**Практика 6.**

# Система Qt Graphics View Framework. QGraphicsView. QGraphicsScene.  QGraphicsItem. [QAbstractGraphicsShapeItem](qthelp://org.qt-project.qtwidgets.511/qtwidgets/qabstractgraphicsshapeitem.html). QGraphicsRectItem. QGraphicsEllipseItem. Стандартныедиалоги. Пользовательские диалоги. QTableWidget. Dock-виджеты.

## Задача

Создадим простейший графический редактор, с помощью которого можно будет рисовать прямоугольники и эллипсы с контуром:

* разного цвета
* разной толщины
* с разным стилем пера (сплошная, пунктирная…).

При отрисовке прямоугольника или эллипса:

* Рамка рисуется пером QPen(сформированным по умолчанию или выбранным в контексте устройства). По умолчанию перо имеет единичную толщину (1 пиксел), черного цвета, сплошное.
* Фон заполняется кистью QBrush (аналогично). По умолчанию кисть - белая

Замечание: задание фона предусмотрите самостоятельно.

### Решая такую задачу «в лоб», можно:

* Создать вспомогательный класс MyRect , в котором
* объявить переменную типа QRect + цвет + толщину + стиль
* реализовать метод для рисования данной линии, например drawMyRect(QPainter\*)

В классе виджета, предназначенного для рисования:

* Завести контейнер объектов типа MyRect\*
* Перегрузить виртуальный метод paintEvent(), в котором отрисовывать все прямоугольники
* Перегрузить виртуальные обработчики событий мыши.
* Зафиксировать в обработчике mousePressEvent() левый верхний угол создаваемого прямоуголника,
* изменять координаты правого нижнего угла в mouseMoveEvent(),
* в обработчике mouseReleaseEvent() закончить рисование.

В результате в контейнер должен быть добавлен указатель на сформированный объект типа MyRect

При таком подходе нужно:

1. явно предусмотреть перерисовку всех графических примитивов (прямоугольников) – в методе paintEvent()
2. явно инициировать перерисовку при изменениях – repaint()
3. явно трансформировать изображение при изменении размеров виджета (центрировать,…)
4. задавать прокрутку в случае, когда логические размеры изображения превышают размеры виджета
5. …

Но! Для решения всех перечисленных проблем и добавления дополнительных возможностей используется система **GraphicsViewFramework**, которая:

* предназначена для вывода двухмерной графики (как векторной, так и растровой)
* реализована согласно парадигме «Модель-Вид-Контроллер» (MVC)
* легко позволяет создавать не только статические, но и динамические сцены, что в свою очередь облегчает взаимодействие с пользователем
* …

## Архитектура Модель-Вид-Контроллер.

В терминах MVC в нашей задаче:

* Сцена представляет собой модель (совокупность нарисованных примитивов + действия над графическими примитивами).
* Контроллер и вид совмещены. За визуализацию данных и взаимодействие с пользователем отвечает вид

## Система Graphics View Framework

Сцена

Представление

указатель

указатель

указатель

указатель

Items:

отображение

Взаимодействие с пользователем

е

QGraphicsItem

**QGraphicsEllipseItem**

QAbstractGraphicsShapeItem

**QGraphicsRectItem**

QGraphicsPathtem

QGraphicsPolygonItem

QGraphicsSimpleTextItem

QGraphicsItemGroup

QGraphicsLineItem

QGraphicsPixmapItem

…

## Начинаем разрабатывать приложение

Создайте заготовку приложения на базе главного окна.

### QGraphicsView

В созданной QtCreator-ом заготовке нас не устраивает тип центрального виджета. По умолчанию в заготовке (которую создает визард) в качестве класса для центрального виджета используется QWidget, а нам хотелось бы использовать два QGraphicsView (для того, чтобы наблюдать как изменения в модели одновременно отражаются в двух представлениях)

### Нам нужно два представления данных, разделяемых сплиттером.

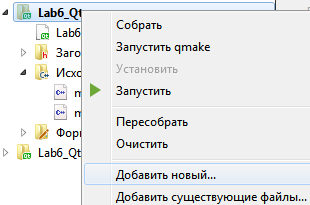
В режиме дизайнера:

* с панели инструментов добавьте на форму два Graphics View
* Выделите Graphics View и Graphics View (при нажатом CTRL) и в контекстном меню выберите «Скомпоновать по горизонтали с разделителем»
* Чтобы полученная компоновка «подстраивалась» под размеры центрального виджета, сделать активным central widget и выбрать горизонтальную компоновку

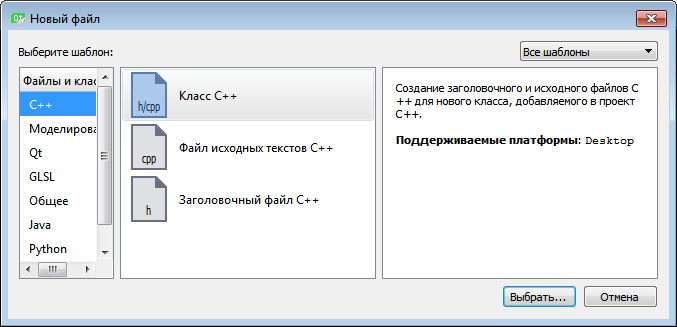


### QGraphicsScene

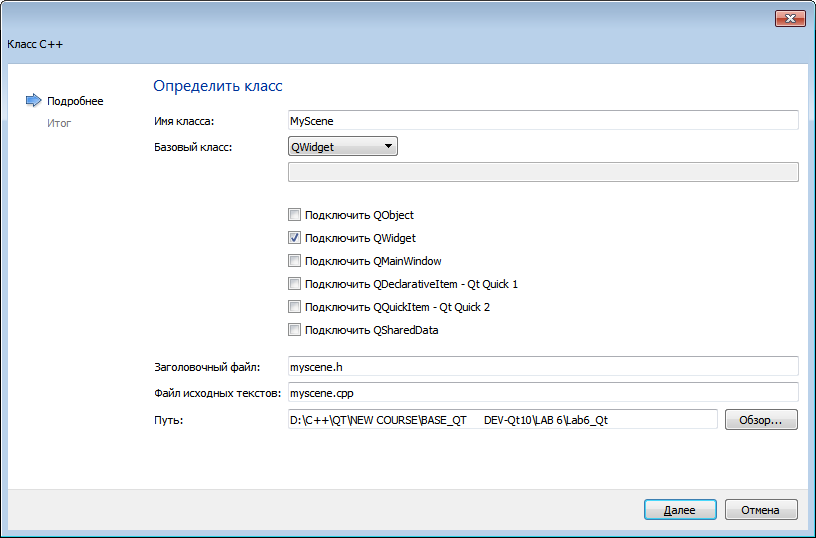
Создайте класс, производный от QGraphicsScene, например, MyScene. Это можно сделать, выбрав команду «Добавить новый…»



В следующем окне выберите команду «Класс С++».



