Lab 2 – Custom Widgets, Graphics View Canvas

Aim: This lab will take you through the steps required to build a Qt application built around custom widgets and the Graphics View framework.

Duration: 4 h

Introduction

The goal of this lab is to create an application for creating and solving jigsaw puzzles.

The application that you will create is based on the Graphics View architecture. The scene based part of the program will be supported by a widget-based dialog built around an application specific custom widget.

You will have to deal with more complex situations than those from the exercises so take your time to study each step and consult the reference manual until you are sure you know what you are doing and why.

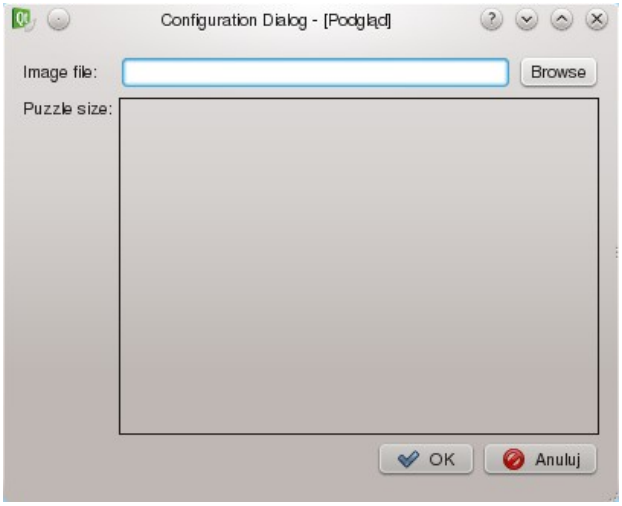
In the lab it is assumed that you feel comfortable with using Qt Creator, creating classes and files in particular. Instructions will focus more on the task at hand than on going into details on how to perform it.

The whole project consists of two parts. The first one is dedicated to creating a dialog for configuring a jigsaw puzzle. This involves creating a custom widget for controlling the number of pieces that will be used.

The second half is about implementing the game itself – the logic, presentation and user interaction. It involves moving objects on a graphics view canvas. It involves tasks such as custom items and items that can stick together.

The configuration dialog

Start by creating an empty project in Qt Creator. In that project, create a dialog such as the one shown in the picture below. Choose to base it on the 'Qt Designer Form Class' when adding it to the project. Name the class `ConfigurationDialog`.



The large rectangle in the middle of the dialog is a `QFrame` object with its `frameShape` property set to `Box`. The rest of the dialog is built from labels, a line edit, a dialog button box and an ordinary push button. Create the dialog and apply the necessary layouts.

After having designed the dialog the first thing that needs to be done is to enable the “Browse” button. It is intended to show a file dialog for choosing an image that will be the base for the jigsaw puzzle. Add a custom slot called `browseFile()` to the code of your dialog and make a signal-slot connection (by placing an appropriate `connect()` statement in the constructor of the dialog) between the button's `clicked()` signal and your custom slot.

```
connect(ui->browseButton, SIGNAL(clicked()), SLOT(browseFile()));
```

Then have the slot use `QFileDialog::getOpenFileName()` method to ask the user for an image file. `getOpenFileName()` accepts a couple of useful parameters. One of them is the file filter string that limits the list of files the user sees, to those matching the given expression. Have a look into the documentation to learn about the format of the filter string.

> To get a list of image file extensions Qt supports you can call `QImageReader::supportedImageFormats()` that will return a list of file extensions. You can use this list to assemble the filter string. Alternatively just put the common list of image extensions into the filter.

The return value of the `QFileDialog` method contains the absolute path to a file chosen by the user or an empty string if the file choosing dialog was canceled. Set the received file path as text of the line edit object.

Now create a new C++ implementation file and add the `main()` function to it. Have it create a `QApplication` object and an instance of your dialog. Then call the dialog's `exec()` method and see if

the button works as expected.

> To quickly access documentation of a method in Creator hover your mouse pointer over the name of the method in your code and press the F1 key on the keyboard.

The custom widget

The next step is to implement a custom widget that will be used to control the puzzle's difficulty level. The widget will replace the frame instance from your configuration dialog.

Create a new C++ class using `QFrame` as the base class. Call it `PuzzleSizeWidget`. Creator will put some code of the class for you in appropriate files.

