

ГЛАВА 17



Введение в компьютерную графику

... как будто волшебный фонарик освещает изнутри
образы на экране...

Т. С. Элиот, «Песнь любви Альфреда Пруффрока»

Графика — одна из наиболее быстро развивающихся отраслей компьютерной индустрии. Известно, что 70% информации человек воспринимает визуально, поэтому графика — это важнейший компонент для взаимодействия человека с компьютером.

Для программирования компьютерной графики часто используются такие классы геометрии, как точки, двумерные размеры, прямоугольники, а также специальные классы для хранения цветовых значений.

Классы геометрии

Группа классов геометрии ничего не отображает на экране. Основное их назначение состоит в задании расположения, размеров и в описании формы объектов.

Точка

Для задания точек в двумерной системе координат служат два класса: QPoint и QPointF. При этом точка обозначается парой чисел X и Y , где X — горизонтальная, а Y — вертикальная координаты. В отличие от обычного расположения координатных осей, при задании координат точки в Qt обычно подразумевается, что ось Y смотрит вниз (рис. 17.1).

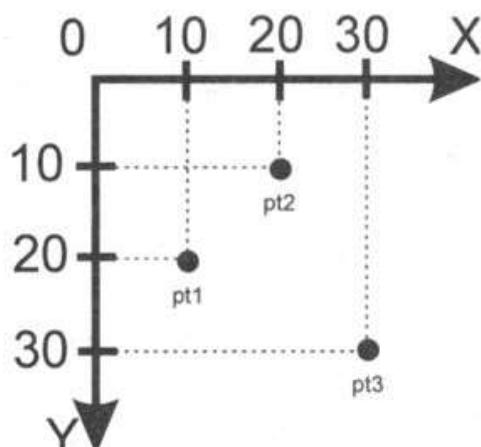


Рис. 17.1. Создание и сложение точек

Класс QPoint описывает точку с целочисленными координатами, а QPointF — с вещественными. Интерфейс обоих классов одинаков, в него входят методы, позволяющие проводить различные операции с координатами, например сложение и вычитание с координатами другой точки. При сложении/вычитании точек выполняется попарное сложение/вычитание их координат X и Y . Следующий пример складывает две точки: pt1 и pt2 (см. рис. 17.1):

```
QPoint pt1(10, 20);
QPoint pt2(20, 10);
QPoint pt3; // (0, 0)
pt3 = pt1 + pt2;
```

Объекты точек можно умножать и делить на числа. Например:

```
QPoint pt(10, 20);
pt *= 2; // pt = (20, 40)
```

Для получения координат точки (X , Y) реализованы методы `x()` и `y()` соответственно. Изменяются координаты точки с помощью методов `setX()` и `setY()`.

Можно получать ссылки на координаты точки, чтобы изменять их значения. Например:

```
QPoint pt(10, 20);
pt.rx() += 10; // pt = (20, 20)
```

Объекты точек можно сравнивать друг с другом при помощи операторов `==` (равно) и `!=` (не равно). Например:

```
QPoint pt1(10, 20);
QPoint pt2(10, 20);
bool b = (pt1 == pt2); // b = true
```

Если необходимо проверить, равны ли координаты X и Y нулю, то вызывается метод `isNull()`. Например:

```
QPoint pt; // (0, 0)
bool b = pt.isNull(); // b = true
```

Метод `manhattanLength()` возвращает сумму абсолютных значений координат X и Y . Например:

```
QPoint pt(10, 20);
int n = pt.manhattanLength(); // n = 10 + 20 = 30
```

Этот метод был назван в честь улиц Манхэттена, расположенных перпендикулярно друг к другу. Возвращаемое значение является грубым приближением к $\sqrt{X^2 + Y^2}$.

Двумерный размер

Классы QSize и QSizeF служат для хранения соответственно целочисленных и вещественных размеров. Оба класса обладают одинаковыми интерфейсами. Структура их очень похожа на QPoint, так как хранит две величины, над которыми можно проводить операции сложения/вычитания и умножения/деления.

Классы QSize и QSizeF, как и классы QPoint, QPointF, предоставляют операторы сравнения `==`, `!=` и метод `isNull()`, возвращающий значение `true` в том случае, если высота и ширина равны нулю.

Для получения ширины и высоты вызываются методы `width()` и `height()`. Изменить эти параметры можно с помощью методов `setWidth()` и `setHeight()`. При помощи методов `rwidth()` и `rheight()` получают ссылки на значения ширины и высоты. Например:

```
QSize size(10, 20);
int n = size.rwidth()++; // n = 11; size = (11, 20)
```

Помимо них, класс предоставляет метод `scale()`, позволяющий изменять размеры оригинала согласно переданному в первом параметре размеру. Второй параметр этого метода управляет способом изменения размера (рис. 17.2), а именно:

- ◆ `Qt::IgnoreAspectRatio` — изменяет размер оригинала на переданный в него размер;
- ◆ `Qt::KeepAspectRatio` — новый размер заполняет заданную площадь, насколько это будет возможно с сохранением пропорций оригинала;
- ◆ `Qt::KeepAspectRatioByExpanding` — новый размер может находиться за пределами переданного в `scale()`, заполняя всю его площадь.

Согласно рис. 17.2, изменение размеров `size1`, `size2` и `size3` может выглядеть следующим образом:

```
QSize size1(320, 240);
size1.scale(400, 600, Qt::IgnoreAspectRatio); // => (400, 600)
```

```
QSize size2(320, 240);
size2.scale(400, 600, Qt::KeepAspectRatio); // => (400, 300)
```

```
QSize size3(320, 240);
size3.scale(400, 600, Qt::KeepAspectRatioByExpanding); // => (800, 600)
```

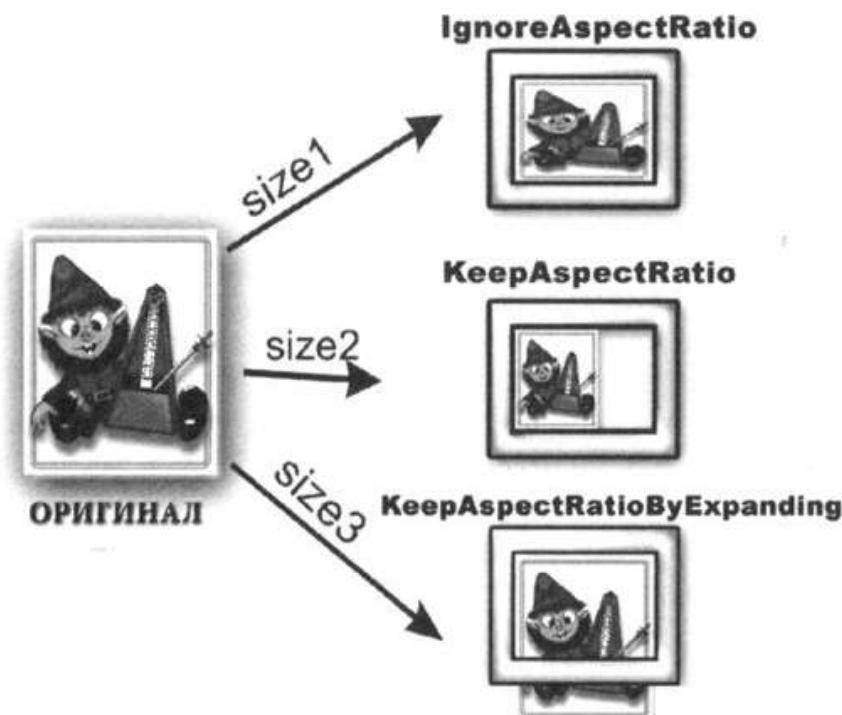


Рис. 17.2. Изменение размеров оригинала

Прямоугольник

Классы `QRect` и `QRectF` служат для хранения целочисленных и вещественных координат прямоугольных областей (точка и размер) соответственно. Задать прямоугольную область можно, например, передав в конструктор точку (верхний левый угол) и размер. Область, приведенная на рис. 17.3, создается при помощи следующих строк:

```
QPoint pt(10, 10);
QSize size(20, 10);
QRect r(pt, size);
```

Получить координаты X левой грани прямоугольника или Y верхней можно при помощи методов `x()` или `y()` соответственно. Для изменения этих координат нужно воспользоваться методами `setX()` и `setY()`.

Размер получают с помощью метода `size()`, который возвращает объект класса `QSize`. Можно просто вызвать методы, возвращающие составляющие размера: ширину `width()` и высоту `height()`. Изменить размер можно методом `setSize()`, а каждую его составляющую — методами `setWidth()` и `setHeight()`.

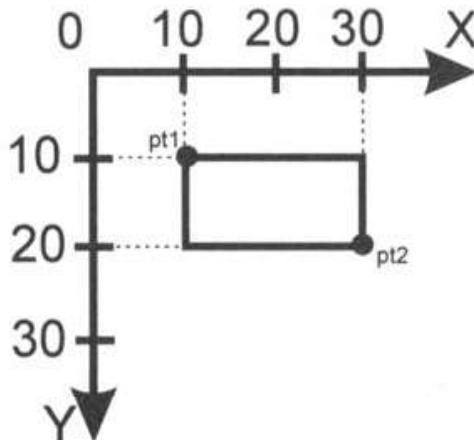


Рис. 17.3. Задание прямоугольной области точкой и размером

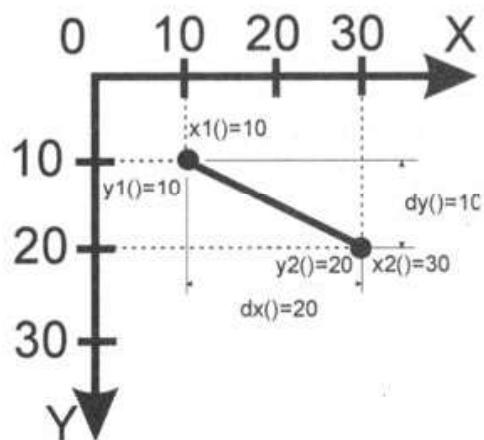


Рис. 17.4. Задание прямой линии двумя точками

Прямая линия

Классы `QLine` и `QLineF` описывают прямую линию или, правильней сказать, отрезок на плоскости в целочисленных и вещественных координатах соответственно. Позиции начальной точки можно получить при помощи методов `x1()` и `y1()`, а конечной — `x2()` и `y2()`. Аналогичного результата можно добиться вызовами `p1()` и `p2()`, которые возвращают объекты класса `QPoint/QPointF`, описанные ранее. Методы `dx()` и `dy()` возвращают величины горизонтальной и вертикальной проекций прямой на оси X и Y соответственно. Прямую, показанную на рис. 17.4, можно создать при помощи одной строки кода:

```
QLine line(10, 10, 30, 20);
```

Оба класса: `QLine` и `QLineF` — предоставляют операторы сравнения `==`, `!=` и метод `isNull()`, возвращающий логическое значение `true` в том случае, когда начальная и конечная точки не установлены.

Многоугольник

Многоугольник (или полигон) — это фигура, представляющая собой замкнутый контур, образованный ломаной линией. В Qt эту фигуру реализуют классы `QPolygon` и `QPolygonF`, в целочисленном и вещественном представлении соответственно. По своей сути эти классы являются массивами точек `QVector<QPoint>` и `QVector<QPointF>`. Самый простой способ инициализации объектов класса полигона — это использование оператора потока вывода `<<`. Треугольник представляет собой самую простую форму многоугольника (рис. 17.5), а его создание выглядит следующим образом:

```
QPolygon polygon;
polygon << QPoint(10, 20) << QPoint(20, 10) << QPoint(30, 30);
```

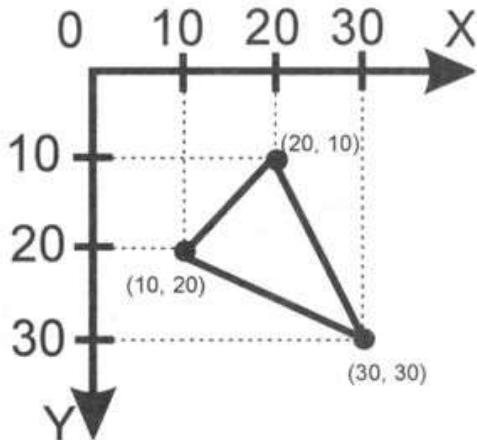


Рис. 17.5. Задание треугольника тремя точками

Цвет

Цвета, которые вы видите, есть не что иное, как свойство объектов материального мира, воспринимаемое нами как зрительное ощущение от воздействия света, имеющего различные электромагнитные частоты. На самом деле, человеческий глаз в состоянии воспринимать очень малый диапазон этих частот. Цвет с наибольшей частотой, которую в состоянии воспринять глаз, — фиолетовый, а с наименьшей — красный. Но даже в таком небольшом диапазоне находятся миллионы цветов. Одновременно человеческий глаз может воспринимать около 10 тысяч различных цветовых оттенков.