Задайте имя класса и выберите базовый класс QWidget



При этом будет сгенерирован файл myscene.h:

#include <QWidget>

class MyScene : public QWidget

{

Q\_OBJECT

public:

explicit MyScene(QWidget \*parent = nullptr);

signals:

public slots:

};

И файл myscene.cpp:

#include "myscene.h"

MyScene::**MyScene**(QWidget \*parent) : QWidget(parent)

{

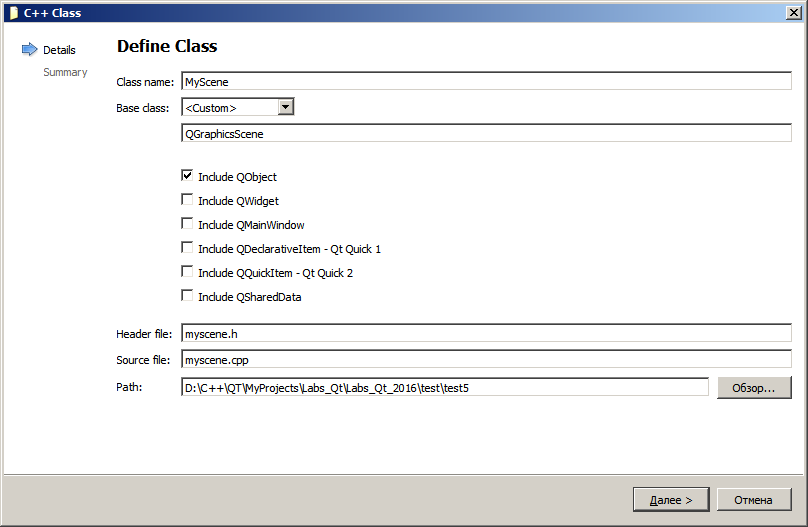
}

ПРИДЕТСЯ внести коррективы в сгенерированные файлы

1. В отмеченных местах кода необходимо заменить имя класса QWidget на QGraphicsScene.
2. **Очень важно**, чтобы в классе MyScene имелся макрос **Q\_OBJECT**.

Замечание: Внимание, возможна ошибка!!!

Если создать класс иным способом, например, так, то будет сгенерирован такой текст:



#include <QObject>

class MyScene : public QGraphicsScene{

public:

MyScene();

};

#include "myscene.h"

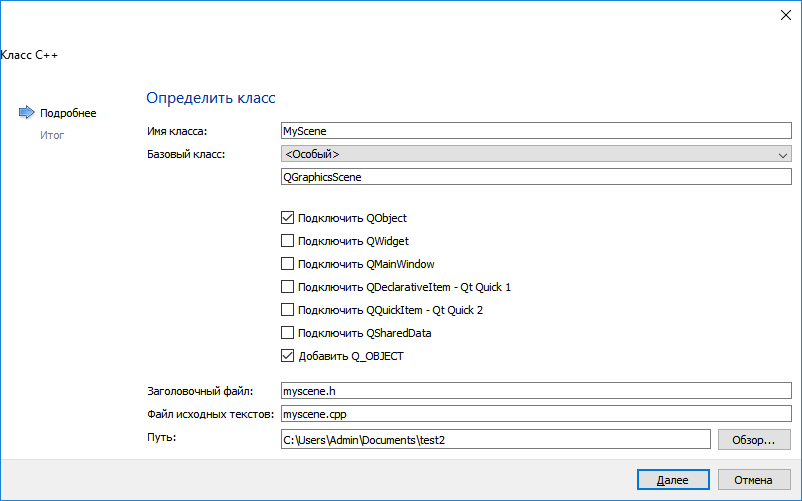
MyScene::MyScene()

{

}

* Нет макроса Q\_OBJECT
* Нет параметра parent в конструкторе

И даже, если Вы используете QtCreator последней версии () и окно создания класса иное и Вы поставили ручками все нужные флажки, корректировки в код придется вносить.



Важно:

* Объект **QGraphicsScene** содержит логические (не физические/оконные) размеры области рисования (или размеры модели). Размеры модели учитываются при отображении примитивов. Если Вы их явно не задаете, то по умолчанию логические размеры изображения будут установлены максимально возможными => при отображении всех примитивов QGraphicsView будет постоянно пытаться изменять начало координат, чтобы картинка выглядела «наилучшим» образом.

**Поэтому!**  
В конструкторе Вашего класса сцены, производного от QGraphicsScene (MyScene), лучше сразу **задать** разумные/приемлемые размеры рисунка, например (0, 0,500,500) посредством метода **QGraphicsScene ::setSceneRect()**

* Координаты сцены задаются плавающими значениями (в отличие от оконных целочисленных координат)
* Далее логично пользоваться исключительно координатами сцены (логическими)
* Если логические размеры сцены больше, чем физические размеры окна, QGraphicsView добавляет скроллинг
* Нет простого правила для установки соответствия между началом координат модели и началом координат окна => для преобразований нужно пользоваться методами класса

В класс MyScene **добавляем**:

1. Перегружаем **специализированные** виртуальные методы «мышиных» событий базового класса :  
   virtual void mousePressEvent (**QGraphicsSceneMouseEvent** \*mouseEvent);  
   virtual void mouseReleaseEvent (**QGraphicsSceneMouseEvent** \*mouseEvent);  
   virtual void mouseMoveEvent (**QGraphicsSceneMouseEvent** \*mouseEvent);  
     
   Важно! Обратите внимание, что все эти методы принимают специализированный тип **QGraphicsSceneMouseEvent.**

Этот класс, в частности, предоставляет Вам координаты курсора в трех разных системах координат:  
**scenePos()** возвращает логические или координаты модели,  
**pos()** – клиентские координаты элемента,  
**screenPos()** – оконные

1. Указатель на текущий рисуемый примитив:   
   **QAbstractGraphicsShapeItem\*current;**
2. Признак – «Идет рисование» -   
   **bool m\_drawingInProcess;**
3. Признак (и соответствующее перечисление, например ShapeType) – «что рисуем» (прямоугольник или эллипс). Этот признак будет формироваться пользователем
4. Объект типа **QPointF m\_startPoint;** для хранения координат левого верхнего угла прямоугольника (а «тянуть» будем **при нажатой правой кнопке** за правый нижний угол)
5. А также для «оживления» рисования нам понадобятся: цвет рамки (QColor), толщина рамки(uint) и ее стиль (Qt::PenStyle). (можно также задать цвет кисти **QBrush** для закрашивания)

При рисовании прямоугольника рамка рисуется пером, а внутренняя область заполняется кистью.

