LXF134:Perl

GD и Perl Создайте динамические изображения в ваших сценариях для Web, и не только

Содержание

[убрать]

Perl: Создаем изображения

Perl Марко Фиоретти

Perl Михаил Смирнов

Часть 1: Язык, созданный для обработки текстов, может неплохо справляться и с графикой. **Михаил Смирнов** объяснит, что к чему.

Как известно, язык программирования *Perl* идеально приспособлен для многочисленных приложений по обработке текста. Однако возможности *Perl* этим не исчерпываются. Применение графических модулей в *Perl* позволит вам быстро рисовать изображения, состоящие из линий, окружностей и текста, манипулировать цветом, вырезать и вставлять другие изображения, выполнять заливку, сохранять на диске файлы изображений в различных форматах, манипулировать видеоданными этих файлов. Наибольший интерес представляет использование графических модулей *Perl* в web-приложениях. Графическая библиотека *GD* имеет интуитивно понятный интерфейс, позволяющий быстро и компактно реализовать пользовательский код программ на *Perl* даже начинающим программистам.

Эффективность использования графического модуля *GD* объясняется просто: библиотека *GD* написана на языке *C* и совместима со всеми основными платформами (Linux, Windows,

...). Интерфейс между *Perl* и библиотекой *GD* обеспечивает модуль *GD.pm*, который использует *GD* как базис для создания графических классов в *Perl. GD.pm* является самозагружаемым интерфейсным модулем, с помощью которого можно создавать изображения на лету, модифицировать существующие файлы изображений и синтезировать новые.

Привет, *GD*!

Библиотека *GD* позволяет создавать цветные рисунки, используя большое количество графических примитивов, и выводить их в различных графических форматах. Модуль *GD* определяет три основных класса:

- класс изображений **GD::Image**, который предназначен для захвата видеоданных изображения и выполняет вызов методов графических примитивов;
- класс шрифтов **GD::Font**, который берет статические шрифты и использует их для воспроизведения текста в графическом виде;
- класс примитивов полигона **GD::Polygon**, применяемый для сохранения списка вершин многоугольников при рисовании сторон полигона и обеспечивающий их воспроизведение в виде изображения.

Обобщенный подход использования графического интерфейса *GD* выглядит следующим образом. На первом этапе, при вызове (инициализации) метода **GD::Image::new(Wx,Hy)**, создается новое пустое изображение, ширина и высота которого задаются параметрами **Wx** и **Hy**. В результате вызова метода будут возвращены видеоданные изображения. Другие методы класса позволяют инициализировать изображение из уже существующих файлов, которые необходимо предварительно прочитать с диска.

На втором этапе, при вызове метода **colorAllocate()** выполняется добавление цвета к цветовой таблице изображения. Входными параметрами вызова является тройка основных цветов: красный, зеленый и синий, которые задаются целыми числами в диапазоне значений от 0 до 255. Метод возвращает индекс цвета цветовой таблицы изображения.

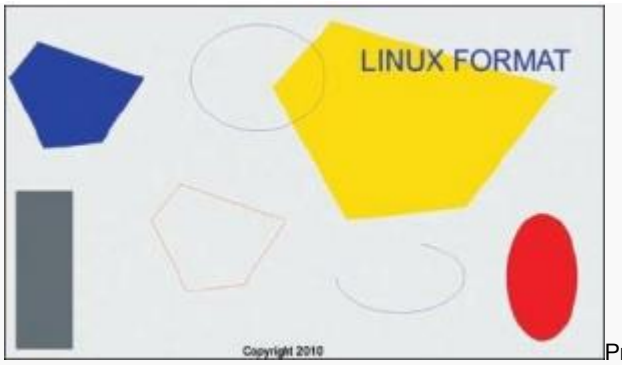

Рис. 1. Пример рисования простых объектов.

После этого, на третьем этапе, можно сделать некоторый рисунок, используя набор графических примитивов класса **GD::Image** и набор методов класса **GD::Polygon**. Полигоны (многоугольники) создаются с помощью метода **new()**, а добавление новых вершин полигона возложено на метод **addPt()**.