Цветовая модель — это спецификация в трехмерной или четырехмерной системе координат, которая задает все видимые цвета. В Qt поддерживаются цветовые модели: *RGB* (Red, Green, Blue — красный, зеленый, голубой), *RGBA* (Red, Green, Blue, Alpha — красный, зеленый, голубой, прозрачный), *CMYK* (Cyan, Magenta, Yellow и Key color — голубой, пурпурный, желтый и «ключевой» черный цвет), *HSV* (Hue, Saturation, Lightness — оттенок, насыщенность, светлота) и *HSV* (Hue, Saturation, Value — оттенок, насыщенность, значение).

Класс `QColor`

С помощью класса `QColor` можно сохранять цвета во всех упомянутых цветовых моделях. Его определение находится в заголовочном файле `QColor`. Объекты класса `QColor` можно сравнивать при помощи операторов `==` и `!=`, присваивать и создавать копии.

Цветовая модель RGB

Наиболее чувствителен глаз к зеленому цвету, потом следует красный, а затем — синий. На этих трех цветах и построена модель RGB (Red, Green, Blue — красный, зеленый, синий). Пространство цветов задает куб, длина ребер которого равна 255 (рис. 17.6) в целочисленном числовом представлении (либо единице в вещественном представлении).

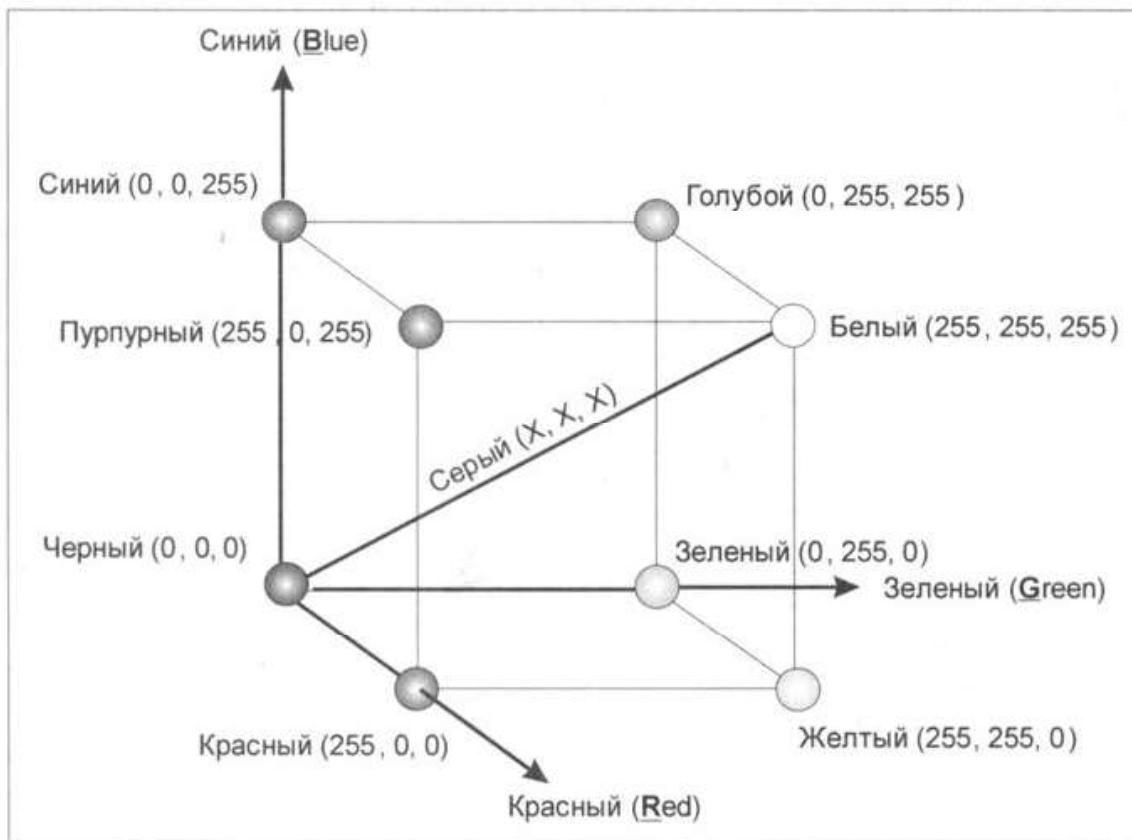


Рис. 17.6. Цветовая модель RGB

Как видно из рис. 17.6, цвет задается сразу тремя параметрами. Первый параметр задает оттенок красного, второй — зеленого, а третий — оттенок синего цвета. Диагональ куба, идущая от черного цвета к белому, — это оттенки серого цвета. Диапазон каждого из трех значений может изменяться в пределах от 0 до 255 (либо от 0 до 1 в вещественном представлении), где 0 означает полное отсутствие оттенка цвета, а 255 — его максимальную насыщенность.

Эта модель является «аддитивной», то есть посредством смешивания базовых цветов в различном процентном соотношении можно создать любой нужный цвет. Смешав, например, синий и зеленый, мы получим голубой цвет.

Для создания цветового значения RGB нужно просто передать в конструктор класса `QColor` три значения. Каналы цвета класса `QColor` могут содержать необязательный уровень прозрачности, значение для которого можно передавать в конструктор четвертым параметром. В первый параметр передается значение красного цвета, во второй — зеленого, в третий — синего, а в четвертый — уровень прозрачности. Например:

```
QColor colorBlue(0, 0, 255, 128);
```

Получить из объекта `QColor` каждый компонент цвета возможно с помощью методов `red()`, `green()`, `blue()` и `alpha()`. Эти же значения можно получить и в вещественном представлении, для чего нужно вызвать: `redF()`, `greenF()`, `blueF()` и `alphaF()`. Можно вообще обой-

тись одним методом `getRgb()`, в который передаются указатели на переменные для значений цветов, например:

```
QColor color(100, 200, 0);
int r, g, b;
color.getRgb(&r, &g, &b);
```

Для записи значений RGB можно, по аналогии, воспользоваться методами, похожими на описанные, но имеющими префикс `set` (вместо префикса `get`, если он есть), после которого идет заглавная буква. Также для этой цели можно прибегнуть к структуре данных `QRgb`, которая состоит из четырех байтов и полностью совместима с 32-битным значением. Эту структуру можно создать с помощью функции `qRgb()` или `qRgba()`, передав в нее параметры красного, зеленого и синего цветов. Но можно присваивать переменным структуры `QRgb` и 32-битное значение цвета. Например, синий цвет устанавливается сразу несколькими способами:

```
QRgb rgbBlue1 = qRgba(0, 0, 255, 255); // С информацией о прозрачности
QRgb rgbBlue2 = qRgb(0, 0, 255);
QRgb rgbBlue3 = 0x000000FF;
```

При помощи функций `qRed()`, `qGreen()`, `qBlue()` и `qAlpha()` можно получить значения цветов и информацию о прозрачности соответственно.

Значения типа `QRgb` можно передавать в конструктор класса `QColor` или в метод `setRgb()`:

```
QRgb rgbBlue = 0x000000FF;
QColor colorBlue1(rgbBlue);
QColor colorBlue2;
colorBlue2.setRgb(rgbBlue);
```

Также можно получать значения структуры `QRgb` от объектов класса `QColor` вызовом метода `rgb()`.

Цвет можно установить, передав значения в символьном формате, например:

```
QColor colorBlue1("#0000FF");
QColor colorBlue2;
colorBlue2.setNameColor("#0000FF");
```

Цветовая модель HSV

Модель HSV (Hue, Saturation, Value — оттенок, насыщенность, значение) не смешивает основные цвета при моделировании нового цвета, как в случае с RGB, а просто изменяет их свойства. Это очень напоминает принцип, используемый художниками для получения новых цветов, — подмешивая к чистым цветам белую, черную или серую краски.

Пространство цветов этой модели задается пирамидой с шестиконечным основанием, так называемым *Hexcone* (рис. 17.7). Координаты в этой модели имеют следующий смысл:

- ◆ оттенок (Hue) — это «цвет» в общеупотребительном смысле этого слова, например: красный, оранжевый, синий и т. д., который задается углом в цветовом круге, изменяющимся от 0 до 360 градусов;
- ◆ насыщенность (Saturation) обозначает наличие белого цвета в оттенке. Значение насыщенности может изменяться в диапазоне от 0 до 255. Значение, равное 255 в целочисленном числовом представлении либо единице в вещественном представлении, соответ-

ствует полностью насыщенному цвету, который не содержит оттенков белого. Частично насыщенный оттенок светлее — например, оттенок красного с насыщенностью, равной 128, либо 0,5 в вещественном представлении, соответствует розовому;

- ◆ значение (Value) или яркость — определяет интенсивность цвета. Цвет с высокой интенсивностью — яркий, а с низкой — темный. Значение этого параметра может изменяться в диапазоне от 0 до 255 в целочисленном числовом представлении (либо от 0 до 1 в вещественном представлении).

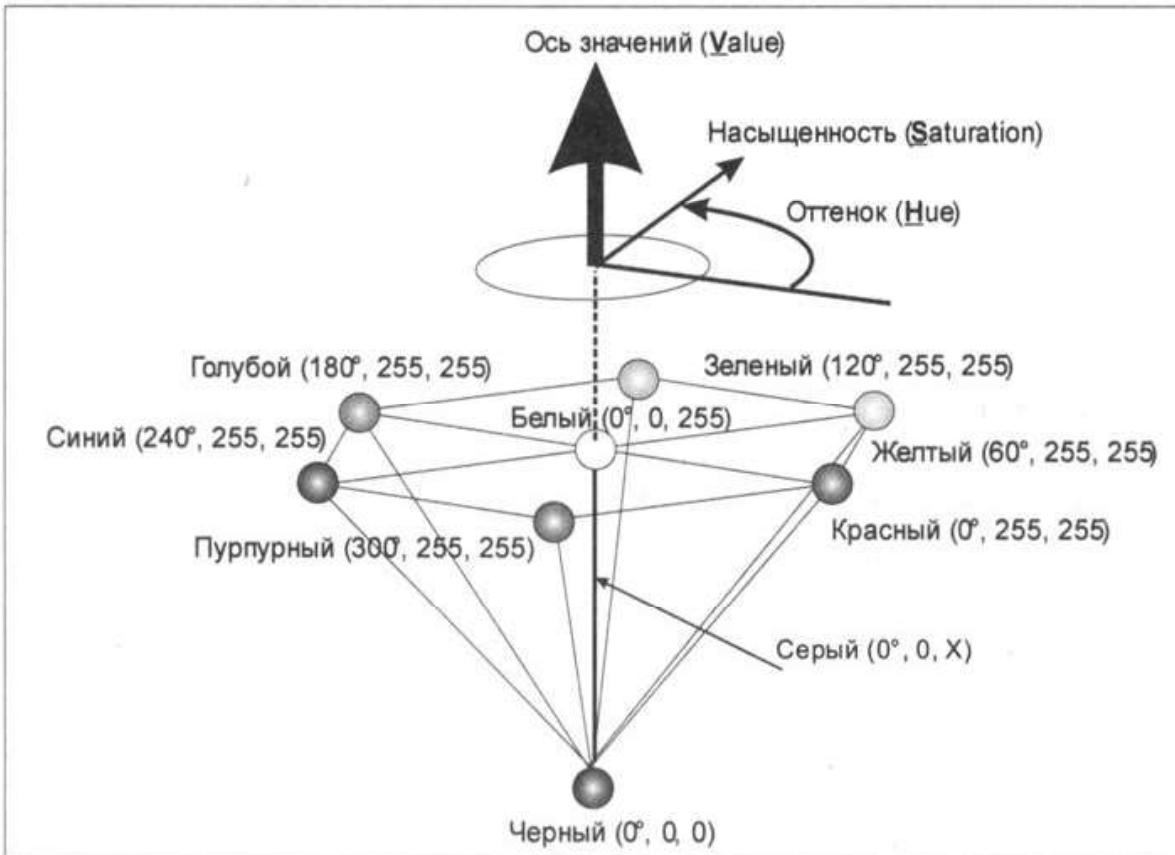


Рис. 17.7. Цветовая модель HSV

Установку значения цвета в координатах HSV можно выполнить с помощью метода `QColor::setHsv()` или `QColor::setHsvF()`:

```
color.setHsv(233, 100, 50);
```

Для того чтобы получить цветовое значение в цветовой модели HSV, нужно передать в метод `getHsv()` адреса трех целочисленных значений (или вещественных, если это `getHsvF()`). Следующий пример устанавливает RGB-значение и получает в трех переменных его HSV-эквивалент:

```
QColor color(100, 200, 0);
int h, s, v;
color.getHsv(&h, &s, &v);
```

Цветовая модель CMYK

Применение модели CMYK (Cyan, Magenta, Yellow, Key color — голубой, пурпурный, желтый, «ключевой» черный цвет) чаще всего связано с печатью. Пространство цветов этой модели так же, как и в случае с RGB, представляет собой куб с длиной сторон равной 255

(рис. 17.8) в целочисленном числовом представлении или единице в вещественном представлении. В отличие от RGB, эта модель является «субтрактивной», то есть вычитаемой. В субтрактивной цветовой модели любой цвет представляется в виде трех величин, каждая из которых указывает, какое количество определенного цвета подлежит исключению из белого.