1. В конструкторе помимо задания логических размеров сцены не забудьте предусмотреть инициализацию переменных

### Связь QGraphicsView и QGraphicsScene

В конструкторе главного окна связываем сцену и виджеты-представления:

MyScene\*scene=new MyScene();//создаем **НАШУ** сцену

Например,

ui->graphicsView->setScene(scene); //и ассоциируем ее с QGraphicsView

## Рисование прямоугольников и добавление их в сцену

Используем тот факт, что классы-обертки для прямоугольников, эллипсов и SimpleText наследуют одному и тому же базовому классу QAbstractGraphicsShapeItem => в тех случаях, когда не нужно пользоваться функциональностью целевых классов, можно использовать методы базового.

Подсказка приведена для прямоугольников, а эллипсы предлагается реализовать **самостоятельно**!

Если пользователь выбрал прямоугольник, процесс рисования прямоугольника:

1. При нажатии **правой** кнопки мыши (так как левая задействована для выделения и перемещения примитива):
   1. зафиксируйте верхний левый угол в (запомните его в переменной m\_startPoint). Координаты можно получить посредством QGraphicsSceneMouseEvent::scenePos()
   2. **динамически** создайте графический объект для описания примитива-прямоугольника – QGraphicsRectItem, а возвращаемый методом указатель запомните в переменной базового типа – указателе m\_current. Когда мы начинаем рисовать, прямоугольник «стянут» в точку =>его ширина и высота ==0  
      m\_current=new QGraphicsRectItem(m\_startPoint.x(),m\_startPoint.y(),0.,0.);
   3. создайте перо с текущими атрибутами и ассоциируйте его с примитивом посредством QAbstractGraphicsShapeItem::**setPen**()
   4. Чтобы в дальнейшем можно было примитив перетаскивать и выделять (select), нужно вызвать метод   
      void QGraphicsItem::**setFlags**(GraphicsItemFlags flags)  
      с флагами QGraphicsItem::ItemIsMovable и QGraphicsItem::ItemIsSelectable
   5. Добавьте новый примитив m\_current в сцену –

QGraphicsScene:: **addItem**()

* 1. установите признак – «Рисуем»
  2. Важно! Для получения базовой функциональности (выделение, перетаскивание…) нужно **обязательно** вызвать метод базового класса QGraphicsScene::*mousePressEvent*(mouseEvent);

1. при перемещении мыши (**только** если происходит рисование):
   1. получаем **текущую** логическую координату курсора -   
      QGraphicsSceneMouseEvent::scenePos()
   2. формируем из m\_startPoint и текущей точки новый прямоугольник QRectF
   3. В классе сцены мы сохраняем QAbstractGraphicsShapeItem \* m\_current . Для того, чтобы задать параметры фигуры (прямоугольника или эллипса), нам нужно будет вызывать НЕ виртуальный метод  
      void QGraphicsRectItem::setRect(const QRectF& rectangle)

или (void QGraphicsEllipseItem::setRect(const QRectF& rectangle))  
В зависимости от типа рисуемой фигуры нужно привести тип указателя базового типа к требуемому производному типу посредством специализированного оператора явного приведения тип в **qgraphicsitem\_cast**

* 1. посредством QGraphicsRectItem::setRect(); изменяем в примитиве значения.

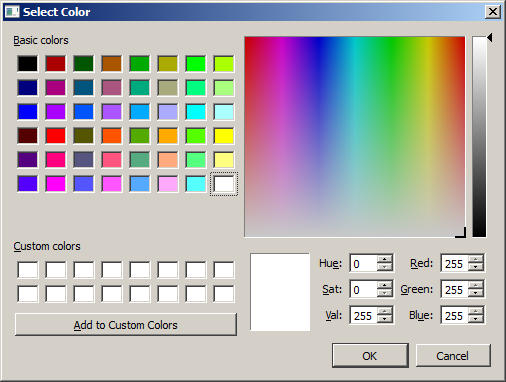
**Замечание:** в результате действий пользователя прямоугольник мог стать «ненормализованным», а хранить лучше в нормальном виде =>сохранять нужно не сформированный прямоугольник, а его нормализованную копию (метод QRectF QRectF::normalized()const возвращает нормализованный прямоугольник)

* 1. **Важно**! При изменении примитива **не нужно** явно вызывать перерисовку окна, так как все изменения отслеживает QGraphicsView и оптимизирует перерисовку
  2. Важно! Для получения базовой функциональности (выделение, перетаскивание…) нужно обязательно вызвать метод базового класса   
     QGraphicsScene::mouseMoveEvent(mouseEvent);

1. При **отпускании** мыши (**только** если происходит рисование) – «рисование текущего прямоугольника закончено»:
   1. Нужно сбросить признак
   2. И обнулить указатель m\_current
   3. Важно! Для получения базовой функциональности (выделение, перетаскивание…) нужно обязательно вызвать метод базового класса   
      QGraphicsScene::mouseReleaseEvent (mouseEvent);

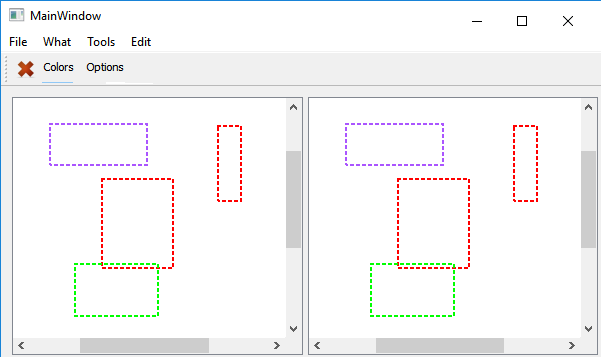
## Стандартный диалог выбора цвета–QColorDialog