Creating a stub

Usually, the first thing to do when implementing a custom widget is to provide all the necessary functions for the item, get them to compile and give some sort of sane visual output – even if the behavior and appearance is incorrect. This is called providing a *stub* for the widget.

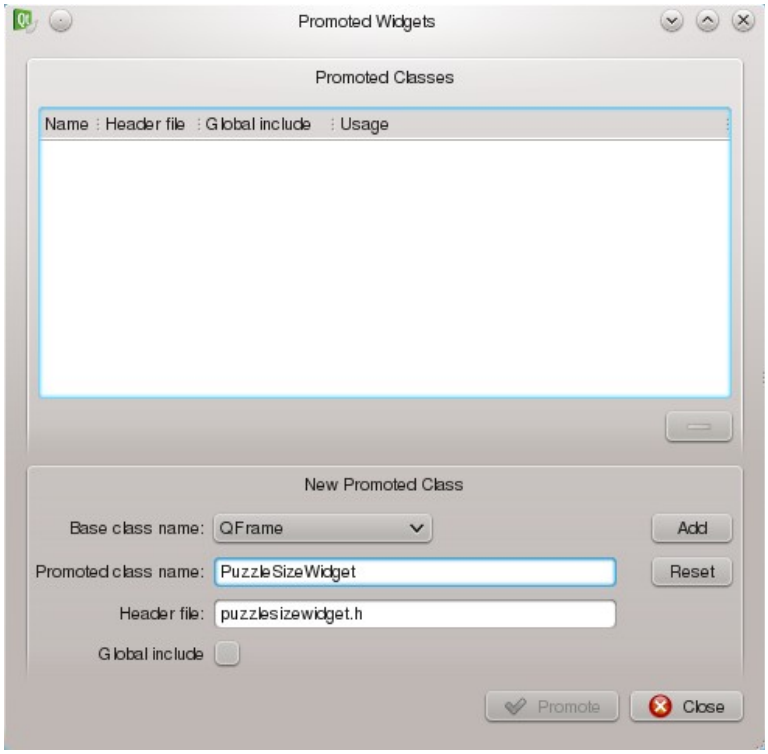
In the case of simple widgets, three things need to be implemented: a constructor, a paint event handler and the `sizeHint()` method. The first two are added automatically by Qt Creator. The only thing left for you is to implement the `sizeHint()` method to define the widget's size to Qt's layout system. You can return some arbitrary positive size for now. You will be able to correct it later on.

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

Promoting to a custom widget

Now that you have a stub for the puzzle size widget, you can place the widget in the configuration dialog instead of the frame that you have put there earlier. To do that, you will "promote" the frame widget to the custom widget class. You can promote an existing object on any form to a custom subclass of the widget in question. That is why the custom widget was derived from `QFrame`.

Open the form in Designer and right click on the frame. Choose "Promote to..." from the context menu and fill out the necessary data as shown in the image below.



In the promotion dialog, click Add and then Promote. The widget has now been promoted.

You will not see any changes in the form while using Designer – the frame is still a `QFrame`. Qt will substitute it with the right class during compilation. Build and run the project to see the effect.

> If at any time you wish to revert a promotion, you can choose “Demote to QFrame” from the context menu of the promoted frame object. You can also replace a widget with an other standard widget – you do that using the “Morph Into” submenu available in the context menu in Designer.

Designing the API

The widget's job will be to let the user choose the number of pieces in both the horizontal and vertical direction. For that purpose, a number of properties are need.

The most important property will hold the current number of pieces in either direction. Call it `value` and make its type `QSize` so that a single property can represent both directions. First define the getter and setter methods and a private class field storing the current value of the property. If the property value influences the way the widget is painted you should call `update()` in the setter to trigger a repaint of the widget.

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

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

It is usually a good practice to emit a signal whenever a property value changes. Declare the

following three signals in your class.

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

Now, add the code emitting those signals to the setValue() method. Remember to only emit a signal if the value really has changed.

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

Finally, declare the property using the `Q_PROPERTY` macro to expose it to Qt's property system. You do that by adding the following line in the class declaration.

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

You can add more arguments to the `Q_PROPERTY` macro. For example you can tell Qt which signal is responsible for informing about changes to the property by appending a `NOTIFY signalName` section to the macro substituting `signalName` part with the real name of the signal.

It is almost always a good idea to declare the setter functions for properties as slots. In this case, declare setValue() as a public slot.