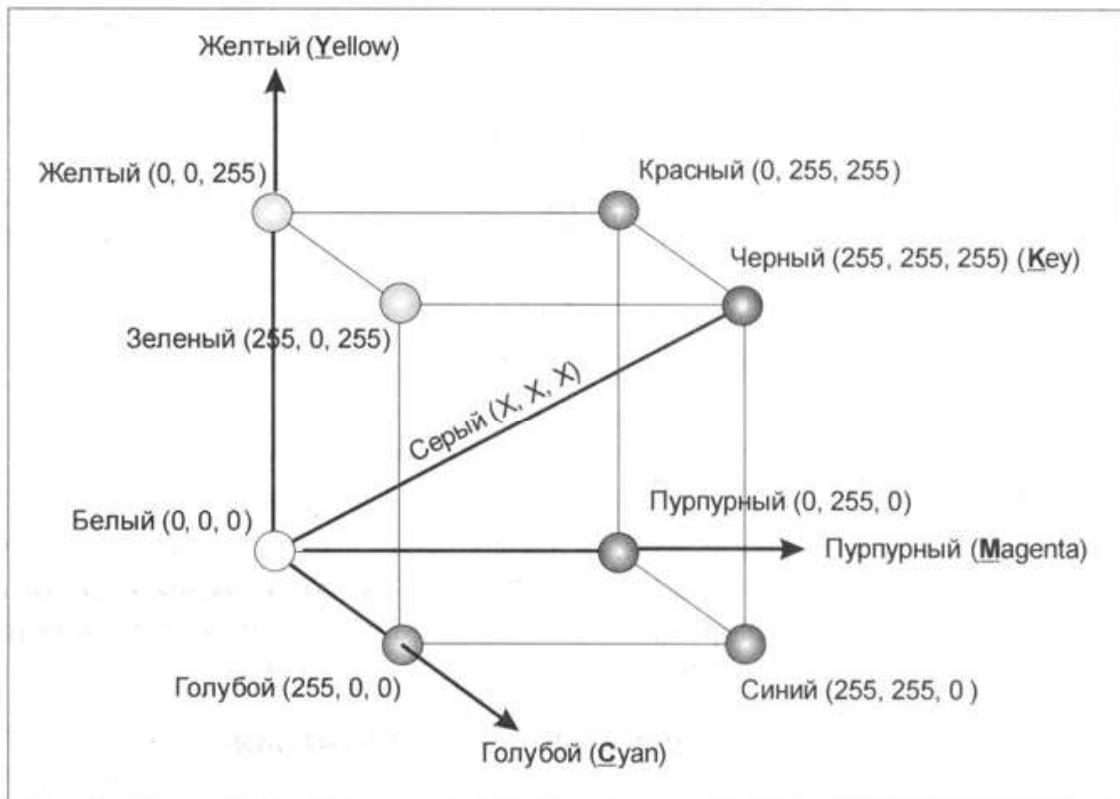


Рис. 17.8. Цветовая модель CMYK

Таким образом, для получения черного цвета на печатающем устройстве необходимо задействовать все три основные составляющие: желтую, голубую и пурпурную. Такой подход не отличается экономичностью и на практике не всегда дает чисто черный цвет, в связи с чем в печатающих устройствах дополнительно используется черная краска. Вот именно поэтому черный цвет и представляет собой четвертую составляющую (Key color) этой модели.

KEY COLOR, A НЕ BLACK!

Конечно, логичнее было бы назвать последнюю компоненту не Key color, а Black. Но Black начинается с буквы «В», и в этом случае могла бы возникнуть путаница с синим цветом (Blue), чья первая буква уже задействована в модели RGB.

Установка цвета для цветовой модели CMYK осуществляется таким же образом, как и в случаях с RGB и HSV. Класс `QColor` предоставляет методы `getCMYK()` и `getCMYKF()` для получения значений, а методы `setCMYK()` и `setCMYKF()` предназначены для их установки.

Палитра

Палитра предоставляет ограниченное (в большинстве случаев числом 256) количество цветовых значений. Цветовые значения адресуются при помощи индексов. Сами индексируемые цветовые значения можно задавать свободно. На рис. 17.9 отображается пикセル, имеющий значение цвета RGB(200, 75, 13), адресуемое индексом 3.

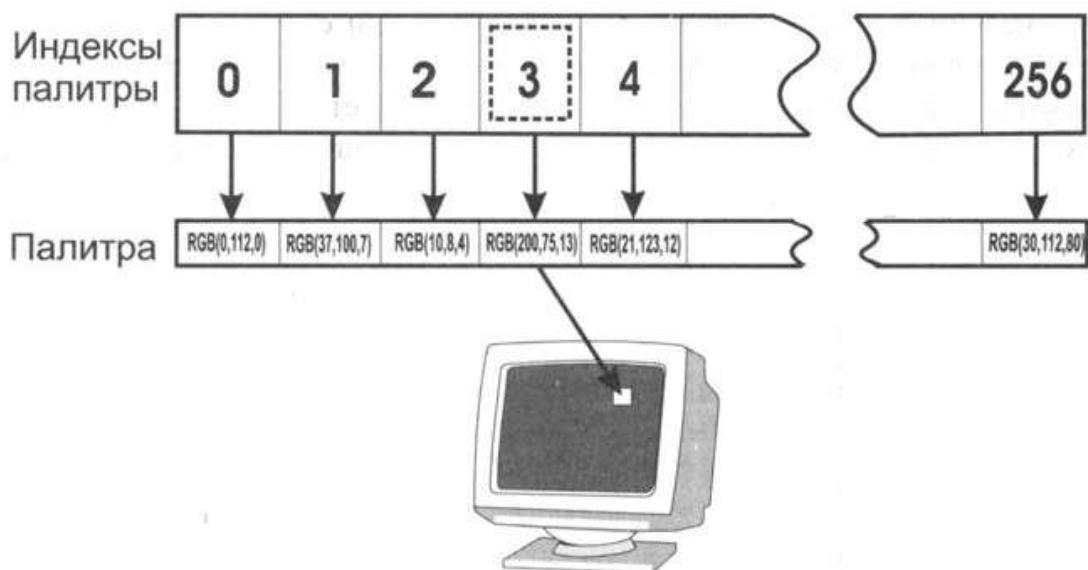


Рис. 17.9. Отображение пикселя, имеющего указанный в палитре цвет

Предопределенные цвета

В табл. 17.1 приведены константы именованных цветов, предопределенных в Qt. Они представляют собой палитру, состоящую из 17 цветов. Конечно, этих цветов недостаточно для получения фотorealистичных изображений, но они удобны на практике, особенно в тех ситуациях, когда требуется отображать основные цветовые значения.

Константы для рисования двухцветных изображений

В общей сложности именованных цветов 19. Две константы, не приведенные в таблице: `Qt::color0` и `Qt::color1` — используются для рисования двухцветных изображений.

Таблица 17.1. Цвета перечисления `GlobalColor` пространства имен `Qt`

Константа	RGB-значение	Описание
<code>black</code>	(0, 0, 0)	Черный
<code>white</code>	(255, 255, 255)	Белый
<code>darkGray</code>	(128, 128, 128)	Темно-серый
<code>gray</code>	(160, 160, 164)	Серый
<code>lightGray</code>	(192, 192, 192)	Светло-серый
<code>red</code>	(255, 0, 0)	Красный
<code>green</code>	(0, 255, 0)	Зеленый
<code>blue</code>	(0, 0, 255)	Синий
<code>cyan</code>	(0, 255, 255)	Голубой
<code>magenta</code>	(255, 0, 255)	Пурпурный
<code>yellow</code>	(255, 255, 0)	Желтый
<code>darkRed</code>	(128, 0, 0)	Темно-красный
<code>darkGreen</code>	(0, 128, 0)	Темно-зеленый
<code>darkBlue</code>	(0, 0, 128)	Темно-синий
<code>darkCyan</code>	(0, 128, 128)	Темно-голубой

Таблица 17.1 (окончание)

Константа	RGB-значение	Описание
darkMagenta	(128, 0, 128)	Темно-пурпурный
darkYellow	(128, 128, 0)	Темно-желтый

Класс QColor предоставляет методы `lighter()` и `darker()`, с помощью которых можно получать значения цвета, делая основное значение светлее или темнее. Методы `lighter()` и `darker()` не изменяют исходный объект цвета, а создают новый. Для этого текущий цвет в модели RGB преобразуется в цвет модели HSV и ее компонент «Значение» (Value) умножается (для `darker()` — делится) на множитель (выраженный в процентах), переданный в этот метод, а затем полученное значение преобразуется обратно в модель RGB. Сделать красный цвет немного темнее можно следующим образом:

```
QColor color = QColor(Qt::red).darker(160);
```

Резюме

Графика играет очень важную роль, ее использование можно встретить практически во всех серьезных программных продуктах.

Qt предоставляет ряд классов геометрии, необходимых при создании программ с графикой. Объекты классов QPoint/QPointF хранят в себе координаты X и Y , описывающие расположение точки на плоскости. Классы размера QSize/QSizeF предназначены для хранения значений ширины и высоты. Классы QRect/QRectF объединяют в себе величины, хранящиеся в объектах классов QPoint/QPointF и QSize/QSizeF. Классы QLine/QLineF и QPolygon/QPolygonF предоставляют возможность описания линий и многоугольников.

Qt поддерживает три цветовые модели: RGB, CMYK и HSV. RGB — это очень распространенная цветовая модель, в которой любой цвет получается в результате смешения трех цветов: красного, зеленого и синего. Цветовая модель CMYK получила большое распространение в полиграфии. В модели HSV цвет задается тремя параметрами: оттенком (Hue), насыщенностью (Saturation) и значением (Value), или, иначе, яркостью.

Представление цвета TrueColor дает возможность получить любой нужный цвет. В него входят все представления, имеющие более 8 битов.

Палитра — это массив, в котором каждому возможному значению пикселя в соответствие ставится значение цвета.

Класс QColor предназначен для хранения цветовых значений и предоставляет множество полезных методов, с помощью которых можно конвертировать цветовые значения из RGB, CMYK в HSV (и наоборот), сравнивать их, делать светлее или темнее.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: <http://qt-book.com/ru/17-510/> или с помощью следующего QR-кода (рис. 17.10):



Рис. 17.10. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 18

Легенда о короле Артуре и контекст рисования

Сэр Артур осмотрел свой меч и остался им доволен.

— Что тебе больше нравится, — сказал Мерлин, — меч или ножны?

— Мне больше нравится меч, — сказал Артур.

— Не мудр твой ответ, — сказал Мерлин, — ибо эти ножны в десять раз драгоценней меча.

Марк Твен,
«Янки из Коннектикута при дворе короля Артура»

Под именем «Артур» (*Arthur*) подразумевается новая архитектура для рисования, создание которой началось с желания использовать методы рисования `QPainter` в `OpenGL`. Затем появилось множество других идей: от точного представления в вещественных координатах до отображения векторной графики, которые получили свое воплощение в этой архитектуре.

Три краеугольных камня этой технологии составляют классы `QPainter`, `QPaintEngine` и `QPaintDevice` (рис. 18.1).

Класс `QPaintEngine` используется классами `QPainter` и `QPaintDevice` неявно и для разработчиков не интересен, если нет необходимости создавать свой собственный контекст рисования. Если же такая необходимость возникла, то вам придется унаследовать этот класс и реализовать некоторые его методы.

Основной класс для программирования графики, с которым нам придется иметь дело, — это `QPainter`. С его помощью можно рисовать точки, линии, эллипсы, многоугольники (полигоны), кривые Безье, растровые изображения, текст и многое другое. Более того, класс `QPainter` поддерживает режим *сглаживания* (*antialiasing*), прозрачность и градиенты, о которых будет рассказано в этой главе.

Контекст рисования `QPaintDevice` можно представить себе как поверхность для вывода графики. `QPaintDevice` — это основной абстрактный класс для всех классов объектов, которые можно рисовать. От него унаследована целая серия классов, показанных на рис. 18.2.

В основном, рисование выполняется из метода обработки события `QPaintEvent` (см. главу 14). Если пользователь запустит программу и выполнит некоторые действия, вследствие которых перекроется, частично или полностью, окно программы, то после его открытия будет сгенерировано событие перерисовки и вызван метод `QWidget::paintEvent()` для тех виджетов, которые должны быть перерисованы. Сам объект события `QPaintEvent` содержит метод `region()`, возвращающий область для перерисовки. Метод `QPaintEvent::rect()` возвращает прямоугольник, который охватывает эту область.