Рисуем разными цветами. Цвет задаем с помощью стандартного диалога:



Этот диалог вызывается только модально. Для вызова используется статический метод QColorDialog::getColor(). Порядок действий:

* Вызов диалога должен инициировать пользователь =>предусмотрите пункт меню + дублирующую кнопку на toolbar-е.
* В классе MyScene реализуйте слот, в котором вызовите диалог. Подсказка: определить, какую кнопку нажал пользователь (Ok или Cancel) можно по значению возвращаемого цвета посредством метода QColor::isValid()
* Связать соответствующее действие со слотом можно в конструкторе MainWindow
* Теперь при создании текущего примитива цвет контура должен задаваться текущим значением



## Пользовательский модальный диалог

### Класс QDialog

Класс **QDialog** является базовым для всех диалоговых окон, представленных в классовой иерархии [Qt](http://qt-doc.ru/).

Хотя диалоговое окно можно создавать при помощи любого виджета (сделав его top-level виджетом, то есть в качестве родителя указать 0), тем не менее, удобнее воспользоваться классом **QDialog**, который предоставляет ряд возможностей, необходимых всем диалоговым окнам. Для создания заготовок удобно пользоваться QtDesigner-ом.

Диалоговые окна подразделяются на две группы:

* модальные
* немодальные.

Режим модальности и немодальности можно установить и узнать при помощи методов  
**QDialog::setModal()** и **QDialog::isModal()** соответственно. При установке true соответствует модальному режиму, a false — немодальному.

### Модальные диалоговые окна

Для блокировки приложения на время вызова диалога запускается цикл обработки событий, который «обслуживает» только диалоговое окно (все остальные события игнорируются). Этот цикл запускается вызовом слота **QDialog::exec()**, который возвращает значение целого типа после закрытия диалогового окна. Это значение информирует о нажатой кнопке и может принимать значения **QDialog::Accepted** или **QDialog::Rejected**, соответствующие кнопкам **Ok** и **Cancel** (Отмена).

Принцип вызова модального диалогового окна примерно следующий:

MyDialog dlg(&data);

if (dlg.exec() == QDialog::Accepted)

{

// Пользователь выбрал Accepted

// Получить данные для дальнейшего анализа и обработки

Data data = dlg.getData();

...

}

Или так:

MyDialog\* pdlg = new MyDialog(&data);

if (pdlg->exec() == QDialog::Accepted)

{

// Пользователь выбрал Accepted

// Получить данные для дальнейшего анализа и обработки

Data data = pdlg->getData();

...

}

delete pdlg;

### Замечание:

### Можно создать в классе пользовательского диалога статический метод (аналогично тому, как сделано в стандартных диалогах) и вызывать его для получения нужной информации, например, так:

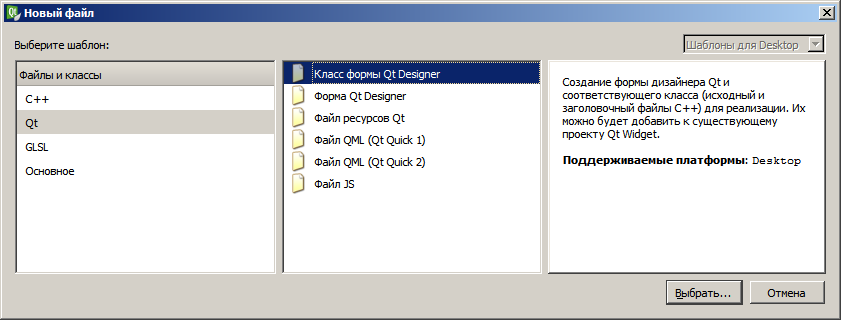
Data data = MyDialog::getData(&data);

### Подумайте, как это сделать.

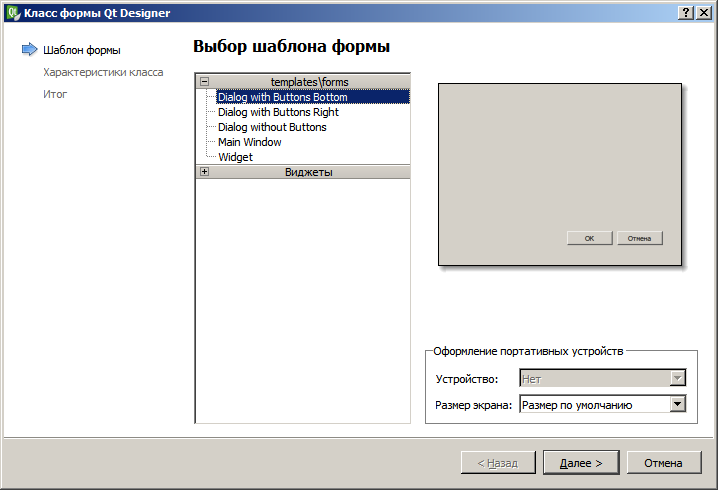
### Создаем свой пользовательский диалог

Хотелось бы иметь возможность одновременно редактировать все атрибуты рисования + видеть результат. Для этого создаем пользовательский диалог:

1. Добавить к проекту класс формы, для этого надо выполнить следующую последовательность действий:
   * Щелкнуть по имени проекта в окне дерева проекта правой кнопкой, в выпавшем меню выбрать пункт «Добавить новый»



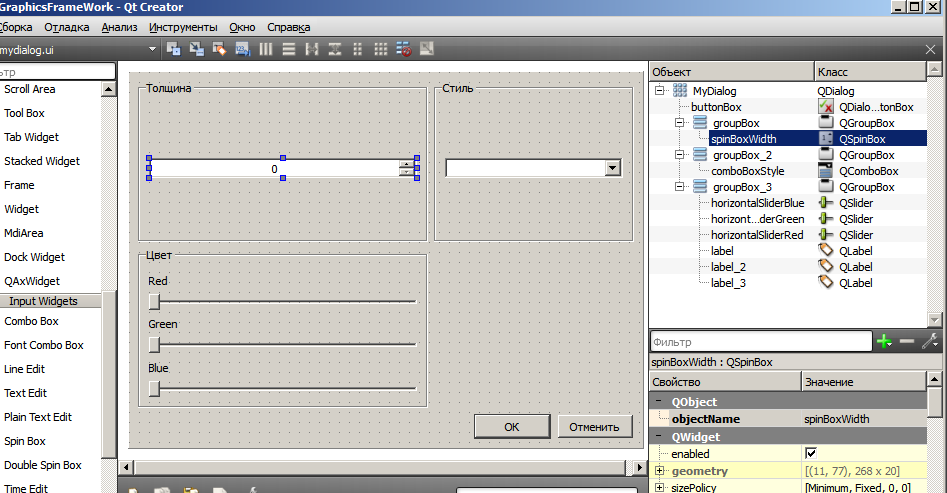
* + В следующем окне выбрать, например «Dialog with Buttons Bottom»



