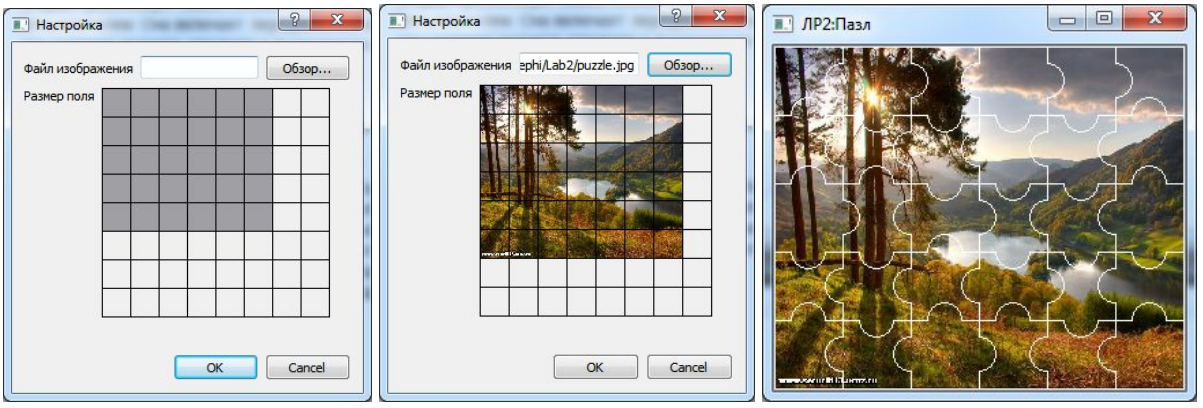
Лабораторная работа №2. (лекции 5-6)

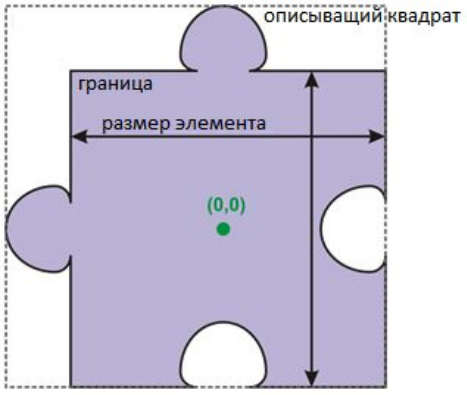
Цель: научиться создавать собственные виджеты и использовать графическую систему Qt. `Qt Meta-object system, Qt resource system, Qt Signal-Slot mechanism, Qt Creator. QApplication, QGraphicsPathItem, QPainterPath, QPainter, QGraphicsScene,QGraphicsView, QGraphicsSceneMouseEvent, QFileDialog, QPixmap,QMouseEvent`

Продолжительность: 4 часа

Задача: разработать приложение-головоломку типа пазл (puzzle). Игровое поле построено на основе графической сцены с использованием специально разработанных виджетов для элементов головоломки и диалога настройки сложности игры. Данная программа сложнее, чем примеры на лекциях, поэтому лучше потратить время на изучения каждого этапа создания приложения и прочитать документацию, чтобы понимать что делается и почему. Выполнение этой работы подразумевает, что вы уже свободно владеете Qt Creator, в частности умеете создавать классы и файлы. Инструкция больше уделяет внимания задаче, которую нужно сделать, а не тому как ее выполнить.

Проект состоит из двух частей. Первая посвящена созданию диалога настройки головоломки (включая создание специального виджета для задания количества элементов игрового поля). Вторая - реализация самой головоломки: логики, отображения и взаимодействия с пользователем. Она включает создание пользовательского виджета для отображения элемента пазла, передвижение элементов по игровому полю и возможность элементов “прилипать” друг к другу.





Требования к приложению.

1. При запуске приложения открывается диалог настройки (выбор файла с изображением и размер игрового поля)
2. Виджеты диалога настройки правильно изменяют размеры при изменении размера самого диалога настройки (правильная компоновка).
3. При нажатии кнопки “Обзор…” и выборе файла с изображением оно загружается и отображается. Если пользователь выбрал недействительное изображение (или его не удалось открыть) должно показаться предупреждение после чего управление снова должно вернуться в диалог настройки. Минимально возможный размер поля 2x2, максимальный 8x8.
4. Клик мышки на элементе(ячейке) виджета предпросмотра игрового поля изменяет размер фонового изображения (подсветки если оно не задано)
5. Перетаскивание курсора мыши при нажатой левой кнопке изменяет размер фонового изображения (подсветки если оно не задано)
6. Правильно обрабатываются кнопки “ОК” и “Отмена” в диалоге настройки. “Ок” - Открывается игровое поле (начало игры с указанными настройками), “Отмена” - завершение приложения.
7. Элемент пазла представляет собой квадратный элемент с коннекторами на сторонах. Коннекторы соседних элементов совпадают.
8. При старте новой игры элементы пазла отображаются на игровом поле (их количество равно заданному в диалоге настройки и они содержат части выбранного изображения). Расположение элементов на игровом поле и конфигурация коннекторов у них произвольные (не повторяются при перезапуске приложения)
9. Элементы возможно перетаскивать с помощью мыши.
10. Если элемент расположить рядом соседним (правильным) элементом с нужной стороны, так чтобы они составляли единое изображение, то они объединятся. Элементы считаются расположенными рядом, если окажутся на расстоянии меньше 5 пикселей (правильные элементы “прилипают” друг к другу) .
11. Собранные части головоломки (объединенные элементы) перетаскиваются мышью целиком (т.е. одновременно все элементы, составляющие эту часть). В том числе и вся картинка целиком для собранного пазла.
12. Отображается диалог с поздравлением, если головоломка полностью собрана.
13. Собранная головоломка содержит все изображение, а не его часть.

