QT — технология для разработки кросс платформенных приложений, реализуемых на языке C++. Компания Nokia выпустила отличную технологию и отличную IDE к ней, которые, на мой взгляд, являются удачным сочетанием простоты написания программ (вам достаточно знать ООП и C++) и одновременной «контролируемости» (доступны все библиотеки для их просмотра). В результате получается оптимизированное приложение, в котором подключено, только то, что необходимо. Конечно в нем нет некоторых готовых реализаций, например события получения фокуса, однако есть все для его реализации.  
  
При изучении QT можно натолкнуться на глобальную проблему реализации графики. А если попробуете искать в Google, то можно найти форум, на котором сидят гуру, и «посылают» в тот же Google, но конечно есть исключения. Макс Шлее пишет в своей замечательной книге об этом, но это больше похоже на нервные наброски. QT Assistant — отличная вещь, но нет готового примера, чтоб просто разобраться. В общем я разобрался благодаря тому, что использовал все три источника и знание ООП. Конечно можно сказать, что при желании можно разобраться легко в этом, но я хочу чтоб человек зашел в Google, набрал «Рисование в QT» и вышел на статью, в которой будет все описано рассказано на примере.

Идеология

В QT не так как в .NET (подключите библиотеку для одного класса и заодно кучу других в подарок и получите машину для поглощения RAM). Идеология в QT следующая: надо создать класс, наследующий public от QWidget. Мы получим класс объекта на котором будем рисовать. Класс можно создать в отдельном заголовочном файле, но я создл его в основном «wigdet.h»:  
class canvas : public QWidget {  
Q\_OBJECT  
public:  
canvas(QWidget\* parent=0):QWidget(parent){}  
~canvas(){}  
void paintEvent(QPaintEvent\*); //вызов метода, который собственно и будет рисовать то, что вы запрограммируете далее  
bool par; /\* если вы захотите рисовать по событию, например нажатие кнопки изменит значение этого параметра. А потом Вы можете сделать какое-нибудь условие на этот параметр, при выполнении которого будет что-нибудь рисоваться. \*/  
  
protected:  
};

Создание объекта для рисования и программирование метода для рисования

Далее после того, как Вы создали класс, можно приступить к декларации объекта и вызова метода. Все это можно делать где у вас проходят события. У меня это «widget.cpp». Декларацию можно осуществить по какому-нибудь событию или по загрузке программы. Она выглядит следующим обрзом  
canvas\* wt = new canvas; //не знаю почему не работает canvas\* wt = new canvas(ui->tab);  
wt=new canvas(ui->tab); //ui->tab\_5 является родительским окном, на расположится наш объект  
wt->show(); //функция говорит сама за себя  
wt->setGeometry(0,210,500,150); // габариты объекта  
  
  
После создания объекта нам необходимо запрограммировать само рисование, то есть метод void paintEvent(QPaintEvent\*). По своей сути он является событием, даже название об этом говорит. Поэтому он программируется отдельно как и любое событие. Желательно его программировать тоже в основном файле с событиями, так как удобно обратиться к объектам GUI. Тут все просто:  
void canvas::paintEvent(QPaintEvent \*)  
{  
QPainter img(this);  
img.drawLine(0,0,150,75); /\* вместо drawLine(0,0,150,75) вы можете вызвать любой метод, который есть в QPainter. \*/  
}

**QT: начинаем работать с графикой**

*Статья "нулевого уровня", но пару примеров кода содержит :)*

**1. Общие принципы**

Общий порядок работы с графикой в QT похож на другие современные среды программирования. В основе работы лежит взаимодействие 3 классов:

* QPainter – "рисовальщик" QT, класс-исполнитель команд рисования;
* QPaintEngine – "движок" рисования, обычно не используемый из кода непосредственно. Он бывает нужен программисту лишь при создании собственных контекстов рисования, отсутствующих в системе;
* QPaintDevice – контекст рисования (канва), который можно понимать как графическое "полотно", состоящее из пикселов.

В обычных случаях программе достаточно "захватить" адрес объекта контекста, отрисовать нужные графические примитивы и освободить контекст:

QPainter painter1; //создание рисовальщика

painter1.begin(this); //захват контекста

painter1.fillRect(0,0,width(),height(),Qt::CrossPattern); //отрисовка

painter1.end();//освобождение контекста

Здесь рисуется заштрихованный прямоугольник на канве текущего виджета. Существенно, что этот код должен выполняться из метода paintEvent, который может быть переопределён в любом пользовательском виджете, являющемся наследником имеющего канву класса (см.п.2).