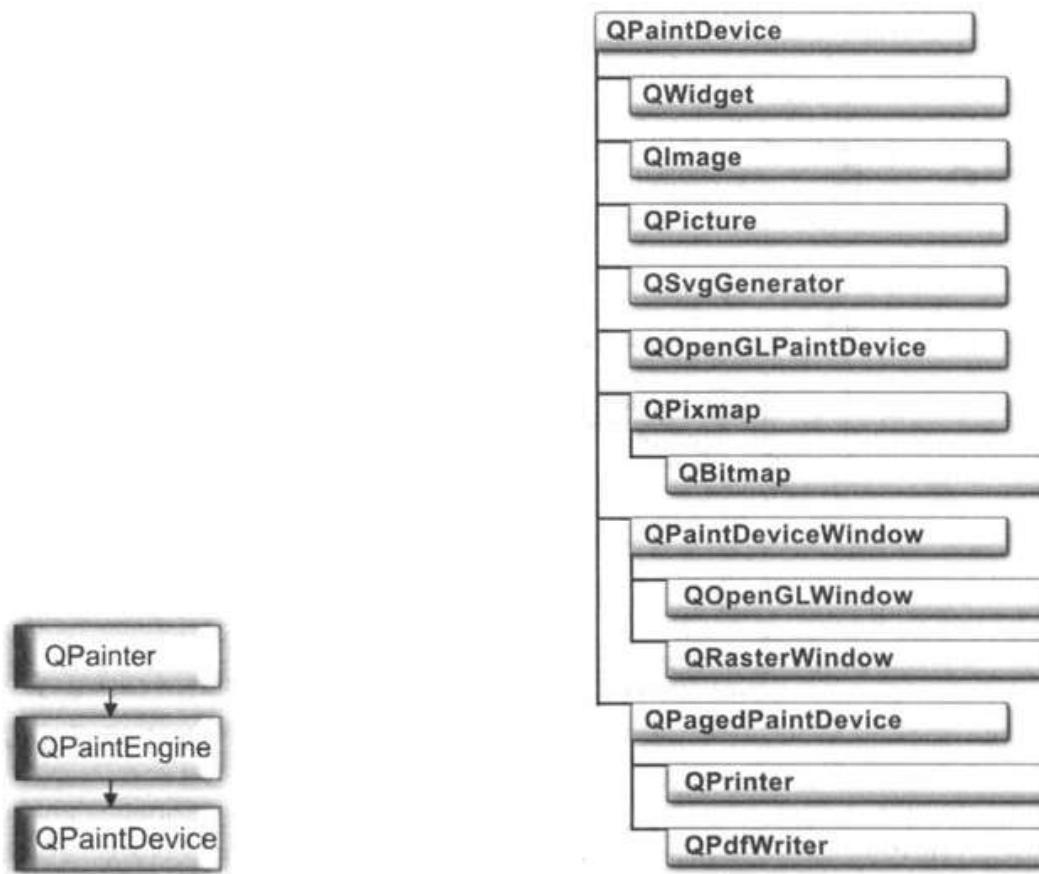


Рис. 18.1. Взаимосвязь классов архитектуры «Артур»

Рис. 18.2. Иерархия классов контекста рисования

Для подавления эффекта мерцания Qt использует технику *двойной буферизации* (double buffering). Двойная буферизация представляет собой очень распространенное и простое решение этой проблемы. Суть заключается в формировании изображения в невидимой области (буфере). Сформированное изображение помещается в видимую область (буфер) за один раз. Это выполняется автоматически, и вам не нужно реализовывать для этого код в `paintEvent()`.

Эффекта прозрачности можно добиться, установив для рисования цвет, содержащий значение прозрачности альфа-канала (см. главу 17). Это значение может изменяться от 0 до 255. Значение 0 говорит том, что цвет полностью прозрачен, а 255 означает полную непрозрачность.

Класс QPainter

Класс **QPainter**, определенный в заголовочном файле `QPainter`, является исполнителем команд рисования. Он содержит массу методов для отображения линий, прямоугольников, окружностей и др. Рисование осуществляется на всех объектах классов, унаследованных от класса **QPaintDevice** (рис. 18.3). Это означает, что то, что отображается контекстом рисования одного объекта, может быть точно так же отображено контекстом и другого объекта.

Чтобы использовать объект **QPainter**, необходимо передать ему адрес объекта контекста, на котором должно осуществляться рисование. Этот адрес можно передать как в конструкторе, так и с помощью метода `QPainter::begin()`. Смысл метода `begin()` состоит в том, что он позволяет рисовать на одном контексте несколькими объектами класса **QPainter**. При использовании метода `begin()` нужно по окончании работы с контекстом вызвать метод `QPainter::end()`, чтобы отсоединить установленную этим методом связь с контекстом рисования, давая возможность для рисования другому объекту (листинг 18.1).

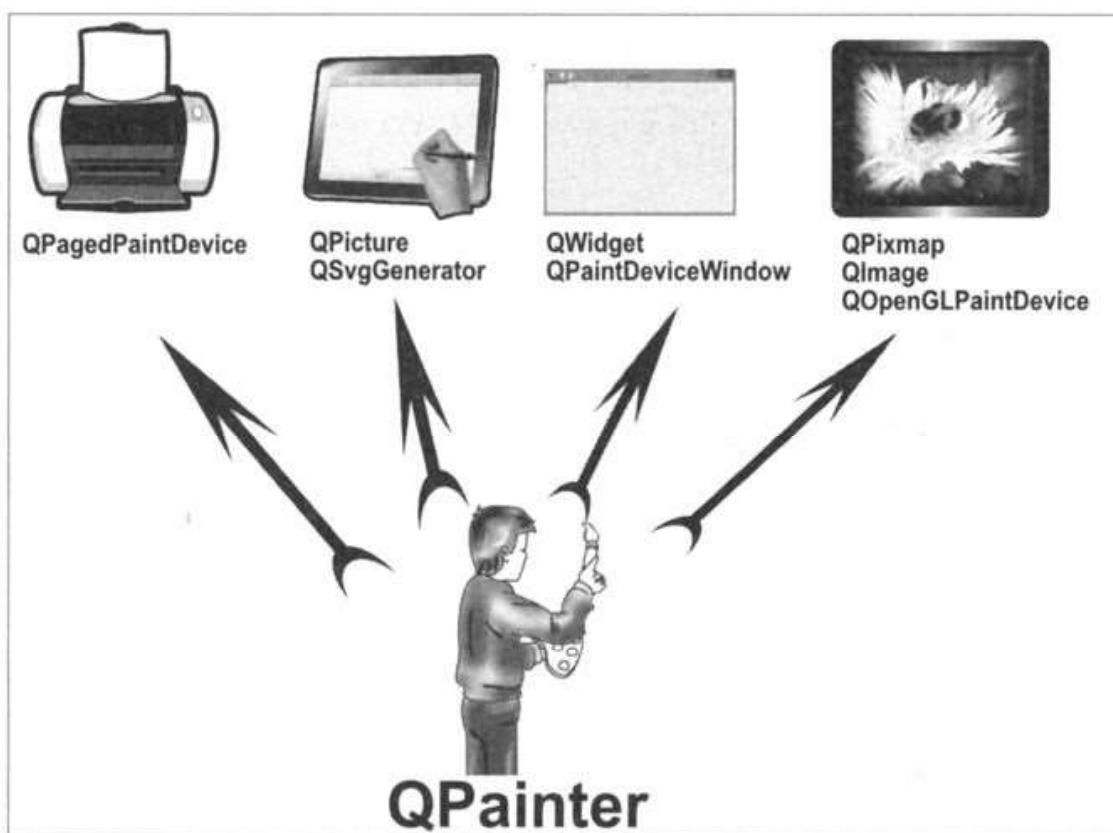


Рис. 18.3. QPainter и контексты рисования

Листинг 18.1. Рисование двумя объектами QPainter в одном контексте

```
...
void MyWidget::paintEvent (QPaintEvent*)
{
    QPainter painter1;
    QPainter painter2;

    painter1.begin(this);
    // Команды рисования
    painter1.end();

    painter2.begin(this);
    // Команды рисования
    painter2.end();
}
```

Но чаще всего используется рисование одним объектом QPainter в разных контекстах (листинг 18.2).

Листинг 18.2. Рисование одним объектом QPainter в двух разных контекствах

```
...
void MyWidget::paintEvent (QPaintEvent*)
{
    QPainter painter;
```

```

painter.begin(this); //контекст виджета
// Команды рисования
painter.end();

QPixmap pix(rect());
painter.begin(&pix); //контекст растрового изображения
// Команды рисования
painter.end();
}

...

```

Объекты QPainter содержат установки, влияющие на их рисование. Это могут быть трансформация координат, модификация кисти и пера, установка шрифта, установка режима сглаживания и т. д. Поэтому, для того чтобы оставить старые настройки без изменений, рекомендуется перед началом изменения сохранить их с помощью метода QPainter::save(), а по окончании работы — восстановить с помощью метода QPainter::restore().

Перья и кисти

Перья и кисти — это основа для программирования графики с использованием библиотеки Qt. Без них не получится вывести на экран даже точку.

Перо

Перо служит для рисования контурных линий фигуры. Атрибуты пера: цвет, толщина и стиль. Установить новое перо можно с помощью метода QPainter::setPen(), передав в него объект класса QPen. Можно передавать и предопределенные стили пера, указанные в табл. 18.1.

Таблица 18.1. Некоторые значения из перечисления PenStyle пространства имён Qt

Константа	Значение	Вид (толщина = 4)
NoPen	0	
SolidLine	1	
DashLine	2	
DotLine	3	
DashDotLine	4	
DashDotDotLine	5	

Толщина линии является значением целого типа, которое передается в метод QPen::setWidth(). Если значение равно нулю, то это не означает, что линия будет невидима, а говорит лишь о том, что она должна быть изображена как можно тоньше.

Если необходимо, чтобы линия не отображалась вообще, то тогда устанавливается стиль NoPen. Зачем же нужно перо, которое не рисует? Бывают и такие случаи, когда и пустое перо пригодится, — например, когда нужно вывести четырехугольник определенного цвета без контурной линии.

Цвет пера задается с помощью метода `QPen::setColor()`, в который передается объект класса `QColor`. Следующий пример создает перо красного цвета, толщиной в три пикселя и со стилем «штрих». Объект пера устанавливается в объекте `QPainter` вызовом метода `setPen()`:

```
QPainter painter(this);
painter.setPen(QPen(Qt::red, 3, Qt::DashLine));
```

Стили для концов линий пера устанавливаются методом `setCapStyle()`, в который передается один из флагов: `Qt::FlatCap` (край линии квадратный и проходит через граничную точку), `Qt::SquareCap` (край квадратный и перекрывает граничную точку на половину ширины линии) или `Qt::RoundCap` (край закругленный и также покрывает граничную точку линии).

Можно устанавливать стили и для переходов одной линии в другую — методом `setJoinStyle()`, передав в него флаги: `Qt::MiterJoin` (линии продлеваются и соединяются под острым углом), `Qt::BevelJoin` (пространство между линиями заполняется) или `Qt::RoundJoin` (угол закругляется). Но эти переходы будут видны только на толстых линиях.

Кисть

Кисть служит для заполнения непрерывных контуров, таких как прямоугольники, эллипсы и многоугольники. Класс кисти `QBrush` определен в заголовочном файле `QBrush.h`. Кисть задается двумя параметрами: цветом и образцом заливки.

Установить кисть можно методом `QPainter::setBrush()`, передав в него объект класса `QBrush` или один из предопределенных шаблонов, указанных в табл. 18.2. Если заполнение не нужно, то в метод `QPainter::setBrush()` следует передать значение `NoBrush`.

Таблица 18.2. Перечисление `BrushStyle` пространства имён `Qt` (выборочно)

Константа	Значение	Вид	Константа	Значение	Вид
NoBrush	0		VerPattern	10	
SolidPattern	1		CrossPattern	11	
Dense1Pattern	2		BDiagPattern	12	
Dense2Pattern	3		FDiagPattern	13	
Dense3Pattern	4		DiagCrossPattern	14	
Dense4Pattern	5		LinearGradientPattern	15	
Dense5Pattern	6		RadialGradientPattern	16	
Dense6Pattern	7		ConicalGradientPattern	17	
Dense7Pattern	8		TexturePattern	24	
HorPattern	9				

Следующие строки устанавливают красную кисть с горизонтальной штриховкой:

```
QPainter painter(this);
painter.setBrush(QBrush(Qt::red, Qt::HorPattern));
```

Если в табл. 18.2 не нашлось подходящей кисти, то можно создать свою собственную с помощью стиля TexturePattern. Чтобы применить этот стиль, нужно передать в метод setTexture() растровое изображение. Растровое изображение можно также использовать и при создании кисти (рис. 18.4):

```
...
void MyWidget::paintEvent (QPaintEvent*)
{
    QPainter painter(this);
    QPixmap pix(":/Alina.jpg");
    painter.setBrush(QBrush(Qt::black, pix));
    painter.drawEllipse(0, 0, 300, 150);
}
```



Рис. 18.4. Заполнение эллипса растровым изображением

Градиенты

Градиент — это плавный переход от одного цвета к другому. В настоящее время применение градиентов стало очень популярно, ведь с их помощью можно придать элементам изображений в ваших приложениях эффект объемности. В основе градиентов лежит гладкая интерполяция между двумя и более цветовыми значениями. Qt предоставляет три основных типа градиентов: линейный (linear), конический (conical) и радиальный (radial).

Линейные (linear) градиенты реализует класс QLinearGradient. Они задаются двумя цветовыми точками контроля и несколькими точками останова (color stops) на линии, соединяющей цвета этих точек. Листинг 18.3 иллюстрирует эту возможность.

Листинг 18.3. Линейный градиент

```
...
void MyWidget::paintEvent (QPaintEvent*)
{
    QPainter painter(this);
    QLinearGradient gradient(0, 0, width(), height());
```

```

gradient.setColorAt(0, Qt::red);
gradient.setColorAt(0.5, Qt::green);
gradient.setColorAt(1, Qt::blue);
painter.setBrush(gradient);
painter.drawRect(rect());
}
...