И, наконец, когда рисунок готов, вы можете преобразовать его, например, в графический формат GIF, с помощью метода **gif()**. Метод будет возвращать двоичное содержимое изображения в формате GIF. Обычно полученный результат выводится в браузер или сохраняется на диске в виде GIF-файла. В качестве примера приведем скрипт, который рисует простые фигуры, манипулирует ими самими и их цветом, а также рисует текст с помощью статических шрифтов:

```
#!/perl/bin/perl
use GD;
$im = new GD::Image(425,250); #--Инициализация изображения
$black = $im->colorAllocate(0, 0, 0); #--Создаем цветовую палитру
$red = $im->colorAllocate(255, 0, 0);
$blue = $im->colorAllocate(0,0,255);
$yellow = $im->colorAllocate(255,250,0);
$bg = $im->colorAllocate(240,240,240);
$gray = $im->colorAllocate(128,128,128);
$im->fill(0,0,$bg); #--Делаем заливку фона
$im->rectangle(0,0,424,249,$black); #--Рисуем границы изображения
$poly = new GD::Polygon; #--Инициализация полигона
$poly->addPt(5,50);
$poly->addPt(25,25);
$poly->addPt(100,50);
$poly->addPt(70,95);
$poly->addPt(30,100);
$im->filledPolygon($poly,$blue); #--Создание полигона
$poly->offset(100,100); #--Смещение полигона
$im->polygon($poly,$red); #--Рисуем только контур
$poly->map($poly->bounds,190,10,390,150); #--Изменение масштаба полигона в
2 раза
$im->filledPolygon($poly,$yellow);
$im->arc(180,50,95,75,0,360,$blue); #--Рисуем эллипсы и дуги
$im->arc(280,190,90,50,290,180,$blue);
$im->arc(380,190,50,90,0,360,$blue);
$im->fill(380,190,$red);
$im->filledRectangle(10, 130, 50, 240, $gray); #--Рисуем четырехугольник
$im->rectangle(10,130,50,240,$black);
$string=”LINUX FORMAT”;
$im->string(gdMediumBoldFont,260,25,$string,$blue); #--Задаем статические
шрифты
$im->string(gdSmallFont,170,235,”Copyright 2010”,$black);
print “Content-Type: image/gif\n\n”;
binmode STDOUT;
print $im->gif(); #--Преобразование данных в формат GIF и вывод
```

Показанный выше код позволяет web-программисту познакомиться на практике с основными командами для рисования и может послужить основой для самостоятельного создания более серьезных программ. Полученный результат можно посмотреть на рис. 1.

Основные методы

В иерархии интерфейсов *GD* наибольшее распространение получили два типа версий для работы с растровой графикой. Первый тип поддерживает манипулирование и преобразование изображений в формате GIF. Второй тип интерфейсов *GD* позволяет работать с такими популярными графическими форматами, как PNG и JPEG.

Для закрепления полученных выше практических навыков рассмотрим несколько «теоретических» моментов инициализации основных методов *GD*. Вызов метода **GD::Image::new(Wx,Hy)** возвращает новое пустое изображение, например:

```
$myImage = new GD::Image(100,100) || die;
```

Метод создает пустой бланк изображения размером 100 × 100 точек. Если размер опущен, он будет составлять 64 × 64 точек. Метод **GD::Image::newFromGif(FILE)** создает изображение на основе данных файла в формате GIF, предварительно прочитанного с диска. Параметр **FILE** представляет собой указатель файла (дескриптор). Дескриптор файла указывается как один из параметров функции **open()**. Если открытие файла было успешным, то вызов возвращает инициализированное изображение, которое затем может быть подвергнуто различным манипуляциям. Например:

```
open (MyGIF,"images/ballon.gif") || die;
$myImage = newFromGif GD::Image(MyGIF) || die;
close MyGIF;
```

MyGIF здесь – указатель файла. Вызов метода **GD::Image::gif** возвращает видеоданные изображения в формате GIF. Затем их можно вывести в браузер и/или записать в файл. Например:

```
open (MyGIF,">images/ballon.gif");
binmode(MyGIF);
print MyGIF $im->gif;
close (MyGIF);
print "Content-Type: image/gif\n\n";
binmode STDOUT;
print $im->gif;
```

Метод **GD::Image::colorAllocate(R,G,B)** предназначен для задания и управления цветом с помощью RGB-спецификации – цветовых компонент R (красный), G (зеленый) и B (синий), и в результате возвращает индекс цветовой таблицы. Пример задания желтого цвета:

```
$yellow = $myImage->colorAllocate(255,255,0);
```