Оценка

Максимальная оценка за выполнение работы - 25 баллов. Оценка выставляется по результатам проверки работоспособности программы по пунктам требований по правилу:

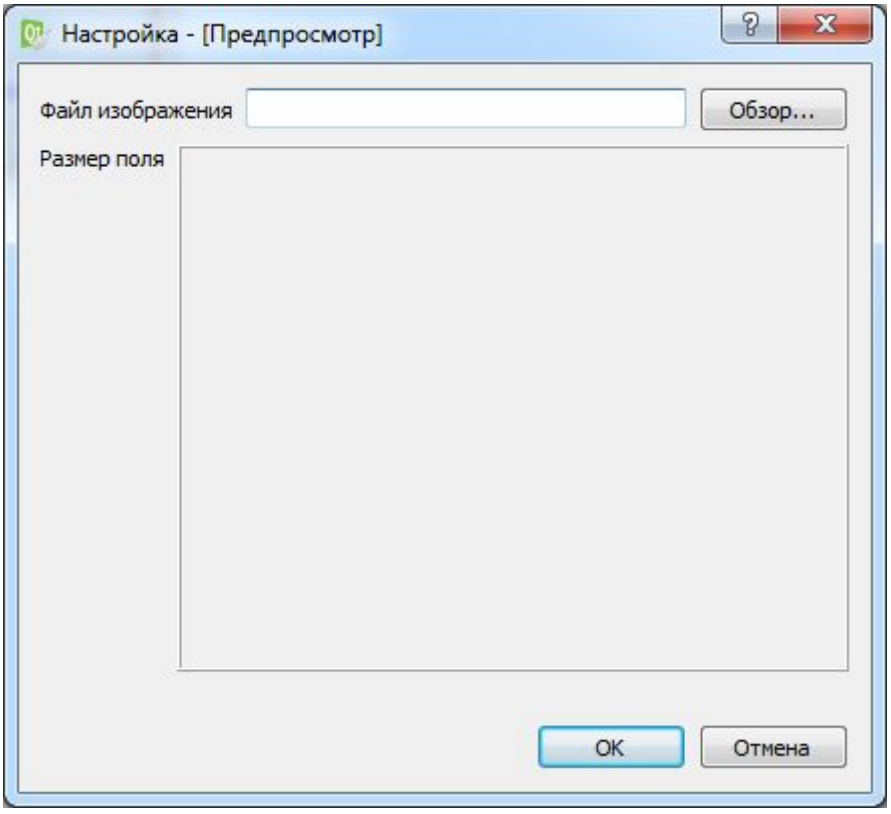
- 1 балл за пункты 1,2,3,9
- 2 балла за каждый из пунктов 4,5,6,8,12,13
- 3 балла за каждый из пунктов 7,10,11

При выполнении работы на 25 баллов самостоятельно, без использования методического материала, оценка увеличивается на **20%** (т.е. можно получить 30 баллов). Если выполнение работы с помощью методического материала займет более чем неделю, оценка уменьшается на **20%** (т.е. можно получить не более 20 баллов).

Методические указания по выполнению лабораторной работы №2.

1.Диалог настройки

Создайте новый пустой проект в Qt Creator. В проект добавьте диалог на основе класса формы `Qt Designer (Form Class)`, как показано на рисунке. Назовите класс диалога `ConfigurationDialog`.



Большая прямоугольная область в центре окна объект `QFrame`, у которого свойство `frameShape=Box`. Остальные элементы диалога: `label, line edit, dialog button box, push button`. Добавьте их и добейтесь правильного расположения с помощью компоновщиков.

После окончания проектирования пользовательского интерфейса, первое, что нужно сделать - активизировать кнопку “Обзор”. Она нужна для для показа диалога выбора файла с изображением для пазла. Добавьте слот `on_browse_button_clicked()` в код диалога и соедините его с сигналом `clicked()` (добавьте вызов `connect()` в конструкторе диалога) .

```
connect(ui->browse_button, SIGNAL(clicked()),
                           SLOT(on_browse_button_clicked()));
```

В слоте используйте `QFileDialog::getOpenFileName()` для выбора файла с изображением. В `getOpenFileName()` используйте фильтр по типу файла. Для получения списка графических форматов (расширений графических файлов) можно вызвать `QImageReader::supportedImageFormats()` - его можно добавить в фильтр. Или просто впишите распространенные расширения графических файлов.

`QFileDialog` вернет полный путь к выбранному файлу или пустую строку если пользователь нажмет “Отмена”. Отобразите этот путь в объекте `line_edit`.

Теперь создайте новый C++ файл и добавьте в него функцию `main()`. Создайте объект `QApplication` и экземпляр вашего диалога. Вызовете метод `exec()` диалога и проверьте, что кнопка “Обзор” работает как ожидается.

2.Виджет для задания количества элементов

Следующий этап - создание специального виджета для установки уровня сложности игры (размер игрового поля, т.е. количество элементов пазла). Виджет будет заменять объект `QFrame` в диалоге настройки.

Создайте новый класс C++ на унаследованный от `QFrame`. Назовите его `PuzzleSizeWidget`. Creator сам создаст новые файлы и добавит в них код класса.