* + Следовать указаниям визарда  
    В результате будут созданы файлы:

dialog.ui, dialog.h, dialog.cpp и ui\_dialog.h

1. Наполнить диалог содержимым (интерфейсными элементами). Что-то вроде:



1. Большинство свойств можно настроить в дизайнере, например, внешний вид, диапазоны слайдеров, спина.
2. Для combobox инициализирующие действия можно произвести в конструкторе – каждому item-у сопоставить выводимый текст + ассоциировать с ним данное (в нашем случае - значение стиля)

Замечание:

А можно использовать map<QString, uint> или map<QString, Qt::PenStyle>

Или map< Qt::PenStyle, QString>

1. Для добавления данных в comboBox используется метод

void QComboBox::addItem(const [QString](qthelp://org.qt-project.qtwidgets.550/qtcore/qstring.html) &*text*, const [QVariant](qthelp://org.qt-project.qtwidgets.550/qtcore/qvariant.html) &*userData* = QVariant());

Замечание:

При работе с combobox удобно использовать методы:

int **[count](qthelp://org.qt-project.qtwidgets.550/qtwidgets/qcombobox.html" \l "count-prop)**() const;

QVariant [**currentData**](qthelp://org.qt-project.qtwidgets.550/qtwidgets/qcombobox.html#currentData-prop)(int*role* = Qt::UserRole) const;

int **[currentIndex](qthelp://org.qt-project.qtwidgets.550/qtwidgets/qcombobox.html" \l "currentIndex-prop)**() const;

QString [**currentText**](qthelp://org.qt-project.qtwidgets.550/qtwidgets/qcombobox.html#currentText-prop)() const;

int [**findData**](qthelp://org.qt-project.qtwidgets.550/qtwidgets/qcombobox.html#findData)**(const QVariant & data, int role = Qt::UserRole,**

Qt::MatchFlags flags =

static\_cast<Qt::MatchFlags> ( Qt::MatchExactly | Qt::MatchCaseSensitive )) const;

int **[findText](qthelp://org.qt-project.qtwidgets.550/qtwidgets/qcombobox.html" \l "findText)** (const QString & text, Qt::MatchFlags flags =

static\_cast<Qt::MatchFlags> ( Qt::MatchExactly | Qt::MatchCaseSensitive )) const;

void [**insertItem**](qthelp://org.qt-project.qtwidgets.550/qtwidgets/qcombobox.html#insertItem)**(int index, const QString & text, const QVariant & userData = QVariant());**

void [**setCurrentIndex**](qthelp://org.qt-project.qtwidgets.550/qtwidgets/qcombobox.html#currentIndex-prop)**(int index);**

void **[setCurrentText](qthelp://org.qt-project.qtwidgets.550/qtwidgets/qcombobox.html" \l "currentText-prop)(const QString & text);**

1. Обратите внимание на то, что в момент появления диалога элементы интерфейса должны отражать **текущие параметры рисования**! =>
   * В конструктор диалога можно передать текущие значения
   * в теле конструктора следует предусмотреть инициализацию интерфейсных элементов согласно полученным параметрам. Для получения компонент цвета из QColor –

### void QColor::getRgb(int \**r*, int \**g*, int \**b*,int \**a* = 0) const

1. Так как, вызвав диалог, пользователь может завершить его, как нажав “OK”, так и “Cancel”, мы не можем напрямую переносить изменения текущих установок в сцену).

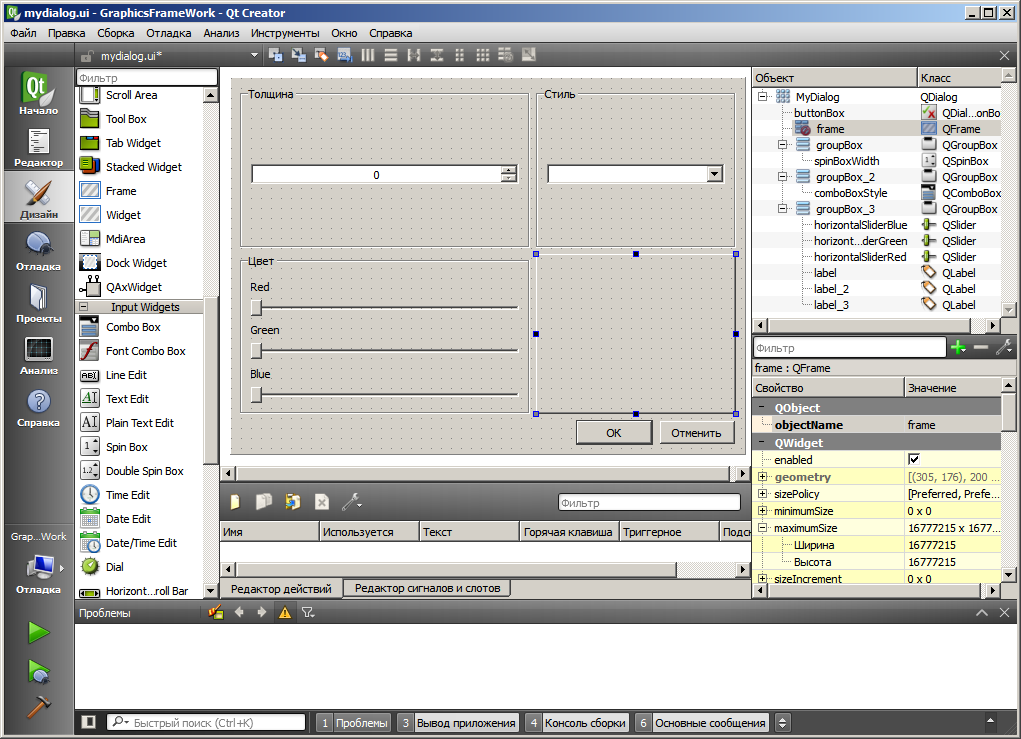
Принять новые значения следует только в том случае, если пользователь нажал OK, например, в методе getData():