Полученные индексы впоследствии можно использовать для задания общего фона изображения или для заливки фигур определенным цветом. Метод **GD::Image::getPixel(x,y)** возвращает индекс цветовой таблицы под определенной точкой изображения с координатами (x,y). Вызов этого метода может быть скомбинирован с функцией **rgb()**, если вам потребуется восстановить список RGB-компонент под выбранной точкой. Например:

```
$index = $myImage->getPixel(20,100);
($r,$g,$b) = $myImage->rgb($index);
```

Метод **GD::Image::transparent(colorIndex)** отмечает индекс **colorIndex** как прозрачный. Участки изображения, нарисованные этим цветом, будут невидимы. Метод применяется для создания прозрачных фонов изображений в Web. Прозрачным в изображении может быть только один цвет. Ниже, например, им будет белый:

```
open(myGIF,"transptest.gif");
$im = newFromGif GD::Image(myGIF);
$white = $im->colorClosest(255,255,255);
$im->transparent($white);
print $im->gif;
close myGIF;
```

Команды рисования

Эти команды хотя и не занимают в иерархии методов *GD* особого положения, но выделяются своей выраженной функциональностью. Интуитивная простота команд (методов) позволит вам быстро нарисовать целый ряд примитивов или запрограммировать произвольный рисунок. Рассмотрим ряд наиболее употребляемых методов.

Метод **GD::Image::setPixel(x,y,color)'** ***создает точку с координатами (x,y) с определенным цветовым индексом*** *color****. Он ничего не возвращает. Система координат начинается в верхнем левом углу и заканчивается в нижнем правом. Нетрудно представить, что растровый перебор координат при наличии функциональной зависимости цветового индекса*** *color* от координат (x,y) может использоваться для синтеза изображений. Реализуем пример кода для формирования двумерного распределения функции **w=sin(x,y)**:

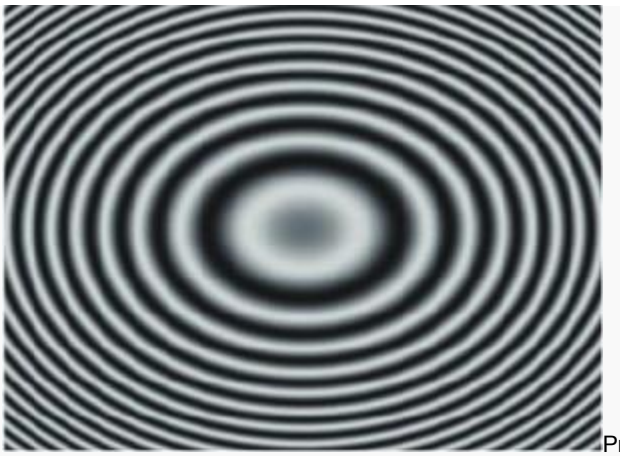

Рис. 2. Пример синтеза изображения.

```
$im = new GD::Image($X,$Y);
$arg=$period*4*$pi*$pi;
for($j=0;$j<$Y; $j++){
 $y = ($j-$Y/2)/$Y;
 $phy = $y*$y;
for($i=0;$i<$X; $i++){
 $x = ($i-$X/2)/$X;
 $phx = $x*$x;
 $w = int($a0 + $amp*sin($arg*($pi - $phx - $phy)));
 $gray = $im->colorResolve($w,$w,$w);
$im->setPixel($i,$j,$gray);
}}
```

Метод **colorResolve(r,g,b)** возвращает индекс цвета, который точно соответствует указанным красным, зеленым и синим компонентам. Если такой цвет не находится в цветной таблице, то метод добавляет цвет в таблицу и возвращает его индекс. На рис. 2 представлен результат синтеза изображения, созданного в соответствии свыше приведенным кодом.