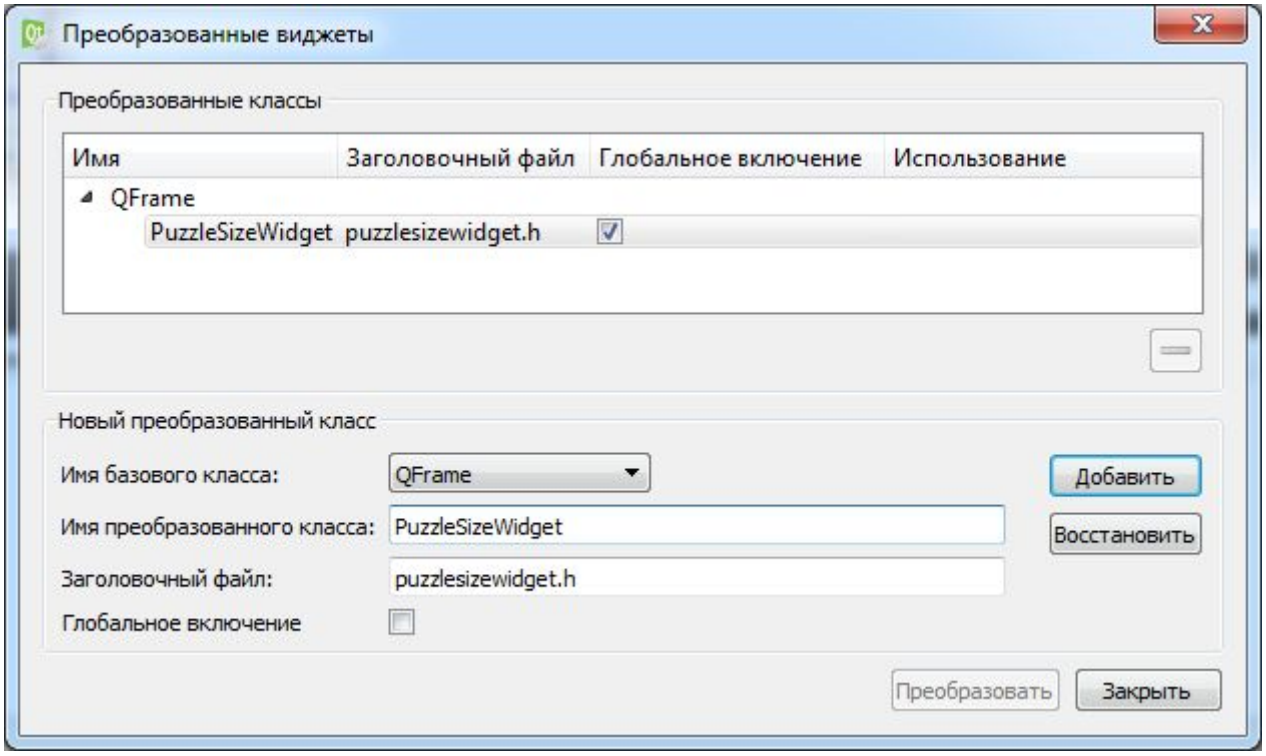
Создание заглушки виджета
Обычно, при создании пользовательских виджетов сначала добавляют все необходимые функции и выводят какую-нибудь осмысленную графическую информацию - даже если она не такая как должна быть. Это называется *заглушкой* для виджета.

В случае простых виджетов нужны только три функции: конструктор, обработчик события `paint` и `sizeHint()`. Первые два автоматически добавляются Qt Creator’ом. Вам останется только реализовать `sizeHint()`, чтобы сообщить компоновщикам информацию о размере пользовательского виджета. Пока можно вернуть произвольный размер, позже мы его исправим:

```
QSize PuzzleSizeWidget::sizeHint() const {
   return QSize(300,300);
}
```

Вставляем заглушку в диалог
Теперь, когда у нас есть заглушка для виджета настройки размера игрового поля, можно поместить ее в диалог настройки. “Проапгейдим” объект `QFrame` до нашего пользовательского виджета. Можно подменять существующие объекты на любые, которые являются его потомками. Поэтому мы унаследовали наш виджет от `QFrame`.

Откройте форму в дизайнере и в контекстном меню рамки (объект `QFrame`) выберите пункт “Преобразовать в ...” (Promote to...) и заполните поля как показано на рисунке



Нажмите “Добавить” (Add) и затем “Преобразовать” (Promote). Мы заменили виджет, но в дизайнере этого не видно, там отображается рамка `QFrame`. Qt подставит нужный класс виджета во время компиляции. Соберите и запустите проект чтобы увидеть наш виджет (пока заглушку) для настройки размера игрового поля.

Разработка программного интерфейса.
Задача виджета - позволить пользователю выбрать размер игрового поля, т.е. количество элементов пазла по каждому направлению. Вызов его метода `value` должен возвращать `QSize`, чтобы передать оба значения. Сначала определим геттер, сеттер и переменную для для хранения текущего значения свойства. При изменении значения свойства в сеттере, следует вызвать `update()` для перерисовки виджета.

```
QSize PuzzleSizeWidget::value() const {
    return size;
}

void PuzzleSizeWidget::setValue(const QSize &s) {
    if(size == s)
        return;
    size = s;
    update();
}
```

Полезной практикой является посылать сигнал при изменении значения. Объявите три сигнала в классе `PuzzleSizeWidget`:

```
    void valueChanged(const QSize&);
    void horizontalValueChanged(int);
    void verticalValueChanged(int);
```

Добавьте код для отправки сигнала в метод `setValue`. Не забудьте, что сигнал надо посылать, только если значение изменилось.

```
void PuzzleSizeWidget::setValue(const QSize &s) {
    if(size == s)
        return;
    QSize old = size;
    size = s;
    emit valueChanged(s);
    if(old.width() != s.width())
        emit horizontalValueChanged(s.width());
    if(old.height() != s.height())
        emit verticalValueChanged(s.height());
    update();
}
```

Ну и наконец добавьте свойство в систему свойств Qt:

```
    Q_PROPERTY(QSize value READ value WRITE setValue)
```

Почти всегда стоит делать сеттеры свойств слотами. Объявите `setValue` как публичный слот. (Можно использовать рефакторинг и позволить QtCreator добавить большинство кода самому. Для этого сначала объявляется свойство, а потом в контекстном меню выбрать “Рефакторинг - добавить отсутствующие члены Q_PROPERTY...”)

Виджету также нужны дополнительные свойства. Пользователь должен контролировать пределы изменения - минимальное и максимальное значение частей на которые можно разбить изображение по каждому направлению.

Добавьте в класс `PuzzleSizeWidget` свойства `minimum` и `maximum` типа `QSize` аналогично тому как вы сделали для `value`. Измените слот `setValue` таким образом, чтобы гарантировать, что значение `size` всегда находится в допустимых пределах, а сеттеры для `minimum` и `maximum` так, чтобы гарантировать, что текущее значение `size` не нарушает новые границы.