1. Реализовать вызов этого диалога. Инициировать вызов диалога можно посредством меню, а слот должен быть в классе сцены:

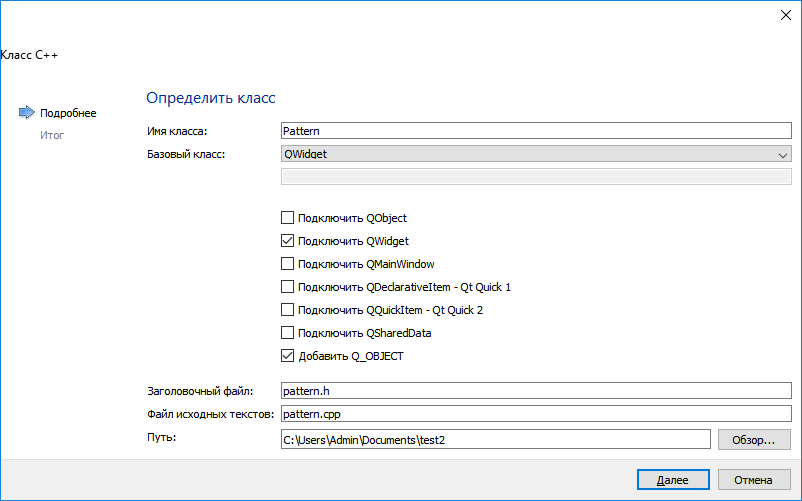
void MyScene::slotUserDialog();

### Пользовательский виджет для демонстрации образца

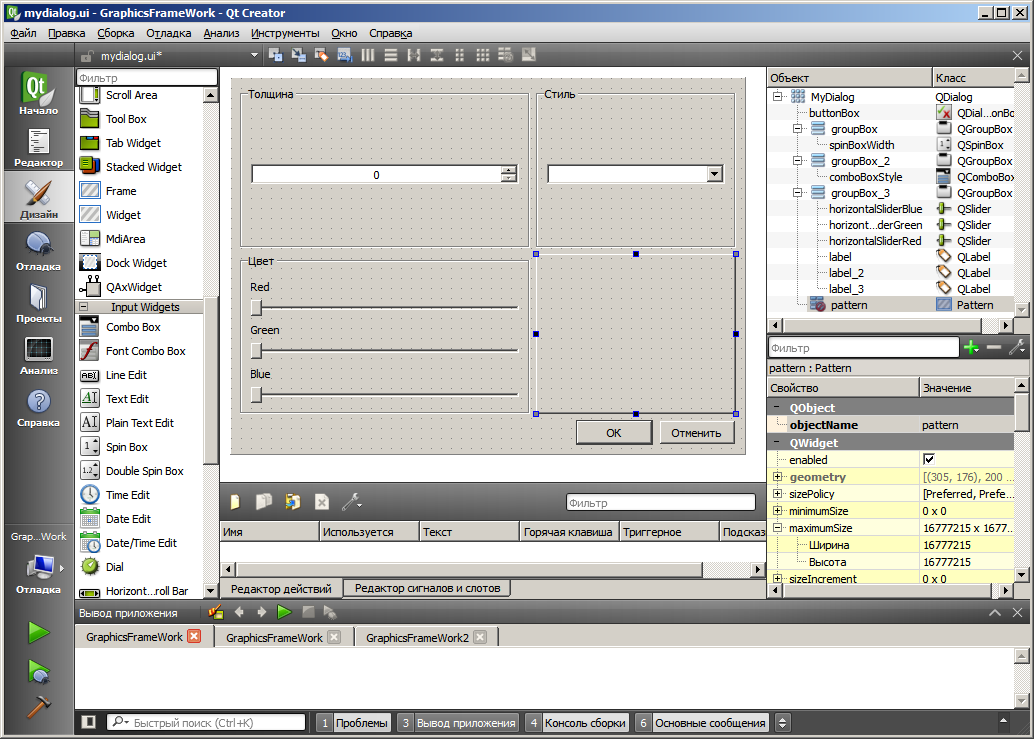
Так как пока редактирование атрибутов «рисования» происходит вслепую, неплохо было бы предоставить пользователю возможность (не завершая диалога) увидеть – чем ему придется рисовать:



* На форме диалога разместите элемент Frame (или Widget)/
* Создайте класс, например, Pattern, который является наследником QFrame ( или QWidget)
* Если хотите создавать наследника QFrame , то все равно следует при создании указать в качестве базового класса **QWidget,**  а уже потом в заголовочном файле и файле реализации внести корректировки (иначе QtCreator создает класс не совсем корректно). См. замечание в части 3.



* в дизайнере преобразуйте тип QFrame в Pattern



* В классе заведите переменную типа QPen (перо должно соответствовать текущим установкам диалога).
* Перегрузите виртуальный обработчик события paintEvent(), в котором просто отрисовывайте линию текущим пером
* При изменении атрибутов пера в диалоге «образец» должен перерисовываться! =>

в классе Pattern создайте слот, например:   
void Pattern::slotChangePen(QPen newPen),

в котором будет сохраняться в переменной класса новое значение  
и вызываться перерисовка виджета

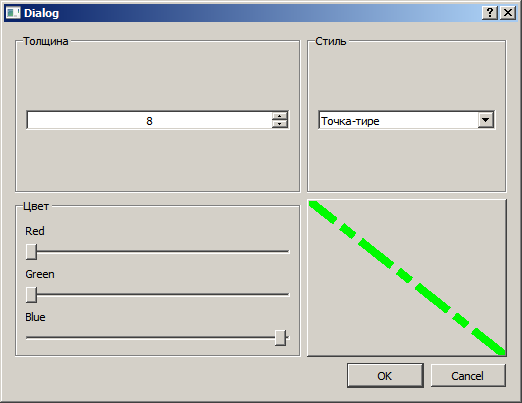
### Модификация класса MyDialog

При любых изменениях нужно перерисовывать образец =>

* в классе MyDialog создаем сигнал, например:   
  void signalChangePattern(QPen);
* в классе MyDialog создать вспомогательный слот, например,   
  void slotSomebodyChanged()  
  в котором
  + из элементов управления получаем текущие значения, из которых формируем перо. Можно воспользоваться методом getData() для получения текущих значений, введенных пользователем.
  + и эмитируем сигнал signalChangePattern(QPen) образцу, чтобы он принял изменения и перерисовался (то есть сработал слот

void Pattern :: slotChangePen(QPen newPen))