It's a good approach to have a slot matching each signal emitted by the class for all write enabled properties, at least for the most popular types. This makes the class more versatile and reusable. In this current situation you can implement slots for setting horizontal and vertical values separately.

The widget also needs some additional properties. The user must be able to control the range of the valid values – the minimum and maximum number of tiles the image can be divided into in each direction.

Repeat the process described for the `value` property to add the `minimum` and `maximum` properties of type `QSize`. Modify the `setValue()` slot to make sure the value always is within the valid range and implement setters of the new properties in such a way that they too make sure the value is constrained to the new range.

You can use the `qBound()` macro to check whether a value is within given range. It

accepts three arguments – the minimum, current and maximum values and returns the current value bound to the range specified by the other two arguments.

We will provide one more widget property. This time, it is purely cosmetic. Implement a `QPixmap` property called `pixmap`. This pixmap will be used give a live preview of what the puzzle is going to look like.

Remember to provide sane default values for all properties in the widget's constructor.

Painting the widget

The most complicated part of creating a custom widget is usually the painting routine. It is best to first come up with a concept of how to implement the painting in a step by step fashion. A good approach to this is to make each step a separate method. The `paintEvent()` method of the puzzle size widget will essentially look like this:

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

First a `QPainter` instance is created for drawing on the widget. Then two calls are made, passing a pointer as an argument to the created painter. The first call, renderValue, draws a highlighted area showing the number of pieces selected (based on the `value` property). If a pixmap was set, it will be used to create the highlight effect. The second call draws a grid of possible pieces (based on the `maximum` property) on top of the highlighted area.

The reason why we call the renderValue() method before calling renderGrid() is because we want the grid to be rendered on top of the highlighted area. This is called the *Painter's Algorithm* where the problem of item visibility is solved by painting objects in the background before objects in the foreground allowing objects closer to the viewer to cover those farther away.

The two methods will need a support method – `cellSize` that returns the size of a single cell based on the allowed maximum and the size of the widget. Let's start by implementing that method.

```
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));
}
```

What essentially takes place is that the best fitting cell size is calculated in each direction separately. Then the smaller of the two values is returned as the size of the cell.

This ensures that the cell is square shaped and all cells in both directions will fit into the widget. Before returning the calculated value we make sure it is not smaller than 4x4 nor smaller than a so

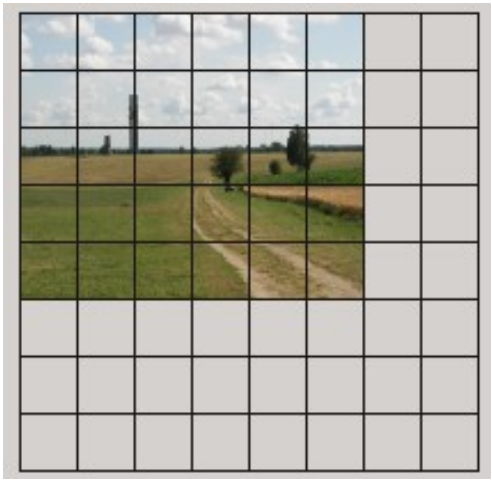
called global-strut. This is to ensure that the area is large enough to be possible to interact with the user.

Knowing the size of a single cell, you can implement the `renderValue()` method. In the body of the method, check whether the pixmap property contains a valid pixmap. If it does, scale the pixmap to the size covering the number of cells determined by the `value` property and render the pixmap to the painter. If there is no pixmap, draw a solid rectangle (set a brush for the painter) instead.

> To be compliant with the platform look&feel you can query the widget's palette() for its highlight() member which holds the widget's highlight's color. You can later change the color by modifying the palette property of the widget in Designer (or Creator's form editor).

The implementation of the `renderGrid()` method is straight forward. Use two nested `for` loops to iterate over the cells (use the `maximum` property to calculate the number of iterations). For each cell, use `QPainter::drawRect()` to draw the outline of the cell. Remember to clear the brush first by setting it to `Qt::NoBrush`.

After building and running the project the effect should be similar to the picture below. Make sure you set a pixmap to the widget to see its live preview.



If the preview image looks distorted, make sure to pass the flag `Qt::SmoothTransformation` to the call that performs pixmap scaling.

