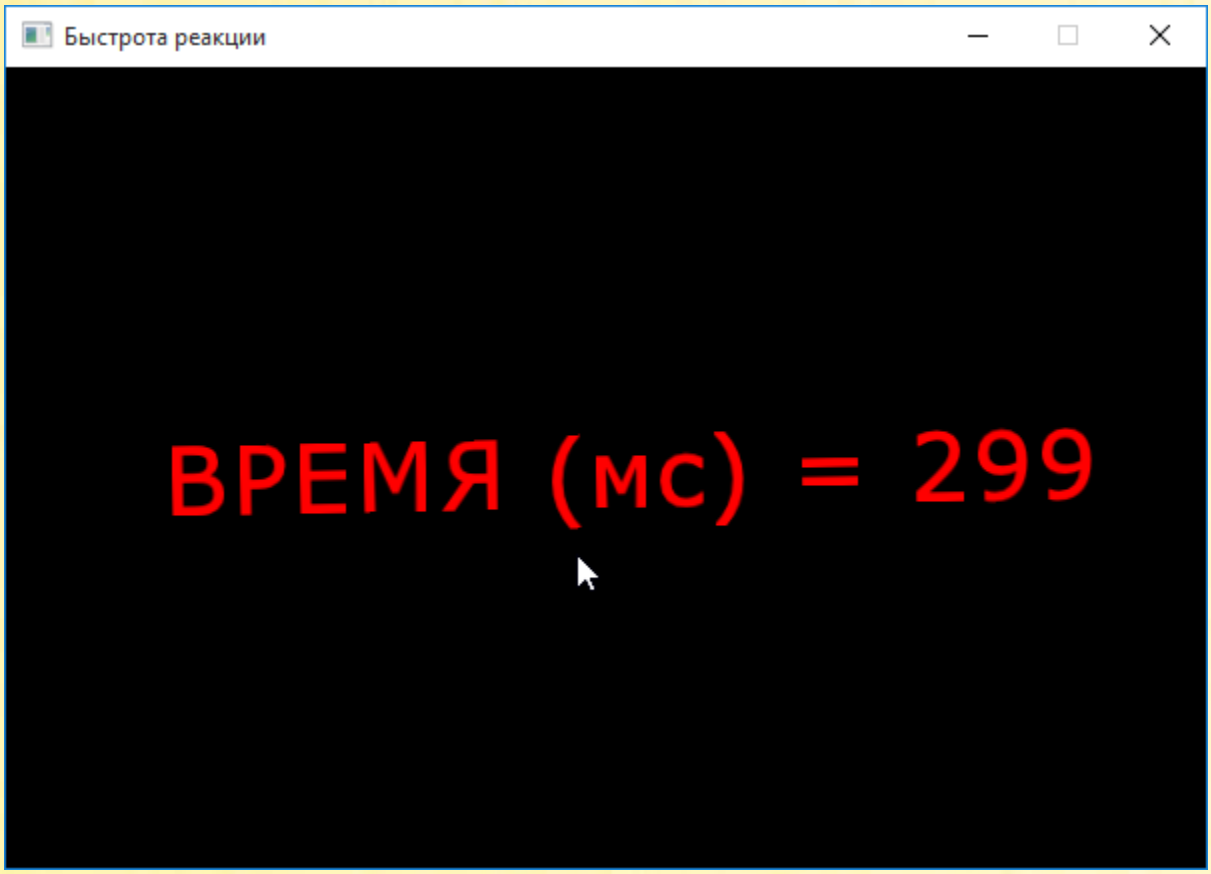
```
                if (_clock.ElapsedTime.AsMilliseconds() > delay) then
                begin
                   wind.Clear(BACKCOLOR);
                   gs := GameStatus.GS_NEWGAME;
                end;
             end;
          end;
        // показываем на экране:
        wind.Display();
    end;
end.
```

А теперь «слайды».



Извините, погрячился:

Неспешная быстрота:



Градиенты и математика

Градиентом в компьютерной графике называют плавный переход одного цвета в другой. Художники используют термин «растяжка».

Цвет в компьютерной графике определяется тремя составляющими: *R(ed) – G(reen) – B(lue)*, или по-русски: Красная – Зелёная – Синяя. Каждая из составляющих имеет 256 оттенков. Всего можно получить 256 x 256 x 256 = 16 777 216 цветов.

В простейшем градиенте значения двух цветовых составляющих фиксированы, а третья изменяется от **0** до **255**, что и создаёт плавный переход цвета. Если градиент **вертикальный**, то составляющая изменяется по высоте окна. Если **горизонтальный** – по ширине. Если **радиальный** – от центра окна к его границам.