```

В листинге 18.3 мы задали три цвета на трех различных позициях между двумя точками контроля. Позиции точек задаются вещественными значениями от 0 до 1, где 0 представляет собой первую контрольную точку, а 1 — вторую. Цвета между этими точками будут интерполированы (рис. 18.5).

Конический (conical) градиент реализуется классом `QConicalGradient` и задается центральной точкой и углом. Распространение цветов вокруг центральной точки соответствует повороту часовой стрелки. В листинге 18.4 приведена реализация конического градиента (рис. 18.6).

Листинг 18.4. Конический градиент

```

...
void MyWidget::paintEvent(QPaintEvent*)
{
    QPainter painter(this);
    QConicalGradient gradient(width() / 2, height() / 2, 0);
    gradient.setColorAt(0, Qt::red);
    gradient.setColorAt(0.4, Qt::green);
    gradient.setColorAt(0.8, Qt::blue);
    gradient.setColorAt(1, Qt::red);
    painter.setBrush(gradient);
    painter.drawRect(rect());
}
...

```

Радиальный (radial) градиент реализует класс `QRadialGradient` и задается центральной точкой, радиусом и точкой фокуса. Центральная точка и радиус задают окружность. На рас-

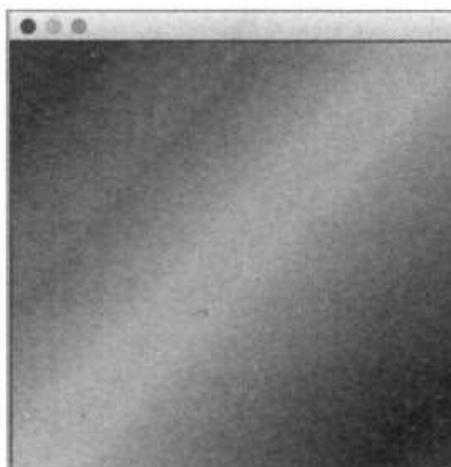


Рис. 18.5. Отображение линейного градиента

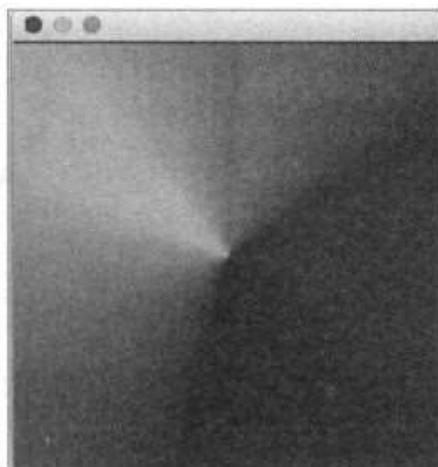


Рис. 18.6. Отображение конического градиента

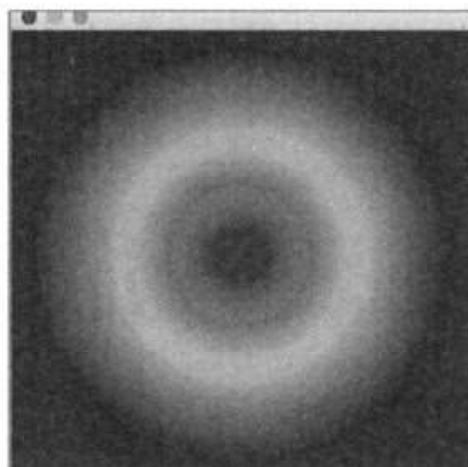


Рис. 18.7. Отображение радиального градиента

пространение цветов за пределами точки фокуса влияет центральная точка или точка, находящаяся внутри окружности. Пример реализации такого градиента приведен в листинге 18.5, а результат показан на рис. 18.7.

Листинг 18.5. Радиальный (лучевой) градиент

```
...
void MyWidget::paintEvent(QPaintEvent*)
{
    QPainter      painter(this);
    QPointF      ptCenter(rect().center());
    QRadialGradient gradient(ptCenter, width() / 2, ptCenter);
    gradient.setColorAt(0, Qt::red);
    gradient.setColorAt(0.5, Qt::green);
    gradient.setColorAt(1, Qt::blue);
    painter.setBrush(gradient);
    painter.drawRect(rect());
}
...
```

Техника сглаживания (Anti-aliasing)

Одним из побочных эффектов, возникающих при рисовании геометрических фигур, является ступенчатость, хорошо заметная на контурах (рис. 18.8). Это и понятно, поскольку такие ступени есть не что иное, как пиксели, через которые проходит контур. Для подавления этого нежелательного эффекта используется техника *сглаживания* (Anti-aliasing). С ее помощью границы кривых можно сделать более гладкими, убирая ступени, образующиеся на краях объектов, что достигается добавлением промежуточных цветов. Это снижает скорость рисования, но улучшает визуальный эффект.

Режим сглаживания распространяется на отображение текста и геометрических фигур. Его можно включить в объекте класса `QPainter` при помощи метода `setRenderHint()`:

```
painter.setRenderHint(QPainter::Antialiasing, true);
```

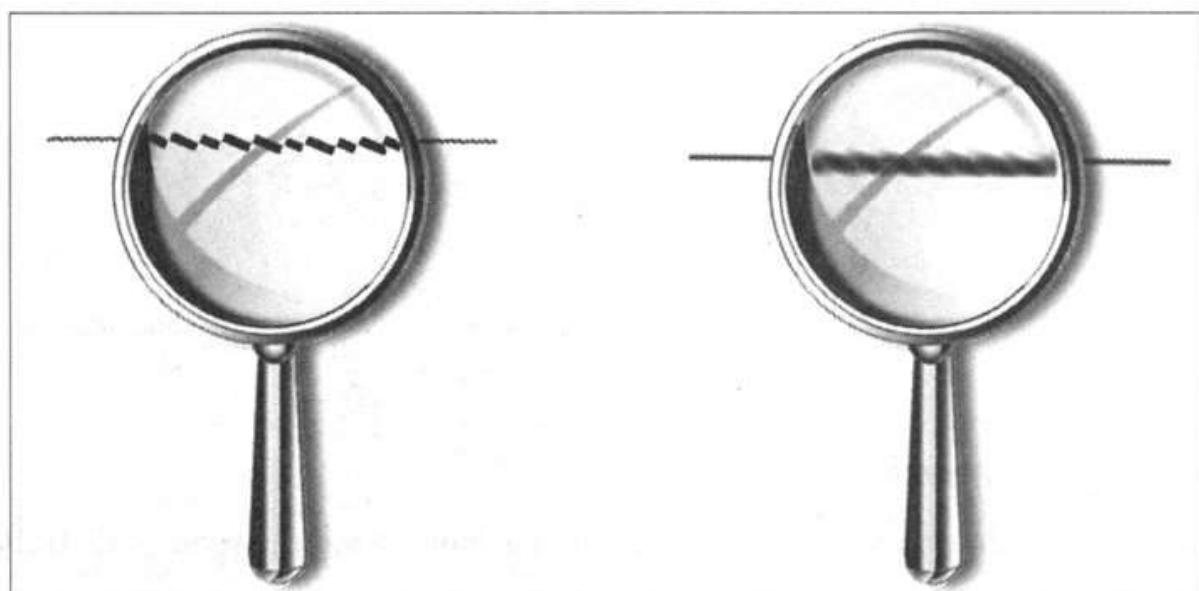


Рис. 18.8. Линия без сглаживания (слева) и со сглаживанием (справа)

Рисование

Отображение фигур — задача несложная, ведь для этого не требуется вычислять расположение каждого выводимого пикселя, так как уже имеется целый ряд методов для вывода практически всех геометрических фигур. Например, вызовом метода `drawRect()` можно нарисовать прямоугольник.

Перед тем как приступить к рисованию, важно понять философию координат пикселя. Она заключается в том, что центр пикселя лежит в его середине. То есть пикセル, находящийся в самом верхнем левом углу, будет иметь координаты (0,5; 0,5). Если мы попытаемся нарисовать пикセル с координатами (0; 0), то `QPainter` автоматически прибавит к ним 0,5, и результатом все равно станет (0,5; 0,5). Этот сдвиг выполняется при выключенном режиме сглаживания. Если этот режим включен, и мы попытаемся нарисовать черный пиксель с координатами (50; 50), то в результате сглаживания будут нарисованы сразу четыре серых пикселя с координатами (49,5; 49,5), (49,5; 50,5), (50,5; 49,5) и (50,5; 50,5). Но если мы попытаемся нарисовать черный пиксель с координатами (49,5; 49,5), то в результате увидим только один пиксель.

Рисование точек

Для отображения точек применяется только перо. В листинге 18.6 на экране рисуются восемь точек (рис. 18.9).

Листинг 18.6. Рисование точек методом `drawPoint()`

```
...
void MyWidget::paintEvent (QPaintEvent*)
{
    QPainter painter(this);
    painter.setPen(QPen(Qt::black, 3));

    int n = 8;
    for (int i = 0; i < n; ++i) {
        qreal fAngle = ::qDegreesToRadians(360.0 * i / n);
        qreal x      = 50 + cos(fAngle) * 40;
        qreal y      = 50 + sin(fAngle) * 40;
        painter.drawPoint(QPointF(x, y));
    }
}
...
```

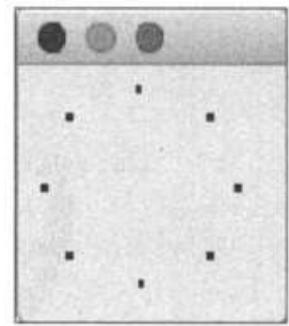


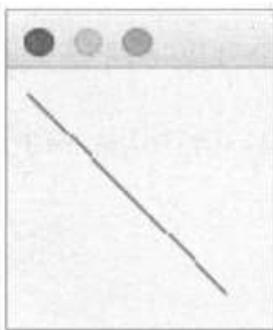
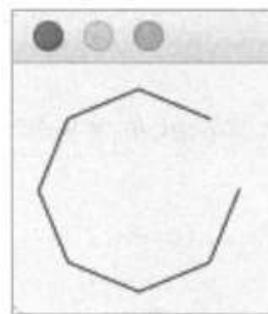
Рис. 18.9. Рисование точек

Рисование линий

Для рисования отрезка прямой линии, идущего из одной точки в другую (рис. 18.10), используется метод `drawLine()`, в который передаются координаты начальной (x_1, y_1) и конечной (x_2, y_2) точек `QPointF` (листинг 18.7).

Листинг 18.7. Рисование линий методом drawLine()

```
...
void MyWidget::paintEvent (QPaintEvent*)
{
    QPainter painter(this);
    painter.setRenderHint (QPainter::Antialiasing, true);
    painter.drawLine (QPointF(10, 10), QPointF(90, 90));
}
...
```

**Рис. 18.10.** Линия**Рис. 18.11.** Соединение точек линиями

Метод `drawPolyline()` проводит линию, которая соединяет точки, передаваемые в первом параметре. Второй параметр задает количество точек, которые должны быть соединены (т. е. число элементов массива). Первая и последняя точки не соединяются. В листинге 18.8 рисуется фигура, показанная на рис. 18.11.

Листинг 18.8. Рисование фигуры методом drawPolyline()

```
...
void MyWidget::paintEvent (QPaintEvent*)
{
    QPainter painter(this);
    painter.setRenderHint (QPainter::Antialiasing, true);

    const int n = 8;
    QPointF a[n];
    for (int i = 0; i < n; ++i) {
        qreal fAngle = ::qDegreesToRadians(360.0 * i / n);
        qreal x      = 50 + cos(fAngle) * 40;
        qreal y      = 50 + sin(fAngle) * 40;
        a[i] = QPointF(x, y);
    }
    painter.drawPolyline(a, n);
}
...
```

Рисование сплошных прямоугольников

Прямоугольник — это очень распространенная геометрическая фигура, посмотрите вокруг — прямоугольные предметы окружают нас везде. Qt содержит два метода для рисования прямоугольников без контурных линий: `fillRect()` и `eraseRect()`. Их внешний вид задается только кистью. В метод `fillRect()` передаются пять параметров. Первые четыре параметра задают координаты (x, y) и размеры (ширина, высота) прямоугольника. Пятый параметр задает кисть.

В метод `eraseRect()` передаются только четыре параметра, задающие позицию и размеры прямоугольной области. Для ее заполнения используется фон, установленный в виджете. Таким образом, вызов этого метода эквивалентен вызову `fillRect()` с пятым параметром — значением, возвращаемым методом `paletteBackgroundColor()`. В листинге 18.9 приведен пример использования методов `fillRect()` и `eraseRect()`, а результат показан на рис. 18.12.

Листинг 18.9. Рисование сплошных прямоугольников методами `fillRect()` и `eraseRect()`

```
...
void MyWidget::paintEvent (QPaintEvent *)
{
    QPainter painter(this);
    QBrush brush(Qt::red, Qt::Dense4Pattern);
    painter.fillRect(10, 10, 100, 100, brush);
    painter.eraseRect(20, 20, 80, 80);
}
...