> Now you can go back and implement the `sizeHint()` method again and possibly also implement `minimumSizeHint()` to take into consideration the maximum number of rows and columns and the minimum cell size allowed. If you base size hint calculations on any properties that might change during runtime, make sure you call `updateGeometry()` after changing the properties so that the size hint gets recalculated.

Handling mouse events

The final part of the widget is the code for handling mouse events. The user should be able to set

the size of the puzzle by clicking a cell or dragging over it with the mouse. To allow this, the `mousePressEvent()` and `mouseMoveEvent()` methods need to be reimplemented.

Before you start implementing the methods, create a support method called `cellAt()` . Let it take a `QPoint` object as argument. This point represents a coordinate in the widget's coordinate space. The method returns another `QPoint` object which's coordinates identify the cell at the given point. Use the `cellSize()` method implemented earlier when implementing this method.

> When creating new methods pay attention to the signatures you give them. If a method does not modify any members of the current object (represented by variable `this`), it should be declared as const. This makes it callable from other const methods and on behalf of const objects. In our case we can give the `cellAt` method a signature of `QPoint cellAt(const QPoint&) const`.

Finally re-implement the two mouse event handlers mentioned earlier. You can get the position of the event in the widget's local coordinate space by querying the `pos()` method of the argument object of the event. Then it is just a matter of checking the cell coordinates and calling `setValue()`.

Completing the configuration dialog

There are still some minor tasks to attend before the configuration dialog is completed.

After the user chooses his options and accepts the dialog, there need to be a way to ask the dialog to return the values set by the user (which image file path and which puzzle size, in this particular case).

This is implemented by providing getter methods in the dialog class. These getters simply query the widgets in the dialog about the actual values, and provide an easy to use interface for the user of the dialog.

In this case, add the `imageFilePath()` and `puzzleSize()` methods to the dialog and implement them to request the relevant values from the dialog's widgets.

Self Check

- Check that you have used layouts properly by resizing the configuration dialog to see whether the widgets adjust to the new window size.
- Ensure that clicking the Browse button and choosing an image file causes the puzzle size widget to show a live preview of the puzzle.
- Ensure that clicking sections of the puzzle size widget causes the preview to adjust to the desired size.
- Make sure all the methods changing the property values contain an `update()` call if the change might influence painting the custom widget.
- Ensure that clicking the Ok button causes `QDialog::exec()` to return `QDialog::Accepted` and clicking the Cancel button returns `QDialog::Rejected`. If this does not work, check whether the signals from the buttons are connected to the `accept()` and `reject()` slots of the dialog.

The game

In this section of the lab you will use the Graphics View framework to implement the actual game. Two things need to be done here: you need a puzzle piece item that will be used to populate the scene, and you need a mechanism for joining pieces and checking the whether the puzzle is complete.

But first, let's create a proper environment for the puzzle.

Preparing the canvas

To be able to develop the puzzle piece item class, create a setup for the game so that you'll be able to test the custom puzzle piece item.

In the `main` function, modify the project so that it constructs an instance of `QGraphicsView` and an instance of `QGraphicsScene` and binds them together using the view's `setScene()` call. Remember about adding a call to show the view and also to replace the `exec` call in the dialog with a call to `QApplication::exec()` method, as we won't be needing the configuration dialog for a while.

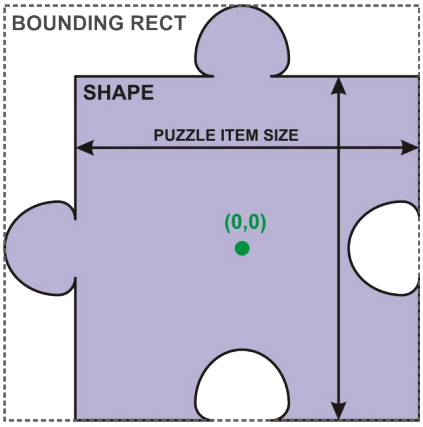
When you run the program you should see a window with a blank canvas. That is going to be your puzzle board.

The puzzle piece item

A jigsaw puzzle is composed of puzzle pieces. You could just use an already available class for representing the piece such as `QGraphicsRectItem` but instead you will create a custom piece with a more delicate shape than just a rectangle. This will add to the jigsaw feel of the application

The picture below shows a sketch of a puzzle item. The origin is located in the center of the square part of the item.