Добавим еще одно, чисто косметическое свойство `pixmap` типа `QPixmap`. Оно будет использоваться для предпросмотра того как будет выглядеть пазл при изменении настроек. Не забудьте добавить код загрузки в него изображения из выбранного файла.

Не забудьте установить значение свойств по умолчанию в конструкторе виджета.

Отрисовка виджета

Функция отрисовки виджета - обычно наиболее сложная часть пользовательских виджетов. Лучше всего сначала разработать и по шагам расписать алгоритм того как будет реализовываться отрисовка. Хорошим подходом будет реализовать затем каждый шаг как

отдельную функцию. Метод `paintEvent()` класса `PuzzleSizeWidget` тогда будет выглядеть приблизительно так:

```
void PuzzleSizeWidget::paintEvent(QPaintEvent * event){
   QPainter painter(this);
   renderValue(&painter);
   renderGrid(&painter);
}
```

Сначала создается экземпляр `QPainter` для рисования виджета. А потом вызываются две функции куда он передается как аргумент. В первой функции, `renderValue`, рисуется подсвеченная область отображающая количество выбранных частей (основанная на свойстве `value`). Если задано изображение `pixmap`, то оно будет использоваться для подсветки. Вторая функция, `renderGrid`, рисует сетку максимального размера (основанная на свойстве `maximum`) поверх подсвеченной области.

Обеим этим функциям потребуется служебный метод `cellSize`, который будет возвращать размер одного элемента, вычисленный на основе `maximum` и текущего размера самого виджета. Начнем с его реализации:

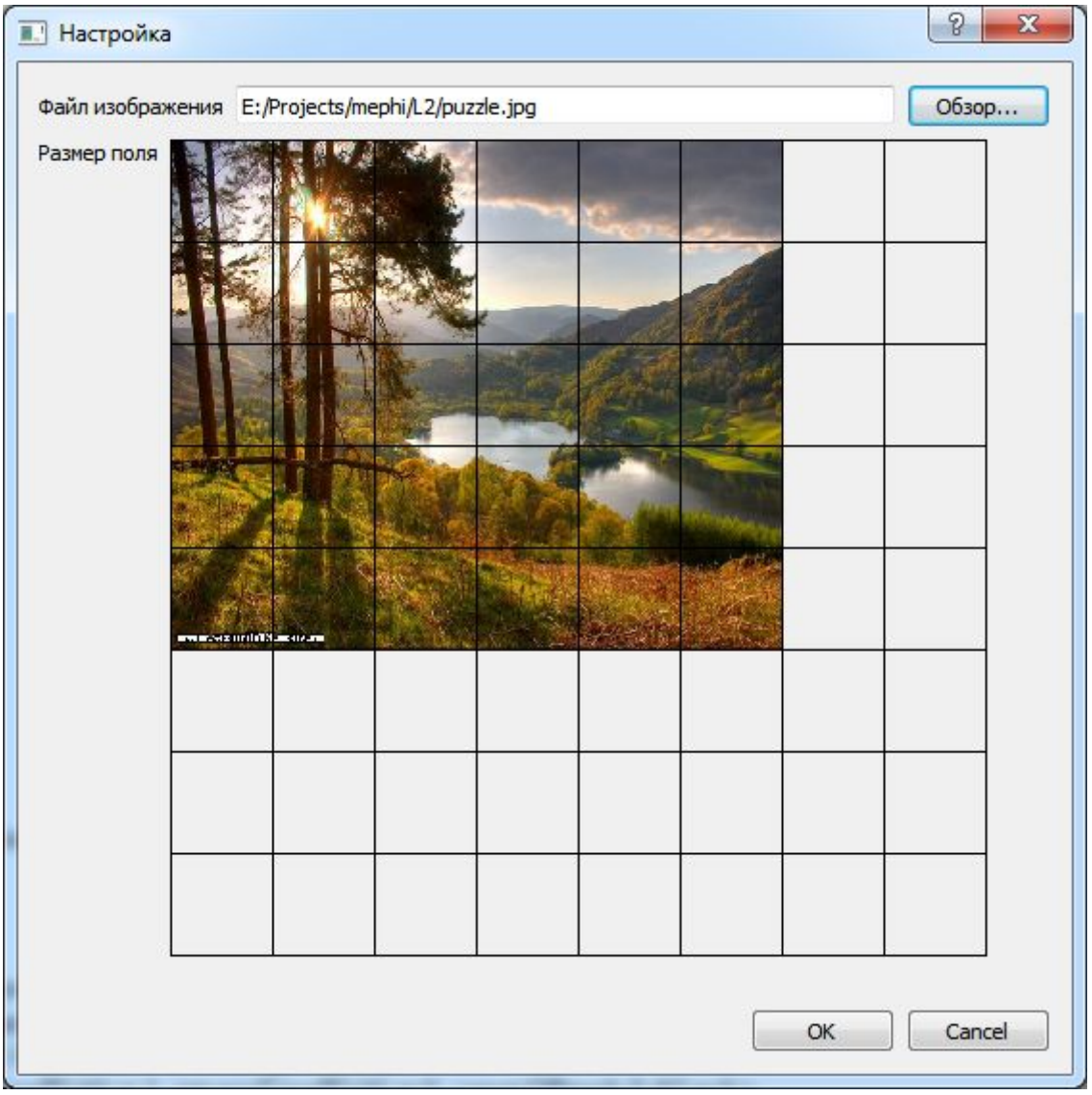
```
QSize PuzzleSizeWidget::cellSize() const {
   int w = width();
   int h = height();
   int mw = maximum().width();
   int mh = maximum().height();
   int extent = qMin(w/mw, h/mh);
   return QSize(extent,extent).
          expandedTo(QApplication::globalStrut()).expandedTo(QSize(4,4));
}
```

Сначала вычисляются размеры элемента по каждому направлению и возвращается минимальный из них как размер одного элемента. Это гарантирует, что элемент будет квадратным и поместится в виджет. Перед тем как вернуть вычисленное значение, проверяется, что размер элемента не меньше чем 4x4 пикселя или не меньше минимального размера элемента интерфейса с которым может взаимодействовать пользователь (т.н. `globalStrut`)

Зная размер одного элемента можно его нарисовать - реализуем `renderValue`. Сначала следует проверить содержится ли действительная картинка в свойстве `pixmap`. Если да, то надо ее отмасштабировать, чтобы она покрывала количество элементов заданное пользователем в свойстве `value` и сгенерировать картинку для отображения. Если же картинки нет, то нарисовать залитый прямоугольник (указать кисть) вместо нее.