```

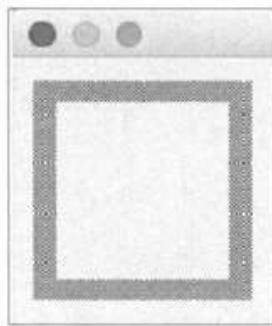


Рис. 18.12. Сплошные прямоугольники

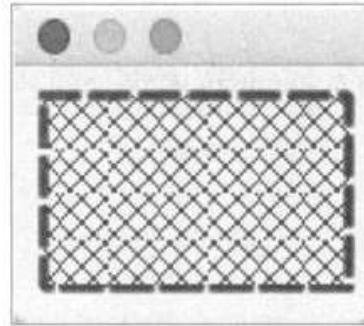


Рис. 18.13. Заполненный прямоугольник

Рисование заполненных фигур

Для рисования фигур также применяются методы, использующие перо `QPen` и кисть `QBrush`. Если требуется нарисовать только контур фигуры, без заполнения, то для этого методом `QPainter::setBrush()` нужно установить значение стиля кисти `QBrush::NoBrush`. А для рисования фигуры без контурной линии можно методом `QPainter::setPen()` установить стиль пера `QPen::NoPen`.

Метод `drawRect()` рисует прямоугольник (рис. 18.13). В него передаются следующие параметры: координаты (x, y) верхнего левого угла, ширина и высота. В этот метод можно передать также объект класса `QRect` (листинг 18.10).

Листинг 18.10. Рисование заполненного прямоугольника методом drawRect()

```
...
void MyWidget::paintEvent (QPaintEvent*)
{
    QPainter painter(this);
    painter.setRenderHint (QPainter::Antialiasing, true);
    painter.setBrush (QBrush (Qt::red, Qt::DiagCrossPattern));
    painter.setPen (QPen (Qt::blue, 3, Qt::DashLine));
    painter.drawRect (QRect (10, 10, 110, 70));
}
...
```

Метод `drawRoundRect()` рисует прямоугольник с закругленными углами (рис. 18.14). Закругленность углов достигается с помощью четвертинок эллипса. Последние два параметра метода задают, насколько сильно должны быть закруглены углы в направлениях осей координат *X* и *Y* соответственно. При передаче нулевых значений углы не будут закруглены, а при задании им значения 100 прямоугольник превратится в эллипс. Прямоугольную область можно задавать объектом класса `QRect` (листинг 18.11).

Листинг 18.11. Рисование прямоугольника с закругленными углами методом drawRoundRect()

```
...
void MyWidget::paintEvent (QPaintEvent*)
{
    QPainter painter(this);
    painter.setRenderHint (QPainter::Antialiasing, true);
    painter.setBrush (QBrush (Qt::green));
    painter.setPen (QPen (Qt::black));
    painter.drawRoundRect (QRect (10, 10, 110, 70), 30, 30);
}
...
```

Метод `drawEllipse()` рисует заполненный эллипс (рис. 18.15), размеры и расположение которого задаются прямоугольной областью (листинг 18.12).

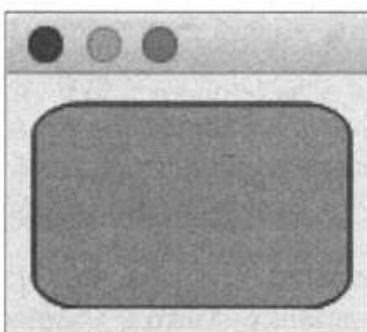


Рис. 18.14. Прямоугольник с закругленными углами

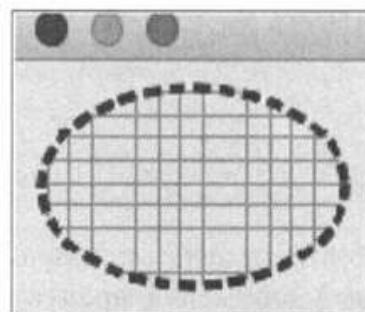


Рис. 18.15. Заполненный эллипс

Листинг 18.12. Рисование заполненного эллипса методом drawEllipse()

```
...
void MyWidget::paintEvent (QPaintEvent*)
{
    QPainter painter(this);
    painter.setRenderHint (QPainter::Antialiasing, true);
    painter.setBrush (QBrush(Qt::green, Qt::CrossPattern));
    painter.setPen (QPen(Qt::red, 3, Qt::DotLine));
    painter.drawEllipse (QRect(10, 10, 110, 70));
}
...
```

Метод `drawChord()` рисует хорду, отсекающую часть эллипса (рис. 18.16). Размеры и расположение эллипса задаются прямоугольной областью, а отображаемая часть — двумя последними параметрами, представляющими собой значения углов. Углы задаются в единицах, равных одной шестнадцатой градуса. Начальная и конечная точки соединяются прямой линией. При положительных значениях двух последних параметров (углов) начальная точка перемещается вдоль кривой эллипса против часовой стрелки. Предпоследний параметр задает начальный угол. Последний параметр задает угол, под которым кривые должны пересекаться (листинг 18.13).

Листинг 18.13. Рисование хорды, отсекающей часть эллипса, методом drawChord()

```
...
void MyWidget::paintEvent (QPaintEvent*)
{
    QPainter painter(this);
    painter.setRenderHint (QPainter::Antialiasing, true);
    painter.setBrush (QBrush(Qt::yellow));
    painter.setPen (QPen(Qt::blue));
    painter.drawChord (QRect(10, 10, 110, 70), 45 * 16, 180 * 16);
}
...
```

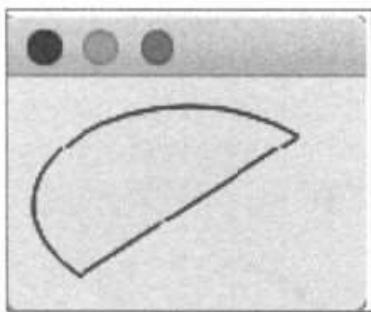


Рис. 18.16. Хорда, отсекающая часть эллипса

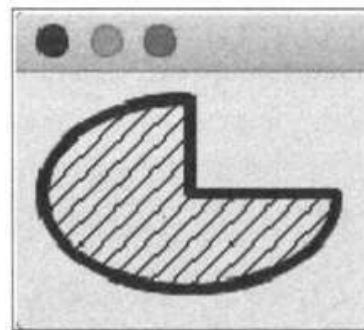


Рис. 18.17. Круговая диаграмма

В мире деловой графики пользуются спросом круговые диаграммы (рис. 18.17). Такие рисунки очень удобны для представления статистических данных. Метод `drawPie()` рисует часть эллипса. Начальная и конечная точки соединяются с центром эллипса. Последние два параметра метода задаются в шестнадцатых долях градуса (листинг 18.14).

Листинг 18.14. Рисование круговой диаграммы методом drawPie()

```
...
void MyWidget::paintEvent (QPaintEvent*)
{
    QPainter painter(this);
    painter.setRenderHint (QPainter::Antialiasing, true);
    painter.setBrush(QBrush(Qt::black, Qt::BDiagPattern));
    painter.setPen(QPen(Qt::blue, 4));
    painter.drawPie(QRect(10, 10, 110, 70), 90 * 16, 270 * 16);
}
...
```

Метод `drawPolygon()` рисует заполненный многоугольник (рис. 18.18), последняя из заданных вершин которого соединена с первой (листинг 18.15).

Листинг 18.15. Рисование заполненного многоугольника методом drawPolygon()

```
...
void MyWidget::paintEvent (QPaintEvent*)
{
    QPainter painter(this);
    painter.setRenderHint (QPainter::Antialiasing, true);
    painter.setBrush(QBrush(Qt::lightGray));
    painter.setPen(QPen(Qt::black));

    int      n = 8;
    QPolygonF polygon;
    for (int i = 0; i < n; ++i) {
        qreal fAngle = ::qDegreesToRadians(360.0 * i / n);
        qreal x      = 50 + cos(fAngle) * 40;
        qreal y      = 50 + sin(fAngle) * 40;
        polygon << QPointF(x, y);
    }
    painter.drawPolygon(polygon);
}
...
```

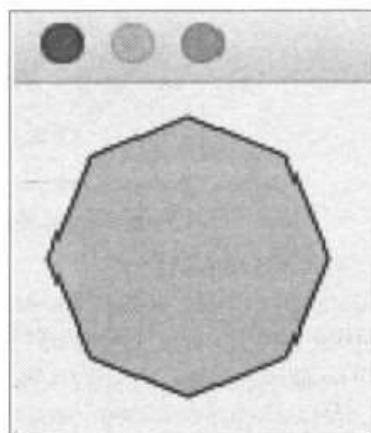


Рис. 18.18. Заполненный многоугольник

Запись команд рисования

Класс QPicture — это контекст рисования, который предоставляет возможность протоколирования команд класса QPainter. С его помощью команды можно даже записывать в отдельные файлы (называемые *метафайлами*), а потом загружать их снова, чтобы повторить ранее проделанные действия. Эти действия можно перенаправлять и на другие контексты рисования, например на принтер или экран. В листинге 18.16 выполняется запись одной команды рисования в файл myline.dat.

Листинг 18.16. Протоколирование команд рисования

```
QPicture pic;
QPainter painter;

painter.begin(&pic);
painter.drawLine(20, 20, 50, 50);
painter.end();

if (!pic.save("myline.dat")) {
    qDebug() << "can not save the file";
}
```

Листинг 18.17 демонстрирует загрузку команд из файла и их выполнение в другом контексте. Для отображения в другом контексте используется метод drawPicture(). Первый параметр этого метода устанавливает позицию, с которой начнется рисование, а во втором параметре передается объект класса QPicture.

Листинг 18.17. Загрузка и выполнение команды в другом контексте рисования

```
...
void MyWidget::paintEvent (QPaintEvent*)
{
    QPicture pic;
    if (!pic.load("myline.dat")) {
        qDebug() << "can not load the file";
    }

    QPainter painter;
    painter.begin(this);
    painter.drawPicture(QPoint(0, 0), pic);
    painter.end();
}
...
```

Трансформация систем координат

Класс QPainter предоставляет очень мощный механизм трансформации координат для отображения объектов. Это позволяет показывать изображения в повернутом, масштабированном, смещенном и склоненном виде (рис. 18.19).

Каждая точка в двумерной системе координат описывается, соответственно, двумя координатами. Трансформация одинаково действует на все точки графического объекта. Для трансформации в классе QPainter определены следующие методы: `translate()`, `scale()`, `rotate()` и `shear()`. Трансформации можно комбинировать, но порядок их следования отражается на конечном результате. Например, если сначала провести операцию скоса, а затем операции поворота и снова скоса, то результат будет отличаться от итога, полученного после двух операций скоса и поворота.

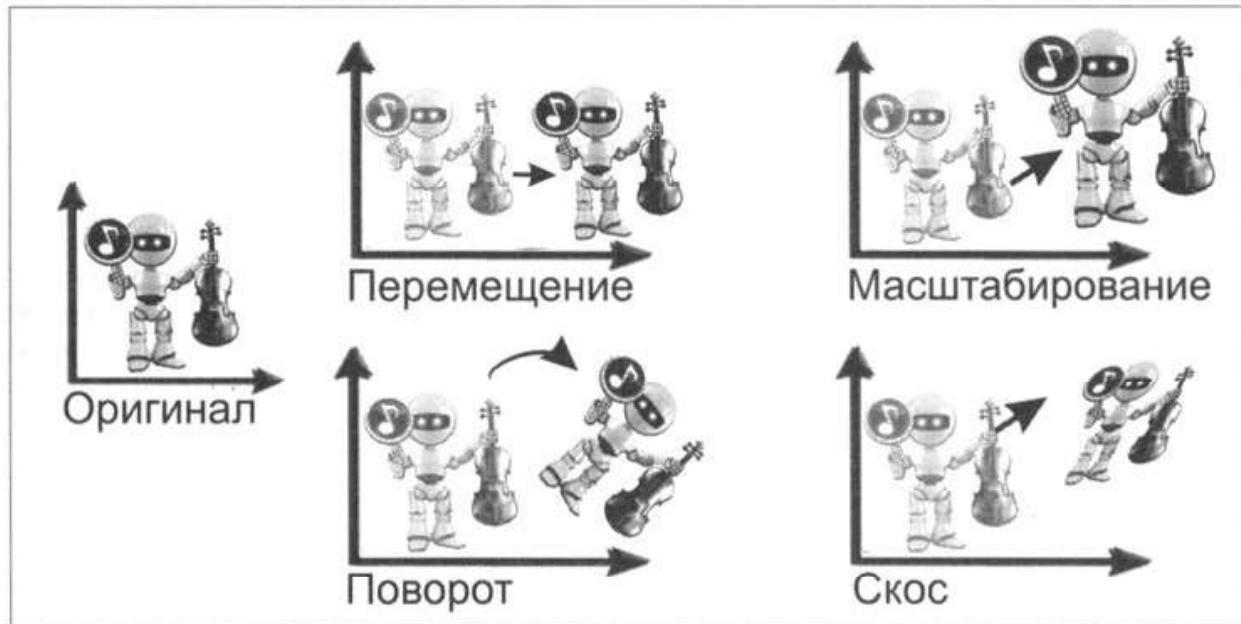


Рис. 18.19. Различные виды трансформации объектов изображений

Для сохранения и восстановления состояний объектов класса QPainter Qt предоставляет методы `save()` и `restore()`. Это очень удобно при получении извне указателя на объект класса QPainter. Можно сохранить его состояние с помощью метода `save()`, а затем проводить с ним разного рода трансформации. По окончании трансформации вызов метода `restore()` вернет объект класса QPainter в исходное состояние. Например:

```
pPainter->save();
pPainter->translate(20, 40);
pPainter->restore();
```