The shape of the piece may extend past the main square part of the piece to form the connectors. This means that the bounding rectangle needs to contain the connectors as well. Of course, as different pieces have different connectors, the shape and bounding rectangle will change. However, it is important to keep the origin of each item in the center of the main part of the piece.



You will now implement the item class. Start with a subclass of the `QGraphicsPathItem` class. For simplicity, the size of each puzzle piece will be constant regardless of the size of the image used to build the puzzle. This means that the image may have to be scaled to adjust to the size.

Let's assume that each puzzle piece is 50 by 50 pixels.

Looking at the sketch of the piece again, you can see that the piece has up to four connectors.

Each of them can be extending out of the piece or into the piece. In addition to this, some sides of the piece can be without connectors (i.e. an edge piece).

In your piece class, add an enum describing the three possible connector configurations.

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

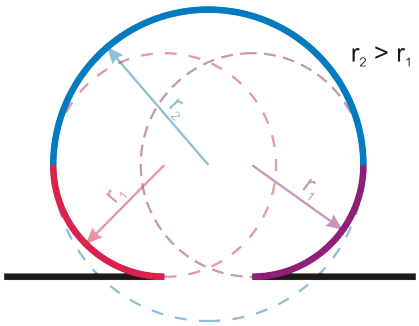
Now you can describe the exact shape of the item by providing values for all four connector positions in the item. Create a constructor for the item that takes four ConnectorPosition values and stores them in private member variables of the class.

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

The constructor should then call some internal method that will construct the piece shape. Call that method `constructShape()`.

You will not be given step by step instructions for constructing the shape , but you will be given some guidelines. If you don't feel up to the task of constructing a path with round connectors, then settle on something easier. For example make the connector deltoid or square.

The general rule for constructing the shape using `QPainterPath` class is to first use `moveTo()` to move the “pen” to one of the corners of the piece.Then to use `lineTo()` for adding sections of the puzzle. Remember to use coordinates relative to the origin of the item. The origin is located in the center of the each piece. Since the piece is 50px wide and 50px high, the upper-left corner of the piece has coordinates (-25, -25).



The diagram to the left shows how to construct a puzzle connector shape by adding three arcs together. For instance, entering from the left via the black line, you then add the red arc (which is a quarter of a circle). Then add the blue one (a half circle with a larger radius). Finally add the purple arc, which is a mirror of the red arc. You then continue adding the other black line. Continue this way round the entire piece.

If you don't feel comfortable with calculating the arc parameters, simply substitute them with a simpler shape. In either case, use the connector position values passed to the constructor to decide whether to draw a connector and in which direction.

When all your lines and connectors have been added, call `QPainterPath::closeSubpath()`. Then set the item's path using `QGraphicsPathItem::setPath()`method. The base class of the path item will take care of providing a proper bounding rectangle based on the path specified.

At this point you should be able to instantiate different puzzle pieces and add them to the scene. The next step is to provide proper painting of each piece.

In order to do that puzzle needs to know the texture for each piece. For that purpose, declare the `setPixmap()` and `pixmap()` methods for setting and retrieving the pixmap from the item. Store the pixmap as a private member of the puzzle piece item. Make sure that the setter calls `update()` as that affects the appearance of the piece.

To have a complete painting of each piece, all you need to do is to create a painting routine that fills the shape of the path with the pixmap.

The easiest way to do that is to limit the drawing area to the one occupied by the path and then paint the pixmap in that area. This is achieved using clipping. Re-implement the item's `paint()`

method. Use `QPainter::setClipPath()` to limit the drawing area and then draw the pixmap to the item.

You can assume that the pixmap has the size of the item's bounding rectangle, i.e. it will cover the connectors when needed. Finally call the base class implementation of paint to give the item an outline.

You are now done with the item for now.

Game logic

Start the implementation of the game logic by subclassing the `QGraphicsScene` class. In your new class, add a method for starting the game that accepts the size of the puzzle and the pixmap to use.

The first task for the method should be to clear the scene for any items, so that you always start with an empty board. After that, write two nested for loops that iterate over rows and columns of the puzzle. In each iteration you will have to create a puzzle piece item.

For each item, you need to decide the connector positions and -types each piece will have so that the pieces match. Connector positions should be randomized. You can use a temporary array (i.e. a vector) of the type `ConnectorPosition` representing the connector positions between each piece. This will make it possible to create pieces that fit properly between rows and columns. You can use the following guidelines when creating the pieces.