Метод **GD::Image::rectangle(x1,y1,x2,y2,color)** рисует четырехугольник, стороны которого имеют определенный цвет **color**. Координаты (x1,y1) и (x2,y2) являются верхним левым и правым нижним углами, соответственно. Пример рисования квадрата размером 90 × 90 точек:

```
$myImage->rectangle(10,10,100,100,$blue);
```

Метод **GD::Image::polygon(polygon,color)** рисует многоугольник с заданным цветом **color**, при этом число вершин не должно быть менее трех. Если последняя вершина полигона не создана, метод будет закрыт. Пример рисования треугольника:

```
$poly = new GD::Polygon;
$poly->addPt(50,0);
$poly->addPt(99,99);
$poly->addPt(0,99);
$myImage->polygon($poly,$blue);
```

Метод **GD::Image::line(x1,y1,x2,y2,color)** рисует линию с определенным цветом **color** от точки с координатами (x1,y1) до точки с координатами (x2,y2). Пример рисования диагональной синей линии от точки с координатами (7,7) до точки с координатами (154,154):

```
$myImage->line(7,7,154,154,$blue);
```

Метод рисования **line()** очень часто применяется для построения графиков функции одной переменной. Фрагмент кода скрипта, реализующий основной цикл формирования линии графика, а также оси абсцисс и значений переменной по оси абсцисс, представлен ниже:

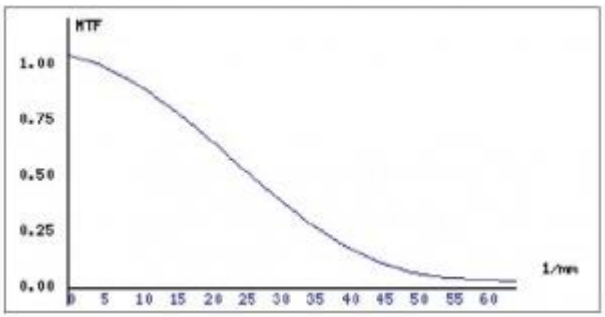

Рис. 3. Пример построения графика функции одной переменной.

```
$f=0;
for($i=0;$i<$N-1;$i++){
$im->line($x,$y[$i],$x+$dx,$y[$i+1],$blue);
$im->line($x,$yh,$x+$dx,$yh,$black);
$im->string(gdTinyFont, $x, $yh+2, $f, $blue);
$f +=5;
$x=$x+$dx;
}
```

Пример рисования графика функции одной переменной показан на рис. 3.

С помощью метода **GD::Image::arc(cx,cy,Wx,Hy,start,end,color)** можно рисовать дуги и эллипсы. Координаты (cx,cy) являются центром эллипса (дуги), а параметры (Wx,Hy) являются шириной и высотой эллипса, соответственно. Часть эллипса покрывается дугой, длина которой управляется параметрами **start** и **end**. Эти параметры задаются в угловой мере от 0 до 360 градусов. Пример рисования полного эллипса:

```
$myImage->arc(100,100,50,35,0,360,$blue);
```

Метод **GD::Image::fill(x,y,color)** служит для заливки областей изображения с заданным цветом **color**. Цвет будет распространяться по изображению, начиная от точки (x,y) до границ объекта, на которых появляется разностный цвет. Пример заливки синим цветом, начиная с точки с координатами x=50 и y=40:

```
$myImage->rectangle(10,10,100,100,$black);
$myImage->fill(50,40,$blue);
```

Копирование

Команды копирования изображений выделяются своей функциональностью и важны с точки зрения создания эффективного кода при рисовании множества повторяющихся участков изображения. При написании кода программы этот метод позволит запомнить некоторую область изображения аналогично тому, как это делается с обычными массивами данных в *Perl*.

Метод **GD::Image::copy(sourceImage,dstX,dstY,srcX,srcY,Wx,Hy)** предназначен для копирования прямоугольных областей от одного изображения к другому. Изображение-источник задается с помощью параметра **sourceImage**. Параметры (srcX,srcY) задают верхний левый угол прямоугольника в изображении-источнике, а (Wx, Hy) указывают ширину и высоту области копирования. Параметры (dstX,dstY) управляют координатами точки, в которой будет начинаться наложение копии на изображение-получатель. Вызов метода возвращает данные изображения-получателя. Пример копирования области **$srcImage** размером 25 × 25 точек в область **$myImage** с началом в точке (10,10) показан ниже:

```
$myImage = new GD::Image(100,100);
$srcImage = new GD::Image(50,50);
$myImage->copy($srcImage,10,10,0,0,25,25);
```

С изобразительной точки зрения, более интересным методом является метод **GD::Image::copyMerge(sourceImage,dstX,dstY,srcX,srcY,Wx,Hy,opaque)**, который отличается от предыдущего всего лишь одним параметром **opaque**. Он может пригодиться для создания коллажей. С помощью параметра opaque в диапазоне значений от 0 до 100 задается степень прозрачности одного изображения относительно другого, например:


Рис. 4. Пример слияния двух изображений с эффектом прозрачности.

```
$image1->copyMerge($image2,$left,$top,0,0,$wx,$hy, 60);
```

На рис. 4 показан пример слияния с эффектом прозрачности двух изображений одного размера. При этом прозрачность второго изображения относительно первого составляет 60 %.