Как и в других подобных графических системах, для рисования используются 2 основных инструмента:

* Перо QPen – для рисования контуров;
* Кисть QBrush – для заполнения контуров цветом.

Непосредственно отрисовкой занимаются методы класса QPainter с названиями на draw (drawLine, drawRect, drawPolygon, drawEllipse) или fill (fillRect, fillPath). Кроме настроек пера и кисти, этим методам часто требуется информация о координатах точек, размерах прямоугольников, текущих цветах и т.п. Для хранения всей этой информации в QT предусмотрен ряд геометрических классов, которые ничего не рисуют, но хранят описания размеров или расположений графических объектов. Большинство геометрических классов реализованы "дважды", например, класс QPoint работает с целочисленными координатами точек на плоскости, а QPointF – с вещественными:

|  |  |  |
| --- | --- | --- |
| Классы | Описание | Основные методы |
| QPoint,  QPointF | Целочисленные и вещественные координаты точки на плоскости | .x(), .y() – получение координат;  .setX(), .setY() – установка координат.  В классах определены сравнения, арифметические действия над координатами |
| QSize,  QSizeF | Размер, то есть, совокупность ширины и высоты | .width(), .height() – получить размеры;  .setWidth(), .setHeight() – установить размеры;  .scale() – масштабировать размер. |
| QRect,  QRectF | Прямоугольник, фактически, это "точка+размер" | См. QPoint, QSize |
| QLine,  QLineF | Отрезки на плоскости | .x1(), .y1(), .x2(), .y2() – координаты начала и конца отрезка;  .dx(), .dy() – проекции на оси 0X и 0Y |
| QPolygon,  QPolygonF | Многоугольник на плоскости, фактически, массив точек (координат вершин) | << добавить точку QPoint  .point(i) – вернуть i-ую точку |
| QColor | Цвет в модели RGB (или HSV) | .setRgb(r,g,b) установить интенсивности цветов;  .red(), .green, .blue(), .alpha() – получить интенсивности цветов и прозрачность |

В классе QPainter также определены удобные методы для манипулирования с системой координат:

* translate – сдвинуть начало координат в указанную точку;
* scale – масштабировать систему координат;
* rotate – повернуть систему координат;
* shear – исказить систему координат (выполнить скос);
* save, restore – сохранить/восстановить состояние рисовальщика.

В большинстве случаев при отрисовке объектов удобнее пользоваться этими методами, чем высчитывать всё в "абсолютных координатах" канвы. Например, мы не меняли геометрию треугольника, определённого в функции drawTriangle и не высчитывали при каждой отрисовке новых координат его вершин:

void drawTriangle (QPainter &ptr, QPoint point) {

QPolygon q;

q << QPoint(0,0) << QPoint(100,0) << QPoint(50,100);

ptr.save();

ptr.translate(point); //сдвинули начало координат в точку, указанную параметром point

ptr.drawPolygon(q);

ptr.restore();

}

void QPaintWidget::paintEvent(QPaintEvent \*event) {

QPainter painter1;

painter1.begin(this);

drawTriangle (painter1,QPoint(100,100));

drawTriangle (painter1,QPoint(200,200));

//...

painter1.end();

}

**2. Работа с графической канвой**

Создадим проект на основе класса QWidget (добавились файлы main.cpp, widget.h, widget.cpp). Так как отрисовка выполняется по событию paintEvent, в заголовочном файле widget.h нам понадобится добавить прототип виртуального метода paintEvent, предусмотренного в каждом виджете. Чтобы виджет мог рисовать на канве, он должен переопределить этот метод:

#include <QWidget>

#include <QPainter>

class Widget : public QWidget

{

Q\_OBJECT

//...

protected:

void paintEvent(QPaintEvent \*);

В файле widget.cpp нам остается написать реализацию метода. Покажем простейшие действия с кистью, пером и строкой текста:

void Widget::paintEvent(QPaintEvent \*event) {

QPainter painter(this); //новый объект "рисовальщика"

painter.setPen (QPen(Qt::red,Qt::SolidLine));

//создать и установить перо - красная сплошная линия

painter.drawLine(0,0,width(),height());

//нарисовать линию через рабочую область формы

painter.setBrush(QBrush(Qt::green,Qt::SolidPattern));