* сигнал от каждого элемента управления соединяем с вспомогательным слотом
* подсказка: посредством сигнала signalChangePattern можно также устанавливать текущие значения при вызове диалога



## Дополнительные задания:

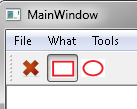
## Рисование эллипсов (кружков)

Рисование эллипсов (кружков) добавьте самостоятельно

Замечание:

* В классе сцены необходимо ввести признак того, какой тип фигуры предполагается рисовать.
* Его можно задавать с помощью перечисления enum ShapeType {RECTANGLE, ELLIPSE};

### Настройте меню и tool bar

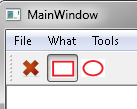
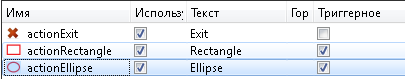
Выбор признака – «какую фигуру рисуем»:

* Посредством меню
* Посредством кнопок на toolbar

* Создайте заготовки для слотов выбора типа рисуемого примитива в myScene.
* И свяжите сигналы triggered() соответствующих действий-actions с созданными слотами.

**QAction Group**

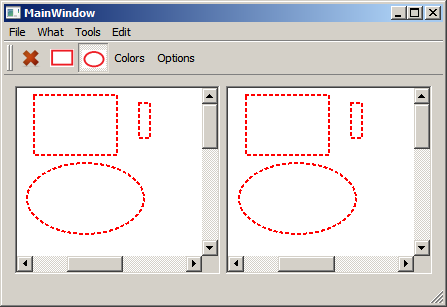
Для того, чтобы на форме было видно какой тип фигуры выбран в текущий момент, удобно объединить действия – action в группу. Тогда при выборе прямоугольника вы увидите:

* Установите для action состояние – триггерное
* В класс mainWindow поместите QActionGroup \* pactionGroup;
* В конструкторе mainWindow после вызова ui->setupUi (this) **динамически** создайте группу QActionGroup и добавьте туда ui->actionRectangle и ui->actionEllipse. При создании можно указать parent\* .
* Выберите какое-либо действие с помощью setChecked () для того, чтобы установить значение по-умолчанию.

Например, ui->actionRectangle->setChecked(true);

Исходя из Ваших дефолтовых значений для пера (кисти), результат может быть:



Проверьте функциональность. Если Вы все сделали корректно, то примитивы можно:

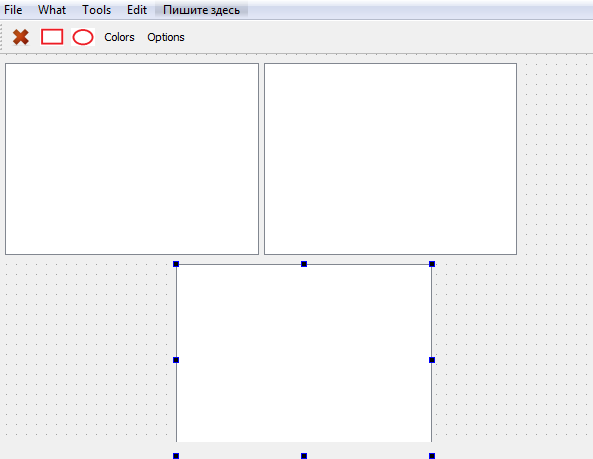
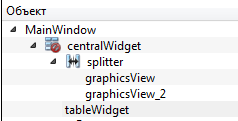
* Выделять (**левым** кликом мыши)
* И перетаскивать (при нажатой **левой** клавиши)

## Сохранение параметров фигуры в QTableWidget

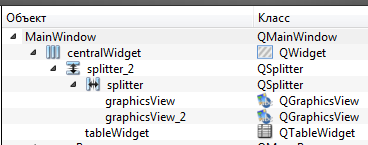
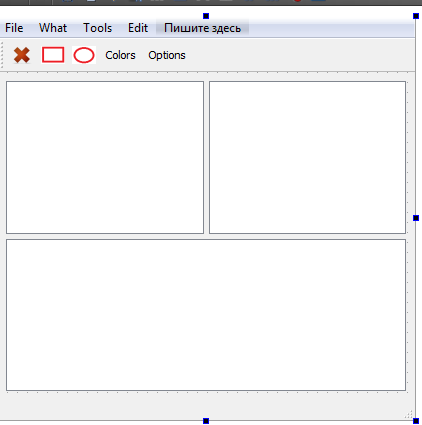
Будем сохранять параметры фигур (прямоугольников, эллипсов) в табличном виджете.

### Модификация главного окна. Добавление Table Widget

1. прежде всего удалите горизонтальный компоновщик, который подстраивал компоновку двух Graphics View со сплиттером к размерам виджета. Для этого выделите виджет окна и выберите «Удалить компоновщик»
2. Выберите на панели инструментов Table Widget и разместите его под компоновкой со сплиттером

* Выделите сплиттер и Table Widget (при нажатом CTRL) и в контекстном меню выберите «Скомпоновать по вертикали с разделителем»
* Чтобы полученная компоновка «подстраивалась» под размеры центрального виджета, сделать активным central widget и выбрать горизонтальную компоновку



### Формирование размеров Table Widget

Требуется сохранять следующие параметры фигуры:

* Тип (прямоугольник, эллипс)
* Геометрические параметры
* Стиль пера
* Толщина пера
* Цвет пера

### Задайте количество столбцов равным 5.

### Это можно сделать в коде конструктора QMainWindow . Для этого используется метод:

void QTableWidget::**setColumnCount**(int *columns*);

* Задать количество столбцов можно и в дизайнере

### Формирование заголовков Table Widget

Задайте заголовки столбцов таблицы. Это можно сделать в коде конструктора QMainWindow или с помощью дизайнера на форме.