Выводим текст

Интерес web-программистов к рисованию текста связан со множеством задач, возникающих при создании графических элементов web-страниц: от случайного набора символов в задаче авторизации пользователя (создание так называемой Captcha), до рисования водяных знаков в изображениях.

Интерфейс *GD* поддерживает минимальный набор статических шрифтов и позволяет рисовать символы и строки текста в графическом виде. Вывод может выполняться как в горизонтальном направлении, так и в вертикальном. Доступными для рисования шрифтами являются следующие: TinyFont, SmallFont, MediumBoldFont, LargeFont и GiantFont. Каждый из этих шрифтов может быть импортирован как глобальная константа или как пакет объектов класса **GD::Font**, например: **GD::Font::gdTinyFont**.

Непосредственно для рисования применяется два основных метода: **string()** и **stringUp()**. Метод **string(font,x,y,string,color)** рисует строку текста, начиная с позиции (x,y), в заданном цвете и выбранном шрифте. Метод **stringUp(font,x,y,string,color)** рисует текст с поворотом на 90 градусов в направлении, противоположном вращению часовой стрелки.

Рассмотренные выше шрифты имеют весьма ограниченный диапазон размеров, и при этом только средний шрифт позволяет выделять символы как жирные. Другой существенный недостаток статических шрифтов – это ограничение по локализации исключительно латиницей.

Использование динамических шрифтов TrueType обеспечивает при рисовании текста практически неограниченные возможности, и прежде всего, локализацию под любой тип кодировки, применяемый в web-программировании. Метод для вызова шрифтов имеет вид **GD::Image->stringTTF()**. Разнообразие же существующих шрифтов снимает ограничения на их доступность. Один из вариантов вызова метода **stringTTF()** имеет вид:

Рис. 5. Пример рисования текста динамическими шрифтами.

```
$image->stringTTF($fgcolor,$fontname,$ptsize,$angle,$x,$y,$str),
```

где аргументы команды – это:

- индекс цвета **fgcolor** для рисования строки текста,
- путь **fontname** к выбранному файлу шрифта (.ttf),
- желаемый размер **ptsize** шрифта,
- угол поворота **angle** в радианах,
- декартовы координаты x,y начала рисования,
- строка текста **str**.

На рис. 5 показан результат рисования строки текста несколькими кириллическими шрифтами: buaril.ttf, bukursv.ttf, buroman.ttf, butimit.ttf, pushkin.ttf, butimes.ttf.

Форматы PNG и JPEG

Как было отмечено выше, второй тип интерфейсов *GD* предоставляет возможность использовать графические форматы PNG и JPEG. Для работы с графическим форматом PNG применяется метод **GD::Image->newFromPng($file)**, который с помощью дескриптора файла может создавать изображения в этом формате. Если открытие файла было успешным, то вызов возвращает инициализированное изображение, например:

```
open (myPNG,”test.png”) || die;
$myImage = newFromPng GD::Image(\*myPNG) || die;
close myPNG;
print “Content-type: image/png\n\n”;
binmode STDOUT;
print $im->png();
```

Существует другой вариант вызова с непосредственным указанием файла изображения:

```
$myImage = newFromPng GD::Image('test.png')
```

Для работы с форматом JPEG применяется метод **GD::Image->newFromJpeg($file)**. Этот метод будет создавать изображение в формате JPEG после чтения файла с диска в том же формате:

```
$quality=75;
$image = GD::Image->newFromJpeg($file);
print “Content-type: image/jpeg\n\n”;
binmode STDOUT;
```

```
print $image->jpeg($quality);
```

Отличительная особенность метода **jpeg()** – наличие параметра качества **$quality**, задаваемого как целое в диапазоне значений от 0 до 100. Чем больше его значение, тем выше качество изображения, и, естественно, больше размер файла на диске. Если он не задан, по умолчанию будет выбрано среднее качество.