//создать и установить кисть - зелёная слошная заливка

QPoint center(width()/2,height()/2);

int rad = qMin(width()/4,height()/4);

painter.drawEllipse(center,rad,rad);

//нарисовать окружность по центру

painter.setFont(QFont("sans-serif",-1,10));

//установить шрифт заданного начертания и размера 10 пт

QRect rect(center.x()-rad,center.y()-rad,rad\*2,rad\*2);

painter.drawText(rect, Qt::AlignCenter, tr("Hello,\nworld!"));

//вывели строку текста, выравненную по центру

}

Теперь класс Widget может рисовать на своей канве. Так как событие paintEvent происходит, в том числе, при изменении размеров окна виджета, картинка всё время будет соответствовать окну формы.

На самом деле, конечно, painter надо прописать в заголовочном файле класса, инициализировать в конструкторе класса, а не создавать каждый раз заново при перерисовке формы. Здесь и выше так сделано только для краткости примера.

**3. Работа с изображениями**

QT может как записывать, так и загружать файлы основных растровых форматов, включая PNG, BMP, ICO, TIFF, JPEG, GIF и некоторые другие. При этом поддерживается контекстно-независимое представление графики. Фактически, это означает, что данные изображений помещаются в массивы, содержащие данные об отдельных пикселах рисунка. Основным классом представления изображений является QImage. Этот класс унаследован от контекста рисования QPaintDevice, что позволяет использовать все методы рисования, определённые в QPainter. Метод класса format() позволяет узнать формат текущего изображения, а метод convertToFormat() – изменить его, вернув новый объект класса QImage. Некоторые значения перечисления Format указаны ниже:

* Format\_Invalid - формат неверен;
* Format\_Mono – монохромное изображение с 1 битом на пиксел;
* Format\_Index8 – данные представляют собой 8-битные индексы цветовой палитры;
* Format\_RGB32 – каждый пиксел представлен 32 битами (интенсивности красного, зелёного и синего, плюс значение альфа-канала, всегда равное 0xFF, то есть, прозрачность не поддерживается);
* Format\_ARGB32 – 32 бита на пиксел с поддержкой альфа-канала прозрачности.

Для создания изображения в конструктор класса достаточно передать его размеры в пикселах и формат:

QImage img(width(),height(),QImage::Format\_RGB32);

Загрузить изображение можно, передав конструктору путь к нужному файлу:

QImage img("my.jpg");

или воспользовавшись методом load:

QImage img; img.load("my.jpg");

QPainter painter(this);

painter.drawImage(0,0,img);

Как видно из кода, отобразить изображение можно методом drawImage класса QPainter. Показать часть изображения можно с помощью дополнительных параметров метода.

Папка, которая является текущей, в общем случае зависит от проекта. Обычно это та папка, где находится файл Makefile проекта.

Сохранить изображение может метод save:

QImage img(width(),height(),QImage::Format\_RGB32);

img.save("my2.jpg","JPG");

Сохранение произойдёт в ту папку, которая является текущей в вашей конфигурации проекта.

Для чтения отдельных пикселов удобен метод pixel:

QRgb pixel = img.pixel(100,100);

QString str; str.setNum(pixel);

setWindowTitle(str);

Записать пикселы можно с помощью метода setPixel:

QPainter painter;

painter.begin(this);

QImage img(width(),height(),QImage::Format\_RGB32);

QRgb rgb;

for (int x=0; x<width(); x++) {

for (int y=0; y<height(); y++) {

int c = qRound(y/255.\*100);

rgb = qRgb(c,c,c);

img.setPixel (x,y,rgb);

}

}

painter.drawImage(0,0,img);

painter.end();

Показанный код рисует линейный градиент в окне формы, для работы нужны директивы

#include <QPainter>

#include <qmath.h>

В классе QImage реализован ряд методов для редактирования изображений:

* invertPixels – позволяет инвертировать цвета пикселов и/или альфа-канал;
* scaled – позволяет масштабировать изображение;
* mirrored – выполняет зеркальное отражение картинки по горизонтали и/или вертикали.

Класс контекстно-зависимого представления изображений QPixmap также унаследован от QPaintDevice. Его использование целесообразно там, где нужен промежуточный буфер для рисования или критична скорость отрисовки графических объектов. Существует также отдельный класс-потомок QBitmap для работы с монохромными изображениями.