Если заголовки столбцов задаете вручную в конструкторе окна, то можно:

* сформировать список строк QStringList и заполнить его
* использовать метод : void QTableWidget::**setHorizontalHeaderLabels**(const [QStringList](qthelp://org.qt-project.qtwidgets.571/qtcore/qstringlist.html) &*labels*);
* настроить режим растяжения для секций заголовка:

//получаем указатель на представление горизонтальных заголовков –

QHeaderView\* pheader =ui->tableWidget->horizontalHeader();

//устанавливаем режим растяжения - в исходякачестве параметра - QHeaderView::Stretch

pheader->setSectionResizeMode(QHeaderView::Stretch);

### Занесение данных о фигуре в таблицу

При добавлении нарисованной фигуры на сцену необходимо сообщить табличному виджету параметры этой фигуры. Для этого:

* в классе MyScene создаем сигнал, например:   
  void signalNewShape(ShapeType , QRectF,QColor,uint,Qt::PenStyle);
* в классе MainWindow создать вспомогательный слот, например,

void slotAddItem(ShapeType shapeType,QRectF rect,

QColor color,uint penWidth,Qt::PenStyle penStyle);

* В слоте необходимо:

1. Добавить строчку в таблицу с помощью слота void QTableWidget::**insertRow**(int*row*);

Номер текущей строки определяется с помощью int **rowCount**() const;

1. Cоздать объект QTableWidgetItem для каждого столбца текущей строки таблицы
2. Запретить редактирование элемента, т.к. в данной задаче это не имеет смысла
3. Поместить созданный объект в табличный виджет :

void QTableWidget::**setItem**(int*row*, int*column*, [QTableWidgetItem](qthelp://org.qt-project.qtwidgets.551/qtwidgets/qtablewidgetitem.html) \**item*);

Замечания:

1. Для настройки свойств табличного элемента QTableWidgetItem можно использовать методы:

### void QTableWidgetItem::setIcon(const [QIcon](qthelp://org.qt-project.qtwidgets.551/qtgui/qicon.html) &*icon*);

### void QTableWidgetItem::setText(const [QString](qthelp://org.qt-project.qtwidgets.551/qtcore/qstring.html) &*text*);

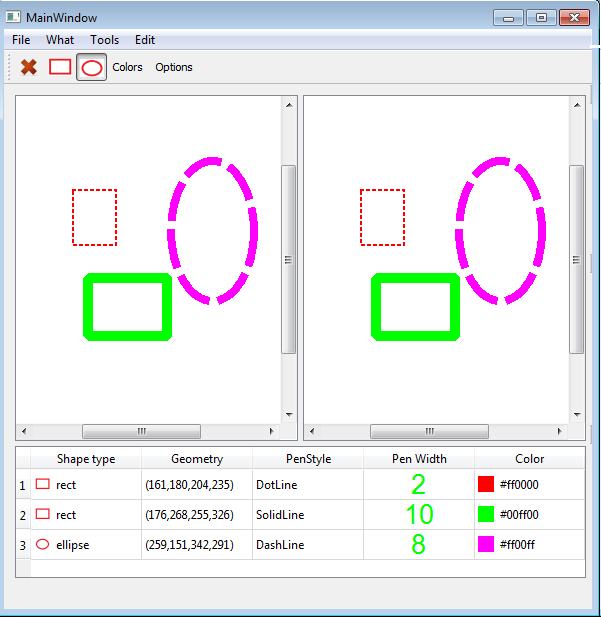
### void QTableWidgetItem::setData(int*role*, const [QVariant](qthelp://org.qt-project.qtwidgets.551/qtcore/qvariant.html) &*value*);

### void QTableWidgetItem::setFlags([Qt::ItemFlags](qthelp://org.qt-project.qtwidgets.551/qtcore/qt.html" \l "ItemFlag-enum)*flags*);

1. Можно поэкспериментировать с заданием различных значений в зависимости от роли, в которой используется элемент. Некоторые роли:

|  |  |  |  |
| --- | --- | --- | --- |
| **Constant** | **Value** | | **Description** |
| Qt::DisplayRole | | 0 | The key data to be rendered in the form of text. ([QString](qthelp://org.qt-project.qtwidgets.551/qtcore/qstring.html)) |
| Qt::DecorationRole | | 1 | The data to be rendered as a decoration in the form of an icon. ([QColor](qthelp://org.qt-project.qtwidgets.551/qtgui/qcolor.html), [QIcon](qthelp://org.qt-project.qtwidgets.551/qtgui/qicon.html) or [QPixmap](qthelp://org.qt-project.qtwidgets.551/qtgui/qpixmap.html)) |
| Qt::EditRole | | 2 | The data in a form suitable for editing in an editor. ([QString](qthelp://org.qt-project.qtwidgets.551/qtcore/qstring.html)) |
| Qt::FontRole | | 6 | The font used for items rendered with the default delegate. ([QFont](qthelp://org.qt-project.qtwidgets.551/qtgui/qfont.html)) |
| Qt::TextAlignmentRole | | 7 | The alignment of the text for items rendered with the default delegate. ([Qt::AlignmentFlag](qthelp://org.qt-project.qtwidgets.551/qtcore/qt.html" \l "AlignmentFlag-enum)) |
| Qt::BackgroundRole | | 8 | The background brush used for items rendered with the default delegate. ([QBrush](qthelp://org.qt-project.qtwidgets.551/qtgui/qbrush.html)) |
| Qt::ForegroundRole | | 9 | The foreground brush (text color, typically) used for items rendered with the default delegate. ([QBrush](qthelp://org.qt-project.qtwidgets.551/qtgui/qbrush.html)) |

В результате должно получиться следующее:



## Экспериментируем с трансформациями примитивов. Dockwidgets

Для управления трансформациями удобно пользоваться стыкуемыми (dock) виджетами. В дизайнере выберите в панели инструментов DockWidget, перетащите его на форму и наполните содержимым:

Осталось связать сигнал кнопки «Повернуть» со слотом mainWindow. Это удобнее сделать с помощью контекстного меню «Перейти к слоту» или посредством лямбда-функции

В слоте вызвать для первого представления метод

void QGraphicsView::rotate(qreal angle)

Аналогично можно управлять масштабом и «скосом»

void QGraphicsView::scale(qreal sx,qreal sy)

void QGraphicsView::shear(qreal sh,qreal sv)

Обратите внимание, что трансформация выполняется над объектом первого представления. Второе представление не изменяется, т.к. данные сцены не изменяются.