- Initialize each item of the connector vector with value `None`.
- If the current column is 0 then the west connector is set to `None`.
- If the current column is the last one in a row, the east connector is set to `None`, otherwise it is chosen randomly and stored in a helper variable.
- If the south connector of the item in the same column of the previous row (stored in the helper vector) is set to `In` then the north connector in the current item is set to `Out` and vice-versa.
- If the current row is the last one, the south connector is set to `None`, otherwise a random position is chosen for the south connector and also stored in the vector for the current column.
- If the helper variable is set to `Out`, the west connector of the current item is set to `In` and vice-versa, if set to `None` then stays `None`.
- At the beginning of processing each row, set the helper variable to `None`.

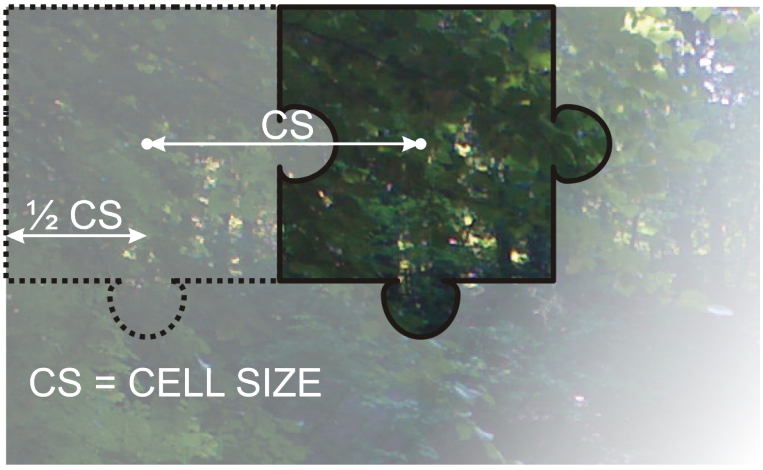
Try implementing this algorithm yourself. Remember about adding the item to the scene and placing it in a proper position. Also ensure that `ItemIsMovable` flag is set for each item. You can check your implementation against the code below.

> To get a pseudo-random value, use a `qrand() % n` statement where `n` is the number of values to choose from. You will get a result in range from 0 to n-1. You should also initialize the random number generator once with a random seed using `qsrand()`. You can use the current time for the seed.

```
/*!
 * Helper function for reversing connector positions
 */
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
    }
}
```

For now, use a hard-code size for the board (e.g. set it to 5x5 pieces) and call the method you just implemented directly from your `main` function. When running the program, you should see empty puzzle pieces set up nicely in a grid with matching connectors.

Now you need to complete the pieces by adding the code responsible for assigning the appropriate part of the original image to each piece of the puzzle. The illustration below shows how to calculate the offset of each piece from the upper-left corner of the image. The offset of the first row and the first column is half the cell size (no west nor north connectors) so in this case 25 pixels. The offset of each next row and each next column is cell size (50px) larger.



Extend the setup method with setting the pixmap to each of the items. Use `QPixmap::copy()` to copy part of a pixmap into a new pixmap. You can use the following piece of code to get the right part of the image.

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

After hard-coding a pixmap and running the code, you should get a window with a solved puzzle.

Based on what you have done so far you should now be able to perform the next task yourself without any special instructions. Make the items appear at random positions within the scene

The remaining piece of the puzzle is to merge correctly positioned pieces and checking if the complete puzzle has been put together.

First add a getter and a setter for another item attribute; a `QPoint` called `coordinates` containing the coordinates of a puzzle piece represented by each item. For instance, the top-left corner piece of the puzzle has coordinates (0,0), the item to the right has coordinates (1,0) and so on.

This will let you to check whether two items next to each other are in the right order. After you have implemented the `coordinates()` and `setCoordinates()` method, use the implemented setter in the loop populating the scene to provide each item with a set of coordinates.

Each puzzle piece can be connected with up to four neighboring pieces, one for each direction. Create a `Direction` enumeration in the piece class containing values `North`, `East`, `South`, `West`.

Then add a 4 element `m_neighbors` array of pointers to the puzzle piece item class as a private member variable of the class. Initialize each element of the array with 0 in the class constructor to note that by default the piece is not connected to any other pieces.