Если высота окна равна **h** пикселей, то можно составить уравнение для вычисления значения составляющей. Путь это будет зелёная составляющая **g**:

g = 255*y/h

Здесь: **y** – это вертикальная координата горизонтального отрезка прямой

Обозначим:

k = 255/h

Тогда **g = ky**.

Мы получили формулу для **прямой пропорциональности**.

Формула весьма скучная, особенно если сравнить её с графическим представлением в образе градиента.

По этой формуле вычисляется переход от синего цвета к циану.

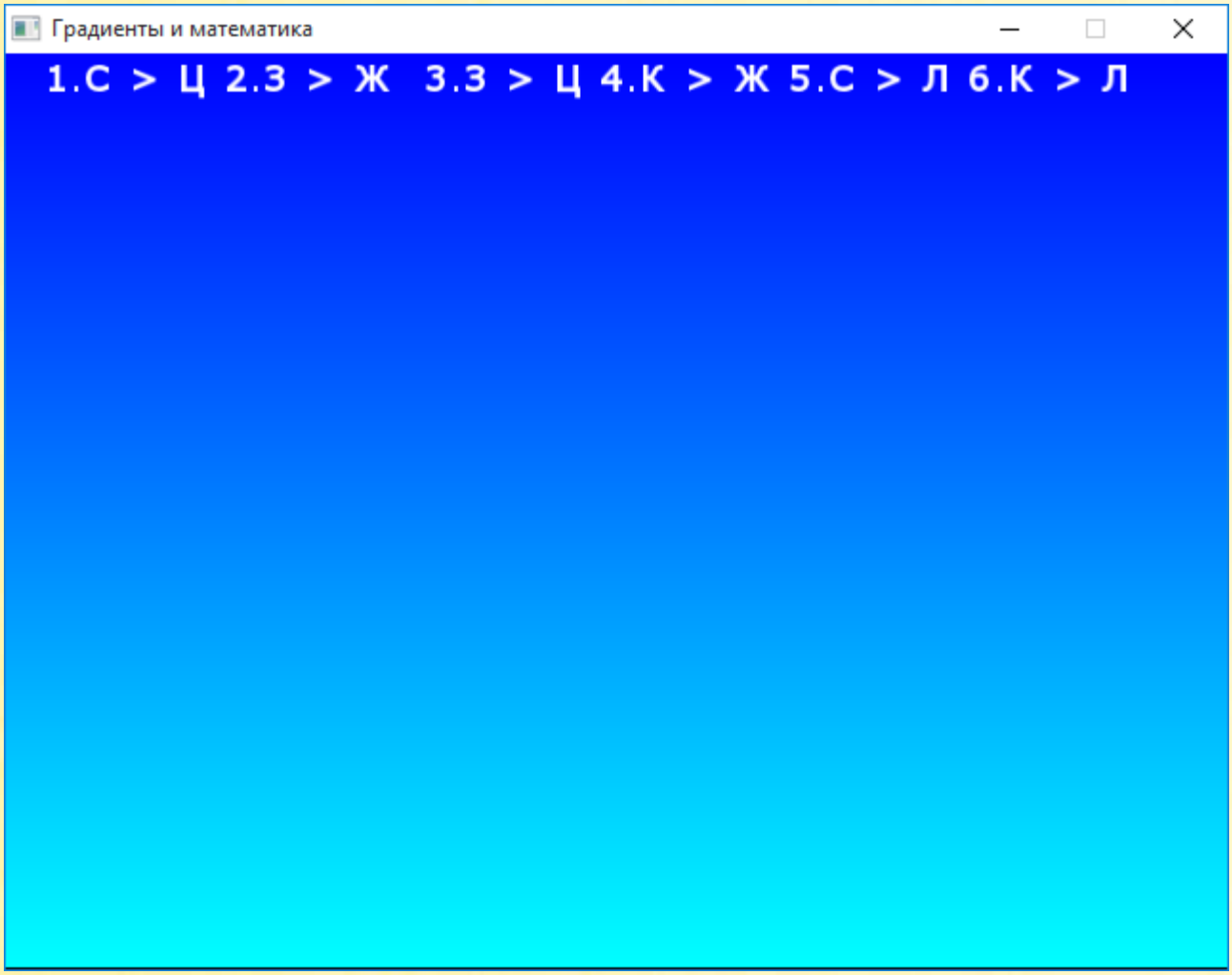
Составляющие синего цвета:

```
Blue → Red = 0, Green = 0, Blue = 255
```