Перемещение

Часто требуется переместить изображение на экране. Класс QPainter предоставляет для этого метод `translate()`, в который передаются два целочисленных параметра. В первом параметре передается значение перемещения по оси X, во втором — по оси Y. Положительные значения первого параметра перемещают объект вправо, а отрицательные — влево. Положительные значения второго параметра смещают объект вниз, а отрицательные — вверх. Например, следующий вызов осуществляет перемещение всех рисуемых объектов вправо на 20 и вниз на 10 пикселов:

```
QPainter painter;
...
painter.translate(20, 10);
```

Масштабирование

Метод `scale()` изменяет размер изображения в соответствии с передаваемыми в него двумя множителями: для ширины и высоты. Значения меньше единицы выполняют уменьшение, больше единицы — увеличение объекта. Например, после следующего вызова ширина всех рисуемых объектов увеличится в полтора раза, а их высота уменьшится наполовину:

```
QPainter painter;
...
painter.scale(1.5, 0.5);
```

Поворот

Одной из основных операций в графике является поворот изображения на определенный угол. Для этого класс `QPainter` содержит метод `rotate()`, в который передается значение типа `double`, обозначающее угол в градусах. При положительных значениях поворот осуществляется по часовой стрелке, а при отрицательных — против нее. Следующий вызов приведет к изображению рисуемых объектов, повернутых (по часовой стрелке) на 30 градусов:

```
QPainter painter;
...
painter.rotate(30.0);
```

Скос

Этот вид трансформации также важен в компьютерной графике. Он реализуется в классе `QPainter` методом `shear()`. Первый параметр задает сдвиг по вертикали, а второй — по горизонтали. Следующий пример осуществляет скос вниз по вертикали:

```
QPainter painter;
...
painter.shear(0.3, 0.0);
```

Трансформационные матрицы

В каждом объекте класса `QPainter` хранится трансформационная матрица. Ее можно считать из объекта, а можно установить созданную матрицу трансформации с помощью метода `QPainter::setTransform()`.

Если для того чтобы получить нужный результат, вам необходимо вызывать несколько методов трансформации, то эффективнее записать их в объект трансформационной матрицы и устанавливать ее в объекте `QPainter` всякий раз, когда необходима трансформация. Например:

```
QTransform mat;
mat.scale(0.7, 0.5);
mat.shear(0.2, 0.5);
mat.rotate(15);
painter.setTransform(mat);
painter.drawText(rect(), Qt::AlignCenter, "Transformed Text");
```

Любая двумерная трансформация может быть описана матрицей размерностью 3×3 :

```
M11 M12 0
M21 M22 0
Dx  Dy  0
```

Если стандартных трансформаций недостаточно, то можно определить свою собственную при помощи значений M11, M12, M21, M22, Dx и Dy, которые задаются в конструкторе класса `QTransform` и представляют собой действительные числа. В табл. 18.3 сведены вместе основные формы трансформации в матричном представлении. Например, установка следующей матрицы в объекте `QPainter` будет соответствовать вызову его метода `translate(20, 10)`:

```
QTransform mat(1, 0, 0, 1, 20, 10);
painter.setTransform(mat);
```

Таблица 18.3. Трансформации

Элемент матрицы	Перемещение	Поворот	Скос	Масштабирование
M11	1	cos(угол)	1	Горизонтальный компонент
M12	0	sin(угол)	Горизонтальный компонент	0
M21	0	-sin(угол)	Вертикальный компонент	0
M22	1	cos(угол)	1	Вертикальный компонент
Dx	Горизонтальный компонент	0	0	0
Dy	Вертикальный компонент	0	0	0

Графическая траектория (painter path)

Графическая траектория может состоять из различных геометрических объектов: прямоугольников, эллипсов и других фигур различной сложности. Ее основное преимущество заключается в том, что, единожды создав траекторию, ее можно отображать сколько угодно раз, одним лишь вызовом метода `QPainter::drawPath()`. Листинг 18.18 реализует траекторию, созданную из трех геометрических объектов: прямоугольника, эллипса и кривой Безье (рис. 18.20).

Листинг 18.18. Создание графической траектории

```
...
void MyWidget::paintEvent (QPaintEvent*)
{
    QPainterPath path;
    QPointF      pt1(width(), height() / 2);
    QPointF      pt2(width() / 2, -height());
```

```

QPointF      pt3(width() / 2, 2 * height());
QPointF      pt4(0, height() / 2);
path.moveTo(pt1);
path.cubicTo(pt2, pt3, pt4);

QRect rect(width() / 4, height() / 4, width() / 2, height() / 2);
path.addRect(rect);
path.addEllipse(rect);

QPainter painter(this);
painter.setRenderHint(QPainter::Antialiasing, true);
painter.setPen(QPen(Qt::blue, 6));
painter.drawPath(path);
}

...

```

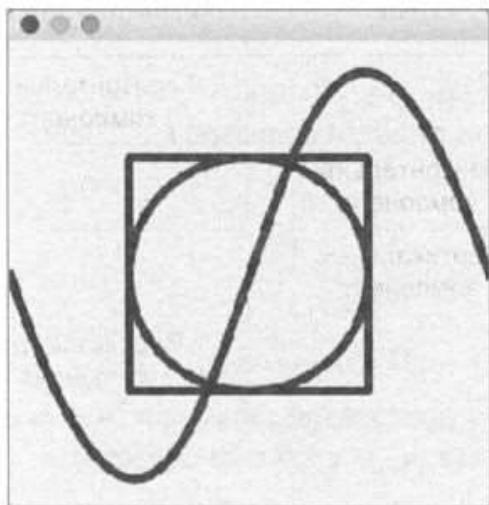


Рис. 18.20. Отображение графической траектории

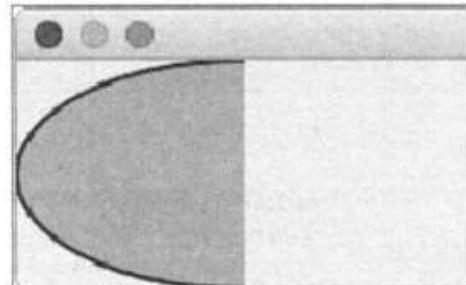


Рис. 18.21. Отсечение фигуры эллипса прямоугольной областью

Отсечения

Отсечения ограничивают вывод графики определенной областью (многоугольником или эллипсом). Если осуществляется попытка рисования за этими пределами, то оно будет невидимым. Установка прямоугольной области отсечения выполняется с помощью метода `setClipRect()`. Листинг 18.19 демонстрирует отсечение фигуры эллипса прямоугольной областью (рис. 18.21).

Листинг 18.19. Отсечение фигуры эллипса прямоугольной областью

```

...
void MyWidget::paintEvent (QPaintEvent*)
{
    QPainter painter(this);
    painter.setRenderHint(QPainter::Antialiasing, true);
    painter.setClipRect(0, 0, 100, 100);
    painter.setBrush(QBrush(Qt::green));

```

```

painter.setPen(QPen(Qt::black, 2));
painter.drawEllipse(0, 0, 200, 100);
}
...

```

Более сложные области отсечения устанавливаются методами `QPainter::setClipRegion()` и `QPainter::setClipPath()`.

В метод `setClipRegion()` передается объект класса `QRegion`. В конструкторе класса можно задать область в виде прямоугольника или эллипса. Например, следующий вызов создаст прямоугольную область с координатами левого верхнего угла (10, 10), а также шириной и высотой, равными 100:

```
QRegion region(10, 10, 100, 100);
```

Область отсечения в виде эллипса, вписанного в такой прямоугольник, создается следующим образом:

```
QRegion region (10, 10, 100, 100, QRegion::Ellipse);
```

В качестве области отсечения можно использовать и многоугольник, передав его в конструктор. Точки в многоугольнике можно установить при помощи оператора `<<`. Например:

```

QRegion region(QPolygon() << QPoint(0, 100)
                << QPoint(100, 100)
                << QPoint(100, 0)
                << QPoint(0, 0)
            );

```

Объекты класса `QRegion` можно комбинировать друг с другом, создавая при помощи методов `united()`, `intersected()`, `subtracted()` и `xored()` довольно сложные области:

- ◆ метод `united()` возвращает область, полученную в результате объединения двух областей;
- ◆ метод `intersected()` возвращает область, полученную в результате пересечения двух областей;
- ◆ метод `subtracted()` возвращает область, полученную в результате вычитания аргумента из исходной области;
- ◆ метод `xored()` возвращает область, содержащую точки из каждой области, но не из обеих областей.

Например:

```

QRegion region1(10, 10, 100, 100);
QRegion region2(10, 10, 100, 100, QRegion::Ellipse);
QRegion region3 = region1.subtract(region2);
painter.setClipRegion(region3);

```

Режим совмещения (composition mode)

Под *режимом совмещения* понимается задание способа совмещения пикселя источника с целевым пикселом при рисовании. Пиксели эти могут иметь различный уровень прозрачности.

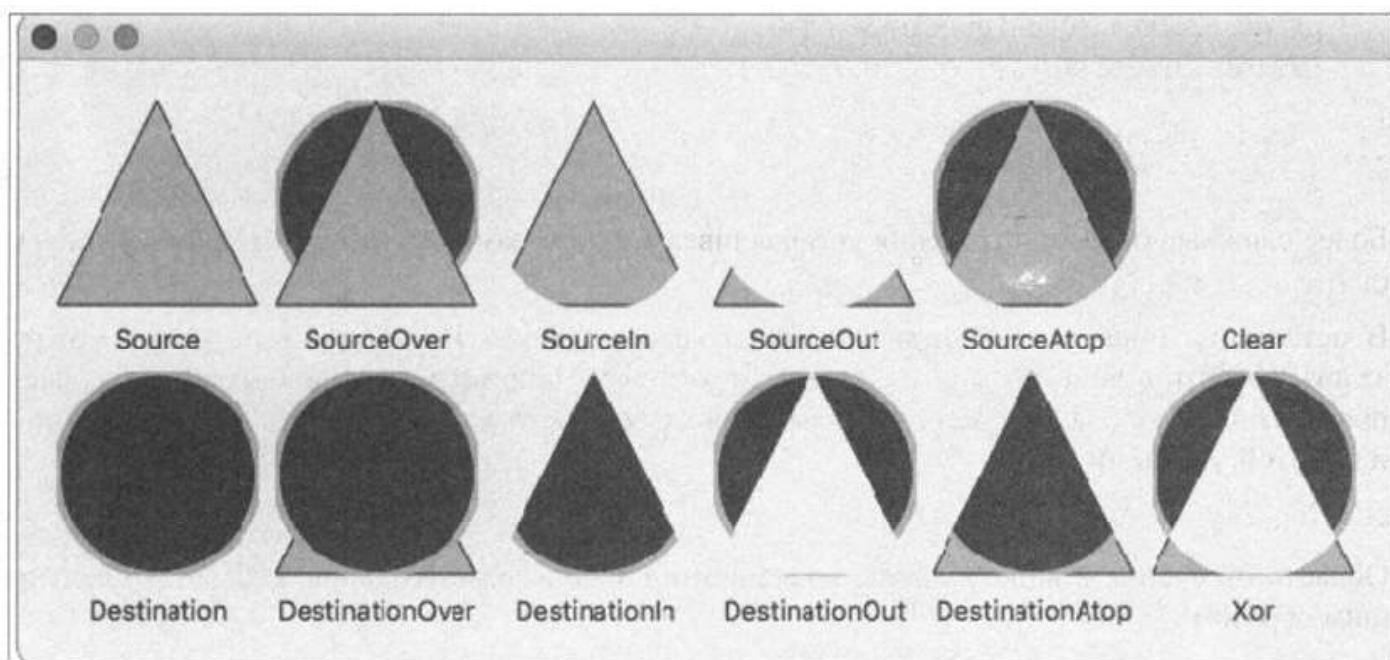


Рис. 18.22. Режимы совмещения

Такой механизм применяется для всех операций рисования, включая кисть, перо, градиенты и растровые изображения. Возможные операции проиллюстрированы на рис. 18.22.

Режим совмещения устанавливается с помощью метода `QPainter::setCompositionMode()`. По умолчанию режимом совмещения является «источник сверху»: `QPainter::CompositionMode_SourceOver`.

Если вам понадобится прохождение по пикселям растрового изображения, чтобы манипулировать альфа-каналом или цветовыми значениями, то прежде всего проверьте, нет ли для решения вашей задачи подходящего режима совмещения.