Реализация `renderGrid()` достаточно проста. Используете два вложенных цикла, чтобы пробежать все элемента по горизонтали и вертикали (используете `maximum` для вычисления количество итераций в цикле). Для каждого элемента используйте `QPainter::drawRect()` для отображения границы. Не забудьте сначала очистить кисть сбросив ее в `Qt::NoBrush`.

После сборки и запуска приложения вы должны получить что-то похожее на картинку ниже. Убедитесь ,что вы задали файл изображения и видите его в предпросмотре.



Обработка сообщений мыши

Заключительная часть программирования пользовательского виджета настройки игрового поля - обработка сообщений мыши. Пользователь должен задавать размер игрового поля или кликая на ячейку или перетаскивая мышь по ним при нажатой левой кнопке. Чтобы это сделать нужно переопределить и реализовать методы `mousePressEvent()` и `mouseMoveEvent()`.

Но перед этим добавим служебный метод `cellAt()`. Передадим в него объект `QPoint` в качестве аргумента. Эта точка будет представлять координаты в системе координат виджета. Метод вернет другой объект `QPoint` с координатами элемента(ячейки) пазла в данной точке. Используйте в качестве образца метод `cellSize()`.

Теперь добавьте код в функции `mousePressEvent()` и `mouseMoveEvent()`. Вы можете получить координаты мыши из события используя метод `pos()` аргумента события. Теперь остается только проверить координаты элемента(ячейки) и вызвать метод `setValue()`.

Завершение диалога настройки

Остается сделать еще несколько мелких доработок, чтобы наш диалог настройки был полностью готов. После того как пользователь сделает свой выбор нужно вернуть из диалога заданные настройки: имя файла с изображением и размер игрового поля.

Это можно реализовать если добавить методы-геттеры в класс диалога. Эти геттеры просто опрашивают виджеты диалога настройки о текущих значениях. Для этого добавьте методы `imageFilePath()` и `puzzleSize()` в класс диалога и реализуйте в них запрос значений у соответствующих виджетов.

Контрольные вопросы:

1. виджеты диалога настройки правильно изменяют размеры при изменении размера самого диалога настройки (правильная компоновка)
2. При нажатии кнопки “Обзор…” и выборе файла с изображением он загружается и отображается в виджете предпросмотра.
3. Клик мышки на элементе(ячейке) виджета предпросмотра изменяет размер фонового изображения (подсветки если оно не задано)
4. Перетаскивание курсора мыши при нажатой левой кнопке изменяет размер фонового изображения (подсветки если оно не задано)
5. При изменении настроек диалог перерисовывается.
6. Правильно обрабатываются кнопки “ОК” и “Отмена” в диалоге настройки (слоты `accept()` и `reject()` диалога соединены с сигналами кнопок)

2.Игра

В этой части лабораторной работы мы будем использовать графическую систему для реализации самой игры. Две вещи нужно сделать на этом этапе: элемент пазла и механизм их соединения с проверкой что сборка пазла завершена.

Подготовка холста

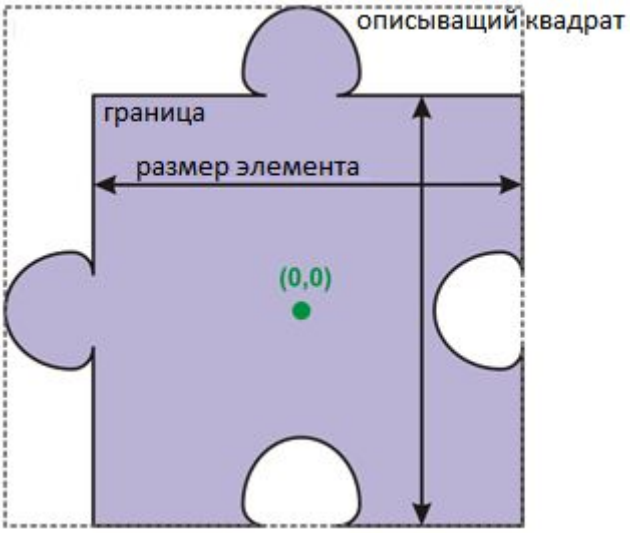
Чтобы разработать класс элемента пазла, создадим игровое поле, чтобы можно было его тестировать. В функции `main` добавим создание двух объектов: класса `QGraphicsView` и класса `QGraphicsScene` и соединим их вызовом метода `setScene()`. Помните, что нужно добавить вызов для отображения `QGraphicsView` и заменить вызов `exec` диалога вызовом `QApplication::exec()`, т.к. пока нам не понадобится диалог настройки.

Запустив приложение вы увидите окно с пустым холстом, который и будет игровым полем.

Элемент пазла

Головоломки типа пазл содержат элементы, которые нужно сложить в определенном порядке. Можно просто использовать готовый класс, такой как `QGraphicsRectItem`, для создания прямоугольного элемента, но лучше запрограммируем собственный элемент с более привычной формой, чем просто прямоугольник. Это добавит ощущение того что перед нами настоящий пазл.

Рисунок ниже показывает эскиз элемента. Начало координат расположим в центре квадрата элемента.



Граница элемента содержит выступы для формирования коннекторов. Это значит что описывающий прямоугольник(квадрат) должен также включать в себя и коннекторы. Конечно, т.к. разные элементы должны иметь разные коннекторы (наружу или внутрь), граница и описывающий прямоугольник будут меняться от элемента к элементу. Поэтому важно сохранить начало координат в центре главного тела элемента (оно всегда квадратное и одного размера).