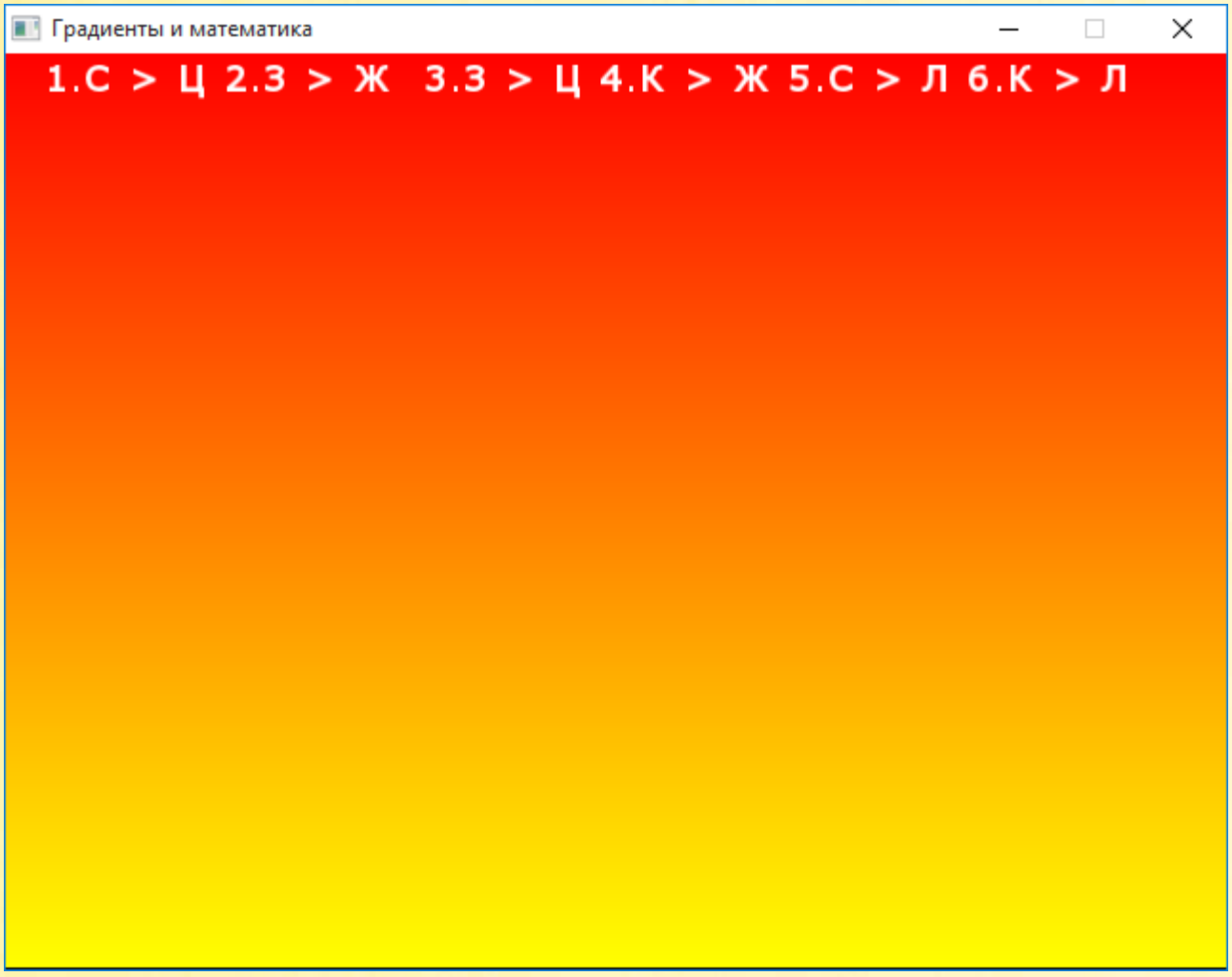
Составляющие цвета циан:

```
Cyan → Red = 0, Green = 255, Blue = 255
```

Таким образом, для рисования вертикального градиента нужно увеличивать значение зелёной составляющей по высоте от 0 до 255. При этом цвет будет изменяться от синего к циану:



Градиенты – это простой, но наглядный пример прямой пропорциональности, выраженной средствами компьютерной графики:


Градиенты и математика
1.С > Ц 2.З > Ж 3.З > Ц 4.К > Ж 5.С > Л 6.К > Л

Библиотека RVLib

Во многих проектах мы пользовались библиотекой RVLib. Пришло время рассказать о ней.