Функция `lbl()`, приведенная в листинге 18.20, предназначена для создания виджетов надписей с установленными растровыми изображениями, иллюстрирующими результат определенной операции совмещения. В виджете надписи методом `setFixedSize()` устанавливается неизменяемый размер, который впоследствии служит для инициализации объекта прямоугольной области `rect`. Этот объект используется для задания размеров исходного (`sourceImage`) и результирующего (`resultImage`) объекта растрового изображения. В качестве исходного изображения мы рисуем, вызывая метод `QPainter::drawPolygon()`, треугольник, заполненный зеленым цветом. В результирующем объекте растрового изображения вызовом метода `QPainter::drawEllipse()` рисуется окружность, заполненная красным цветом. Затем методом `QPainter::setCompositeMode()` устанавливается режим совмещения, переданный в функцию `lbl()` в переменной `mode`, и растровое (`sourceImage`) изображение рисуется в результирующем изображении методом `QPainter::drawImage()`. Метод `QLabel::setPixmap()` устанавливает результирующее растровое изображение в виджете надписи. В конце возвращается указатель на созданный виджет надписи.

Листинг 18.20. Функция `lbl()` — создание виджетов надписей с установленными растровыми изображениями (файл `main.cpp`)

```
QLabel* lbl(const QPainter::CompositionMode& mode)
{
    QLabel* plbl = new QLabel;
    plbl->setFixedSize(100, 100);
    plbl->paintEvent = &paintEvent;
    plbl->setCompositionMode(mode);
    return plbl;
}
```

```

QRect rect(plbl->contentsRect());
QPainter painter;

QImage sourceImage(rect.size(), QImage::Format_ARGB32_Premultiplied);
sourceImage.fill(QColor(0, 0, 0, 0));
painter.begin(&sourceImage);
painter.setRenderHint(QPainter::Antialiasing, true);
painter.setBrush(QBrush(QColor(0, 255, 0)));
painter.drawPolygon(QPolygon() << rect.bottomLeft()
                    << QPoint(rect.center().x(), 0)
                    << rect.bottomRight()
                    );
painter.end();

QImage resultImage(rect.size(), QImage::Format_ARGB32_Premultiplied);
painter.begin(&resultImage);
painter.setRenderHint(QPainter::Antialiasing, true);
painter.setCompositionMode(QPainter::CompositionMode_SourceOver);
painter.setPen(QPen(QColor(0, 255, 0), 4));
painter.setBrush(QBrush(QColor(255, 0, 0)));
painter.drawEllipse(rect);
painter.setCompositionMode(mode);
painter.drawImage(rect, sourceImage);
painter.end();

plbl->setPixmap(QPixmap::fromImage(resultImage));
return plbl;
}

```

В основной функции, приведенной в листинге 18.21, создаются виджеты надписей с изображениями (вызовы функций `lbl()`) и поясняющими надписями, описывающими примененный режим совмещения. Все элементы размещаются при помощи менеджеров компоновки (`layout`) табличного размещения (`pgrd`) на поверхности виджета (`wgt`).

Листинг 18.21. Создание виджетов надписей с изображениями — основная функция (файл main.cpp)

```

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget wgt;

    //Layout Setup
    QGridLayout* pgrd = new QGridLayout;
    pgrd->addWidget(lbl(QPainter::CompositionMode_Source), 0, 0);
    pgrd->addWidget(new QLabel("<CENTER>Source</CENTER>"), 1, 0);
    pgrd->addWidget(lbl(QPainter::CompositionMode_SourceOver), 0, 1);
    pgrd->addWidget(new QLabel("<CENTER>SourceOver</CENTER>"), 1, 1);
    pgrd->addWidget(lbl(QPainter::CompositionMode_SourceIn), 0, 2);
    pgrd->addWidget(new QLabel("<CENTER>SourceIn</CENTER>"), 1, 2);
    pgrd->addWidget(lbl(QPainter::CompositionMode_SourceOut), 0, 3);

```

```
pgrd->addWidget (new QLabel("<CENTER>SourceOut</CENTER>"), 1, 3);
pgrd->addWidget (lbl(QPainter::CompositionMode_SourceAtop), 0, 4);
pgrd->addWidget (new QLabel("<CENTER>SourceAtop</CENTER>"), 1, 4);
pgrd->addWidget (lbl(QPainter::CompositionMode_Clear), 0, 5);
pgrd->addWidget (new QLabel("<CENTER>Clear</CENTER>"), 1, 5);
pgrd->addWidget (lbl(QPainter::CompositionMode_Destination), 2, 0);
pgrd->addWidget (new QLabel("<CENTER>Destination</CENTER>"), 3, 0);
pgrd->addWidget (lbl(QPainter::CompositionMode_DestinationOver), 2, 1);
pgrd->addWidget (new QLabel("<CENTER>DestinationOver</CENTER>"), 3, 1);
pgrd->addWidget (lbl(QPainter::CompositionMode_DestinationIn), 2, 2);
pgrd->addWidget (new QLabel("<CENTER>DestinationIn</CENTER>"), 3, 2);
pgrd->addWidget (lbl(QPainter::CompositionMode_DestinationOut), 2, 3);
pgrd->addWidget (new QLabel("<CENTER>DestinationOut</CENTER>"), 3, 3);
pgrd->addWidget (lbl(QPainter::CompositionMode_DestinationAtop), 2, 4);
pgrd->addWidget (new QLabel("<CENTER>DestinationAtop</CENTER>"), 3, 4);
pgrd->addWidget (lbl(QPainter::CompositionMode_Xor), 2, 5);
pgrd->addWidget (new QLabel("<CENTER>Xor</CENTER>"), 3, 5);
wgt.setLayout (pgrd);

wgt.show();

return app.exec();
}
```

Графические эффекты

Графические эффекты можно применять к любым виджетам, а также и к объектам класса `QGraphicsItem` (см. главу 21). Устанавливаются эффекты при помощи метода `setGraphicsEffect()`. Основным классом в иерархии графических эффектов является класс `QGraphicsEffect`. Всего доступно четыре эффекта: размытие (`blur`), расцвечивание (`colorization`), тень (`drop shadow`) и непрозрачность (`opacity`). Каждый из этих эффектов реализован в отдельном классе, каждый такой класс унаследован от класса `QGraphicsEffect` (рис. 18.23).

Если четырех эффектов для вас недостаточно, вы можете реализовать класс собственного эффекта, — для этого необходимо унаследовать класс `QGraphicsEffect` и перезаписать метод `draw()`, потому что именно этот метод вызывается всякий раз, когда эффект нуждается в перерисовке. В методе `draw()` в вашем распоряжении будет указатель на объект `QPainter`, с помощью которого можно выполнить все необходимые графические операции.

Кроме того, графические эффекты можно очень удачно комбинировать с анимацией (см. главу 22).

Покажем использование трех эффектов на примере, приведенном в листингах 18.22 и 18.23 (рис. 18.24).

Функция `lbl()`, приведенная в листинге 18.22, создает виджеты надписей сразу с растровыми изображениями. В нее передается объект эффекта, который применяется вызовом метода `setGraphicsEffect()`. В случае, если указатель на объект эффекта равен нулевому значению, то это означает, что объект эффекта отсутствует, и метод применения эффекта вызываться не должен. В завершение из метода возвращается указатель на созданный виджет надписи.

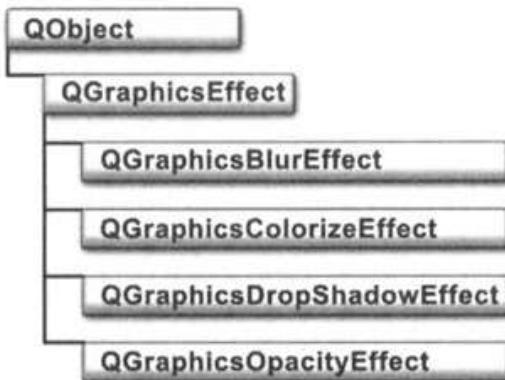


Рис. 18.23. Классы графических эффектов

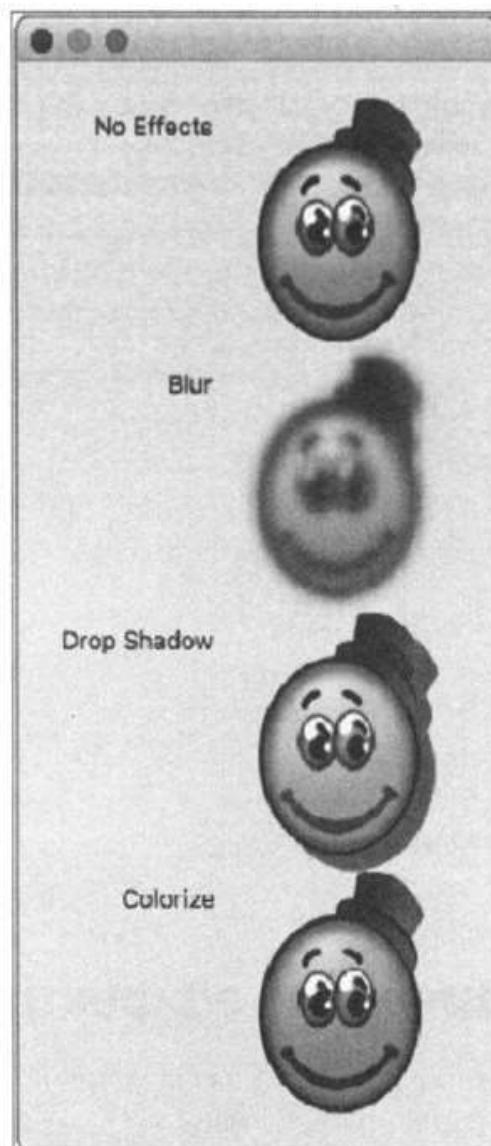


Рис. 18.24. Демонстрация трех графических эффектов: размытие (Blur), тень (Drop Shadow) и расцвечивание (Colorization)

Листинг 18.22. Функция `lbl()` — создание виджетов надписей с установленными растровыми изображениями (файл `main.cpp`)

```

QLabel* lbl(QGraphicsEffect* pge)
{
    QLabel* plbl = new QLabel;
    plbl->setPixmap(QPixmap(":/happyos.png"));

    if (pge) {
        plbl->setGraphicsEffect(pge);
    }
    return plbl;
}
  
```

В основной функции программы, приведенной в листинге 18.23, создаются объекты трех эффектов (указатели `pBlur`, `pShadow` и `pColorize`). При помощи табличного размещения `QFormLayout` мы размещаем надписи с виджетами, к которым были применены эффекты.

Листинг 18.23. Создание объектов эффектов — основная функция (файл main.cpp)

```
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget wgt;
    QGraphicsBlurEffect* pBlur = new QGraphicsBlurEffect;
    QGraphicsDropShadowEffect* pShadow = new QGraphicsDropShadowEffect;
    QGraphicsColorizeEffect* pColorize = new QGraphicsColorizeEffect;

    //Layout Setup
    QFormLayout* pform = new QFormLayout;
    pform->addRow("No Effects", lbl(0));
    pform->addRow("Blur", lbl(pBlur));
    pform->addRow("Drop Shadow", lbl(pShadow));
    pform->addRow("Colorize", lbl(pColorize));
    wgt.setLayout(pform);

    wgt.show();

    return app.exec();
}
```

Резюме

Объект класса QPainter выполняет рисование на объектах классов, унаследованных от класса QPaintDevice. Класс QPainter предоставляет методы для рисования точек, линий, эллипсов, растровых изображений, текста (см. главу 20) и др. Для рисования класс QPainter предоставляет перо и кисть. Перо служит для рисования дуг, линий и контуров замкнутых фигур. Кисти используются для заполнения внутренней части фигуры. Все команды рисования можно сохранять в объектах класса QPicture, который, в свою очередь, позволяет сохранять команды в файле и считывать их оттуда.

Эффект мерцания возникает в том случае, когда пиксели за короткие промежутки времени перерисовываются разными цветами. Использование механизма двойной буферизации, встроенного в Qt, автоматически подавляет этот нежелательный эффект.

Qt поддерживает три типа градиентов: линейный, конический и радиальный. С их помощью можно придать объемность некоторым элементам вашей программы.

Режим сглаживания позволяет улучшить визуальный эффект, убирая ступенчатость на контурах выводимых геометрических фигур.

Класс QPainter предоставляет возможность проведения геометрических преобразований, таких как перемещение, поворот, масштабирование и скос.

Отсечения используются для ограничения вывода графики заданной областью. Класс QRegion служит для задания областей, которые могут иметь очень сложные формы.

Графическая траектория позволяет задавать произвольные формы геометрических фигур, соединяя геометрические фигуры вместе.

Режим совмещения задает способ объединения пикселя источника и целевого пикселя.

При помощи объектов, унаследованных от класса `QGraphicsEffect`, можно легко применять к виджетам различные графические эффекты.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: <http://qt-book.com/ru/18-510/> или с помощью следующего QR-кода (рис. 18.25):

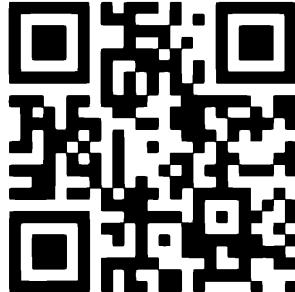


Рис. 18.25. QR-код для доступа на страницу комментариев к этой главе