Реализуем класс элемента пазла. Назовем его `PuzzlePiece` и унаследуем от `QGraphicsPathItem`. Для простоты примем размер элемента не зависящим от размера картинки пазла. Т.е.возможно придется масштабировать размер изображения для подгонки к размеру элементов. Допустим каждый элемент будет размером 50x50 пикселей.

Взглянем еще раз на эскиз. Можно увидеть,что элементы могут иметь до 4 коннекторов. Каждый из которых может быть или выступающим(направлен наружу) или впадающим(направлен внутрь). Или же сторона не имеет коннектора (для крайних элементов).

Добавим в наш класс перечисление, описывающее возможные конфигурации коннекторов.

```
enum ConnectorPosition { None, Out, In };
```

Теперь можно описать точную форму элемента задав типы всех четырех коннекторов. Добавим конструктор в класс `PuzzlePiece`, который принимает 4 аргумента и сохраняет их в приватных переменных:

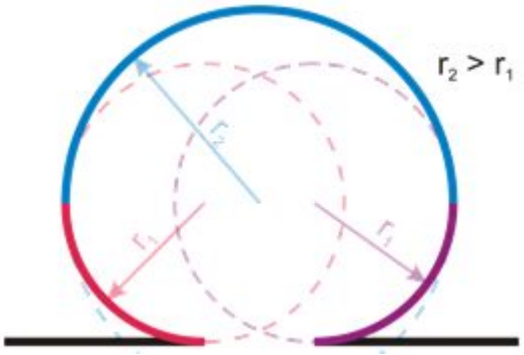
```
PuzzlePiece(ConnectorPosition north, ConnectorPosition east,
            ConnectorPosition south, ConnectorPosition west);
```

Затем конструктор должен вызвать несколько внутренних методов для построения границы элемента. Назовем его `constructShape()`.

Не будем пошагово описывать построения границы, но несколько советов все же дам. Если вас пугает задача построения границы с закругленными коннекторами то сделайте что-то попроще, например трапецевидные или прямоугольные.

Основное правило построение пути границы элемента используя класс `QPainterPath` - это сначала вызвать `moveTo()` для перемещения карандаша в один из углов. Затем вызывайте

`lineTo()` для добавления секций. Помните, что начало координат находится в центре элемента. Т.к. размер элемента 50x50, то верхний левый угол будет иметь координаты (-25,-25).



На диаграмме слева показано как можно построить закругленный коннектор соединив несколько дуг. Например, к левой черной прямой линии границы, добавляется красная дуга (на четверть круга), затем добавляется голубая дуга (на пол-окружности) и наконец сиреневая дуга (на четверть круга) - зеркальное отражение красной. Дальше добавляется черная прямая линия границы.

Если вам неохота вычислять параметры отрисовки дуг, просто замените их более простой формой. В любом случае используйте конфигурацию коннектора `ConnectorPosition` для определения того нужно ли его рисовать и в каком направлении.

Когда все линии и коннекторы будут добавлены вызовите `QPainterPath::closeSubpath()`. Затем задайте построенный путь в элемент используя `QGraphicsPathItem::setPath()`. Базовый класса пути позаботится о формировании правильного описывающего прямоугольника для пути.

На этом этапе вы должны показать отображение различных типов элементов пазла на игровом поле. Следующий шаг - правильно нарисовать каждый элемент.

Чтобы это сделать нужно знать текстуру каждого элемента. Объявим методы `setPixmap()` и `pixmap()` для установки и получения изображения для элемента. Сохраните картинку как приватный член класса `PuzzlePiece` .Убедитесь что сеттер вызывает `update()` для перерисовки элемента.

Для завершения рисования каждого элемента, нужно создать функцию рисования, которая заполняет внутреннюю область пути границы элемента картинкой.

Самый простой способ сделать это - ограничить область рисования путем границы, а затем отобразить картинку в этой области. Это достигается путем обрезки. Переопределите у элемента функцию `paint()`. Используйте `QPainter::setClipPath()` для ограничения области рисования и отобразите картинку в элементе.

Теперь мы закончили реализацию класса элемента пазла.

Игровая логика.

Начните реализацию игровой логики с создания класса на базе `QGraphicsScene`. В этом классе добавьте метод `setup()` для настройки игрового поля перед началом игры в который передается размер пазла и изображения (заданные в диалоге настройки).

Первой задачей этого метода должна быть очистка игрового поля, т.к. новая игра всегда начинается с пустого поля. После этого, добавьте два вложенных цикла `for` для перебора строк и колонок игрового поля. На каждой итерации нужно создать элемент пазла - `PuzzlePiece`.

Для каждого элемента вам нужно определить конфигурации коннекторов так чтобы они подходили друг к другу. Расположение коннекторов (In, Out) должно быть случайным. Можно использовать временный массив (например вектор) типов `ConnectorPosition` представляющий коннекторы каждого элемента. Это сделает возможным создание элементов, с точным соответствием коннекторов между рядами и колонками. Можно использовать следующий алгоритм:

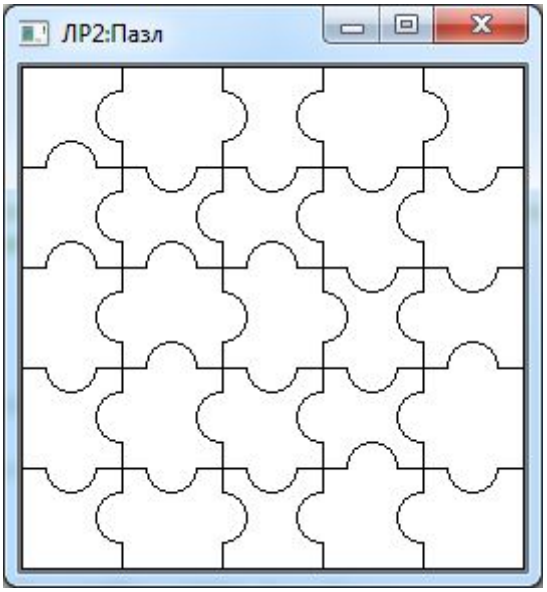
- Создайте для каждого элемента вектор коннекторов со значением None
- Если номер колонки 0, то западный коннектор элемента равен None
- Если номер колонки равен размеру поля (последний), то восточный коннектор равен None
- Если южный коннектор элемента в той же колонке предыдущего ряда (хранящийся во вспомогательном векторе) равен In, то северный коннектор текущего элемента равен Out и наоборот
- Если номер ряда равен размеру поля (последний), то южный коннектор равен None, иначе задается случайная позиция для южного коннектора, которая сохраняется в векторе текущей колонки
- Если вспомогательная переменная равна Out, то западный текущего элемента равен In и наоборот. Если None, то остается None
- В начале обработки каждого ряда, вспомогательный вектор сбрасывается в None