Next, add a public method for linking the items. Let the method take a pointer to the item that is to be linked to, and a `Direction` enum value for determining the direction of the connection. Remember to link both ways. If item A is connected to item B from the `East` side, then item B is connected to item A from the `West` side, etc.

To link items automatically when they are in the right position, you will need a method available

that will search through the neighborhood of the item for items with appropriate coordinates. The algorithm will be triggered by mouse release event. Re-implement the mouseReleaseEvent event handler method in your item class. From that handle, invoke the algorithm described next.

The neighbor checking algorithm is a recursive scan of linked items. In each call the item tries to find new neighbors and afterwards calls itself on each of the linked items. To prevent infinite loops, a stop condition is introduced. The algorithm carries a reference to a set of already visited nodes. When it detects that the current node has already been visited, it stops the recursion and returns.

```
void PuzzlePiece::checkNeighbors(QSet<QPoint> &checked){
  if(checked.contains(coordinate()))
      return; // stop condition
  checked.insert(coordinate()); // add yourself to visited
  findneighbor(North);          // find N neighbor
  findneighbor(East);           // find E neighbor
  findneighbor(South);          // find S neighbor
  findneighbor(West);           // find W neighbor

  // recursive calls:
  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);
}
```

From the mouse release event handler, create an empty set and call the `checkNeighbors()` method of an item. Also, do not forget to call the base class implementation of the event handler or else you will lose ability to move items.

A non-const reference to the set is used here so that all recursive calls operate on the exact same object and all changes made to the object will be seen by each and every recursive call.

The `checkNeighbors` method relies on the `findNeighbor` method. The method takes a direction and tries to locate an item that should be positioned in this direction relative to the current item. Implement this method based on the following guidelines.

- If there already is a linked element in this direction, stop searching.
- The item you are looking for represents the piece next to the current piece. Use the `coordinates` property to determine this.
- The item should have a position that changes according to the size of a puzzle piece (50px) in the requested direction. Use `QGraphicsScene::itemAt()` to ask the scene for an item that collides with a given point.
- It is best to provide a tolerance of a few of pixels from the expected location of a neighbor. You can calculate the difference of positions and use `QPoint::manhattanLength()` to check if it is small enough (e.g. less than 7 pixels) to treat the item as properly positioned – this will allow an average difference of 3 pixels in each direction;
- If you find a match, link the items together and move one of the pieces so that they are

perfectly aligned to each other (you can make use of the expected position calculated earlier).

> Manhattan length is a special distance metric calculated as a sum of distances between points in each dimension. For a 2D plane the distance between points P(x1, y1) and Q(x2,y2) is |x1-x2| + |y1-y2|. It's much faster to calculate than the Euclidean distance and is a meaningful approximation of the latter in many situations (mainly when we're interested in relative distance between many points rather than their absolute distance).

For the puzzle to behave as expected, one more feature needs to be added. Items linked together need to move in unison. The easiest way to do that is to re-implement the `itemChange()` method. When an item receives an `ItemPositionHasChanged` change, it simply moves all tiles that it is linked to into their new positions.

This will trigger position changes in those items, which in turn trigger calls to their `itemChange` methods. They, in turn, will move their neighbors into position. As a result all linked items will snap into place. Remember to subscribe to receiving geometry changes in the items by setting the `ItemSendsGeomentryChanges` flag for each item.

The only thing left is to check if the puzzle is complete. The game is complete when all items are linked to each other. This happens when the set passed to `checkNeighbors` contains all the elements in the puzzle when the function is done, so a simple `QSet::count()` does the trick.

Self-Check

- Check that there is an appropriate number of puzzle pieces on the board.
- Check that puzzle connectors match and that items lying on the edge of the puzzle don't have connectors sticking out over the edge.
- Check that items “snap” into place when put at an appropriate place next to a neighbor piece; if it doesn't happen, check the implementation of the algorithm for finding neighbors.
- Check that moving a piece linked to another piece causes the other piece to move along the dragged item.
- Make sure that solving the jigsaw puzzle triggers the win condition.

Putting it all together

To build a complete game from the pieces, you need to add the configuration dialog to the application again.

When handling the dialog make sure you respect the return value of `QDialog::exec()`. If the user cancels the dialog then quit the program without showing the board.

Also, make sure to handle the situation that occurs when the user has chosen an invalid image file. A good approach would be to show a `QMessageBox` with an error message before returning to the configuration dialog again.