Для графических проектов нам, естественно, понадобится библиотека sfmlnet-graphics-2.dll, которую нужно добавить к модулю:

```
unit RVLib;

{$reference 'sfmlnet-graphics-2.dll'}
uses System;
uses SFML.Graphics;
```

В библиотеку следует включать процедуры и функции, которые часто используются в программах. Например, функцию для создания случайного цвета:

```
var
  rand := new Random();
```

```
// ПОЛУЧАЕМ СЛУЧАЙНЫЙ ЦВЕТ
function GetRandomColor(alpha: boolean := false): Color;
begin
    // красная составляющая:
    var r := rand.Next(256);
    // зелёная составляющая:
    var g := rand.Next(256);
    // синяя составляющая:
    var b := rand.Next(256);
    // прозрачность:
    var a := rand.Next(256);

    if (not alpha) then
        Result := new Color(r,g,b)
    else
        Result := new Color(r, g, b, a);
end;
```

Часто нужно переводить цвет из одной цветовой модели в другую. Мы использовали 2 функции для конвертирования цвета HSV в RGB. Они действуют практически одинаково, но лучше применить в программе ту и другую, чтобы сравнить результаты:

```
// КОНВЕРТИРУЕМ ЦВЕТ HSV В RGB
function HSV2RGB(h: double; s: double := 100; v:double := 100): Color;
begin
   s := s / 100.0;
   v := v / 100.0;
   var r := 0.0;
   var g := 0.0;
   var b := 0.0;

   //оттенки серого:
   if (s = 0) then
   begin
      r := v;
      g := v;
      b := v;
   end
   else
   begin
      var Hi := Ceil(Math.Floor(h / 60));
      var f := h / 60 - Hi;

      //цветовые оси:
      var p := v * (1 - s);
      var q := v * (1 - (s * f));
      var t := v * (1 - (s * (1 - f)));

      case Hi of
         0: begin
               r := v;
               g := t;
               b := p;
            end;
         1: begin
               r := q;
               g := v;
               b := p;
            end;
```

```
            2: begin
                  r := p;
                  g := v;
                  b := t;
               end;
            3: begin
                  r := p;
                  g := q;
                  b := v;
               end;
            4: begin
                  r := t;
                  g := p;
                  b := v;
                end;
            5: begin
                  r := v;
                  g := p;
                  b := q;
                end;
         end;
      end;
      Result := new Color(Ceil(r * 255),
                          Ceil(g * 255),
                          Ceil(b * 255));
end;
```

```
// КОНВЕРТИРУЕМ ЦВЕТ HSV В RGB
function Clamp(i: double): byte;
begin
   if (i < 0) then
      Result := 0
   else if (i > 255) then
      Result := 255
   else
      Result := Ceil(i);
end;

function HsvToRgb(_h: double; S: double := 1.00; V: double := 1.00):
Color;
begin
// h = 0-360
// s,v = 0-1
```

```
// r,g,b = 0-255

   var H := _h;
   while (H < 0) do H += 360;
   while (H >= 360) do H -= 360;
   var R, G, B: double;
   if (V <= 0) then
   begin
       R := 0;
       G := 0;
       B := 0;
   end
   else if (S <= 0) then
   begin
       R := V;
       G := V;
       B := V;
   end
   else
   begin
      var hf := H / 60.0;
      var i := Ceil(Math.Floor(hf));
      var f := hf - i;
      var pv := V * (1 - S);
      var qv := V * (1 - S * f);
      var tv := V * (1 - S * (1 - f));
      case (i) of
         0: begin
               R := V;
               G := tv;
               B := pv;
            end;
         1: begin
               R := qv;
               G := V;
               B := pv;
            end;
         2: begin
               R := pv;
               G := V;
               B := tv;
            end;
         3: begin
               R := pv;
               G := qv;
```

```
                B := V;
            end;
          4: begin
                R := tv;
                G := pv;
                B := V;
            end;
          5: begin
                R := V;
                G := pv;
                B := qv;
            end;

          6: begin
                R := V;
                G := tv;
                B := pv;
            end;
          -1: begin
                R := V;
                G := pv;
                B := qv;
              end;
           // ошибка:
           else begin
                R := V;
                G := V;
                B := V;
              end;

        end;

        var _r := Clamp(R * 255);
        var _g := Clamp(G * 255);
        var _b := Clamp(B * 255);
        Result := new Color(_r,_g,_b);
    end;
end;
```

Графическая библиотека SFML бедна стандартными цветами, поэтому её нужно дополнить.

Их можно поместить как статические переменные в класс ColorEx:

```
type
  ColorEx = class
     //function GetGold := new Color($ff, $d7, 0);
     //public class property Gold : Color read GetGold;

     public class AliceBlue := new Color($f0, $f8, $ff);
     public class AntiqueWhite := new Color($fa, $eb, $d7);
     public class Aqua := new Color(0, $ff, $ff);
     public class Blue := new Color($0, $0, $ff);
     public class Brown := new Color($A5, $2A, $2A);
     public class Cyan := new Color(0, $ff, $ff);
     public class Gold := new Color($ff, $d7, 0);
     public class Green := new Color(0, $80, $00);
     public class Red := new Color($ff, $00, 0);
     public class Yellow := new Color($ff, $ff, 0);
  end;
```

Либо использовать **поля**. Этот вариант длиннее, но более надёжен.

В качестве самостоятельного задания можете перенести все функции в этот класс.