Попробуйте сами реализовать этот алгоритм. Помните, что нужно добавить элемент в сцену и расположить его в нужном месте. Также не забудьте установить флаг `ItemIsMovable` для всех элементов. Можете проверить свою реализацию по коду приведенному ниже.

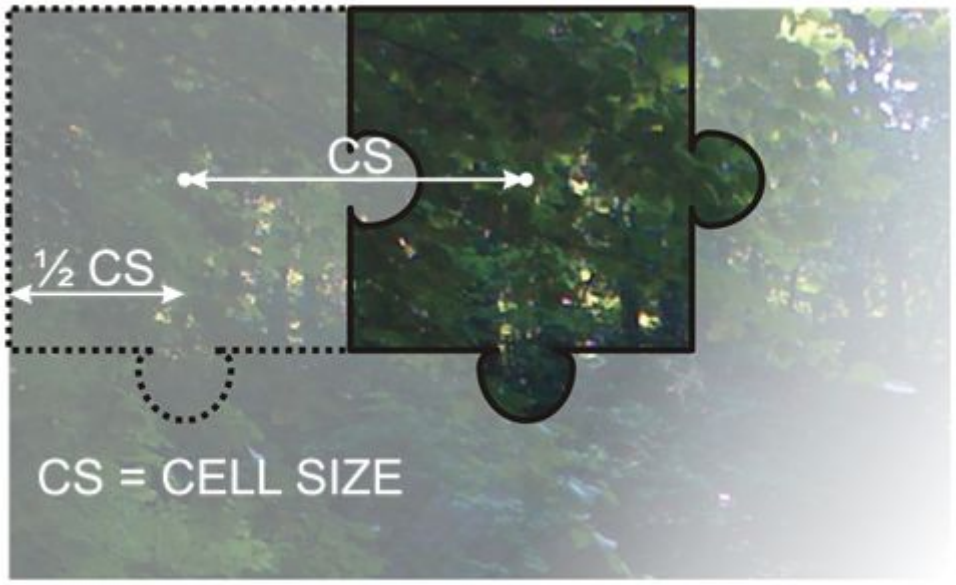
```
/*!
* получение парного коннектора
*/
PuzzlePiece::ConnectorPosition reverse(PuzzlePiece::ConnectorPosition pos)
{
      switch(pos){
      case PuzzlePiece::None: return PuzzlePiece::None;
      case PuzzlePiece::In: return PuzzlePiece::Out;
      case PuzzlePiece::Out: return PuzzlePiece::In;
      }
      return PuzzlePiece::None; // safeguard
}
// ...
PuzzlePiece::ConnectorPosition storedWest;
QVector<PuzzlePiece::ConnectorPosition> prev(size.width(),
                                             PuzzlePiece::None);
for(int row = 0; row < size.height(); ++row) {
```

```
      storedWest = PuzzlePiece::None;
      for(int col = 0; col < size.width(); ++col) {
        PuzzlePiece::ConnectorPosition curr[4]; // N, E, S, W

        curr[0] = reverse(prev[col]);
        curr[1] = qrand() % 2 ? PuzzlePiece::In : PuzzlePiece::Out;
        curr[2] = qrand() % 2 ? PuzzlePiece::In : PuzzlePiece::Out;
        curr[3] = reverse(storedWest);
        if(col==size.width()-1) curr[1] = PuzzlePiece::None;
         if(row==size.height()-1) curr[2] = PuzzlePiece::None;

         PuzzlePiece *piece = new PuzzlePiece(curr[0], curr[1],
                                              curr[2], curr[3]);
         addItem(piece); // add item to scene
         piece->setFlag(QGraphicsItem::ItemIsMovable);
         piece->setPos(col*50, row*50); // put the piece in place

         storedWest = curr[1]; // store east for next column
         prev[col] = curr[2];  // store south for next row
      }
}
```

На данном этапе задайте размер игрового поля в коде (например 5x5) и вызовете метод `setup()`, который вы только что реализовали, прямо из функции `main`. Вы должны увидеть пустые элементы расположенные в правильную решетку с совпадающими коннекторами.



Теперь нужно завершить программирование элементов - добавить код отвечающий за отображение в каждом элементе нужной части исходного изображения. Рисунок ниже поясняет как вычислить смещение каждого элемента относительно верхнего левого угла изображения. Смещение первого ряда и колонки равно половине размера элемента (нет западных и северных коннекторов), т.е. в нашем случае 25 пикселей. Смещение каждого следующего ряда и колонки равно размеру элемента - 50 пикселей.



Добавьте в метод `setup()` код установки изображения для каждого элемента. Используйте для копирования нужной части исходного изображения в новое. Можете использовать следующий фрагмент кода для получения части изображения:

```
QRect rect = piece->boundingRect().toRect(); // (0,0) в центре
const int cellSize = 50;
rect.translate(0.5*cellSize+col*cellSize, 0.5*cellSize+row*cellSize);
QPixmap px = pixmap.copy(rect);
piece->setPixmap(px);
```

Если явно задать изображение в коде и запустить программу вы должны увидеть решенную головоломку.

Учитывая все то, что вы сделали к настоящему моменту, для вас не составит труда закончить программу без каких-то специальных инструкций. Расположите элементы в произвольном порядке на игровом поле.

Оставшаяся часть задачи - “склеить” элементы расположенные на своих местах (соседние) и проверить что собран весь пазл.

Сначала добавьте в класс элемента геттер и сеттер для еще одного атрибута `coordinates` типа `QPoint` содержащего координаты куска пазла на игровом поле, представленного элементом. Например левый верхний элемент пазла имеет координаты (0,0), элемента справа от него (1,0) и т.д. Это позволит проверить находятся ли два элемента рядом и в правильном ли порядке. . После того как вы реализуете методы `coordinates()` и `setCoordinates ()` используйте их в цикле создания элементов для установки координат в каждом элементе.

Каждый элемент пазла может соединяться максимум с 4 другими элементами, по одному в каждом направлении. Создайте перечисление `Direction` в классе элемента содержащее значения `North, East, South, West.`

Добавьте массив `m_neighbors` из 4х указателей на класс элемента как приватный член класса `PuzzlePiece` в котором будем запоминать правильно собранные соседние элементы. Проинициализируете его нулями в конструкторе класса, т.к., изначально элементы не соединены ни с какими другими элементами.

Добавьте публичный метод `link` для соединения элементов. Метод `link` принимает на вход указатель на связанный элемент и значение `Direction` для указания в каком направлении он находится. Не забудьте связать элементы взаимно, в обоих направлениях. Если элемент A восточной стороной связан с элементом B, следовательно B связан западной стороной с A и т.д.

Чтобы связать элементы автоматически, когда они окажутся рядом с правильной стороны, нужен метод для поиска среди всех уже связанных соседей `m_neighbors` элемента с подходящими координатами. Алгоритм должен запускаться при отпускании кнопки мыши. Переопределите метод `mouseReleaseEvent` в классе элемента и выполните в нем алгоритм описанный ниже.

Алгоритм проверки соседа заключается в рекурсивном обходе связанных элементов. В каждом вызове элемент пытается найти нового соседа, после чего вызывается такой же метод (самого себя) у всех связанных элементов. Для предотвращения бесконечного цикла нужно условие прерывания поиска. Алгоритм хранит список уже посещенных элементов. Если оказывается что элемент уже посещался, то рекурсия прерывается.

```
void PuzzlePiece::checkNeighbors(QSet<QPoint> &checked){
    if(checked.contains(coordinate()))
      return; // условия прерывания рекурсии
    checked.insert(coordinate()); // запомнить посещенный элемент
    findneighbor(North);        // найти N соседа
    findneighbor(East);         // найти E соседа
    findneighbor(South);        // найти S соседа
    findneighbor(West);         // найти W соседа

     // рекурсивные вызовы:
    if(m_neighbors[North])
        m_neighbors[North]->checkNeighbors(checked);
    if(m_neighbors[East])
     m_neighbors[East]->checkNeighbors(checked);
    if(m_neighbors[South])
        m_neighbors[South]->checkNeighbors(checked);
    if(m_neighbors[West])
        m_neighbors[West]->checkNeighbors(checked);
       }
```

В обработчике `mouseReleaseEvent` создайте пустой набор `QSet<QPoint>` для хранения посещенных элементов и вызовите метод `checkNeighbors` для текущего элемента. Не забудьте вызвать метод `mouseReleaseEvent` базового класса, чтобы не потерять возможность двигать элементы.

Метод `checkNeighbors` использует метод `findneighbor`, который принимает на вход направление и пытается определить, находится ли там элемент который должен быть

расположен по этому направлению относительно текущего элемента в правильно собранном пазле. Реализуйте этот метод по алгоритму:

1. Если уже есть связанный элемент, то прекратить поиск
2. Элемент который вы ищите является соседним, определите его по его координатам текущего элемента.
3. Элемент который вы ищите должен иметь координаты, которые можно вычислить в соответствии с размером элемента (50) и требуемым направлением. Используйте `QGraphicsScene::itemAt()` для запроса у сцены элемента, который соприкасается с этой точкой.
4. Лучше всего оставить запас в несколько пикселей от предполагаемого точного расположения элемента.
5. Если вы найдете подходящий элемент, свяжите их и пододвиньте один из элементов, чтобы они соприкасались без зазора (можно использовать рассчитанное ранее ожидаемое положение) - получится эффект “прилипания”

Ну и чтобы головоломка вела себя так, как должен себя вести настоящий пазл осталось сделать еще одно. Связанные элементы должны двигаться как единое целое. Самый простой способ сделать это - переопределить метод `itemChange()` . Когда элемент получает событие изменения `ItemPositionHasChanged` нужно просто сдвинуть все связанные элементы.

Это, в свою очередь, вызовет событие изменения для всех связанных элементов. Они передвинут своих связанных соседей и так по цепочке. В результате весь собранный кусок пазла переместится на новое место. Не забудьте подписаться на получение сообщений об изменении установив для всех элементов флаг `ItemSendsGeomentryChanges`.

Осталось только проверка, что пазл полностью собран. Игра закончена, когда все элементы будут связаны друг с другом. Это случится, когда набор, переданный в `checkNeighbors` будет содержать все элементы пазла по окончании выполнения функции. Т.е. простой вызов `QSet::count()` даст нам ответ на этот вопрос.

Контрольные вопросы:

1. Коннекторы соседних элементов совпадают. Нет коннекторов у элементов по краям картинки
2. Элементы “прилипают”, если расположить рядом два соседних элемента
3. Собранный кусок пазла перемещается целиком (все связанные элементы)
4. Правильно определяется момент окончания игры - когда вся картинка сложена и передвигается единым куском.

3.Собираем все вместе

Чтобы завершить нашу головоломку нужно вернуть в приложение диалог настройки сложности. Верните `QDialog::exec()` для вызова диалога. Если пользователь нажмет в нем “Отмена” следует завершить приложение без отображения игрового поля.

Также нужно обработать ситуацию, если пользователь выбрал недействительное изображение (или его не удалось открыть) должно показаться предупреждение `QMessageBox` после чего управление снова должно вернуться в диалог настройки.