LECTURE 18

GUI Programming Part 2

GUI PROGRAMMING

Our goal: the triangle peg game!

BASIC PYQT

Last lecture, we created a basic PyQt4 application which had a few buttons and a menu bar.


```
class PegGameWindow(QtWidgets.QMainWindow):
    def __init__(self):
        ...
    def setup(self):
        ...
        self.central_widget = QtWidgets.QWidget(self)
        new_button = StartNewGameBtn(self.central_widget)
        quit_button = QuitBtn(self.central_widget)
        self.setCentralWidget(self.central_widget)

        exit_action = QtWidgets.QAction('Exit', self)
        exit_action.triggered.connect(QtWidgets.qApp.quit)

        menu_bar = self.menuBar()
        file_menu = menu_bar.addMenu('File')
        file_menu.addAction(exit_action)

        self.show()
```

LAYOUT MANAGEMENT

Before we can add more components to our application, we need to talk about layout management in PyQt4.

The two options for managing the position of widgets are

- Absolute positioning
- Layout classes

Absolute positioning is as simple as calling the `move()` method on a widget with some arguments that specify a position (x,y). This can be impractical for a few reasons. First, applications might look differently on different platforms (or even using different fonts) as they won't scale. Secondly, changes in layout will be much more tedious.

LAYOUT MANAGEMENT

Layout classes automatically position and resize widgets to accommodate space changes so that the look and feel is consistent.

Every `QWidget` subclass may have a layout specified which gives a widget control over:

- Positioning of child widgets.
- Sensible default sizes for widgets.
- Sensible minimum sizes for widgets.
- Resize handling.
- Automatic updates when contents change (font size, text or other contents of child widgets, hiding or showing a child widget, removal of child widgets).

LAYOUT MANAGEMENT

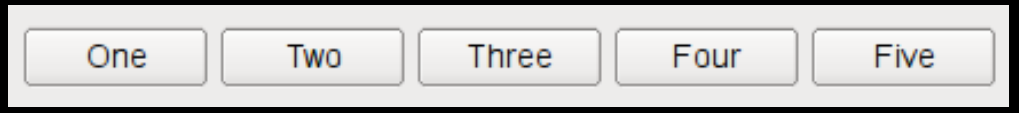
There are a large number of layout classes but the most common include:

- Box Layout (`QBoxLayout`): lays out widgets in a horizontal row (`QHBoxLayout`) or vertical column (`QVBoxLayout`) from left-to-right or top-to-bottom.
- Grid Layout (`QGridLayout`): lays out widgets in a 2-D grid where widgets can occupy multiple cells.
- Form Layout (`QFormLayout`): layout class for managing forms. Creates a two-column form with labels on the left and fields on the right.

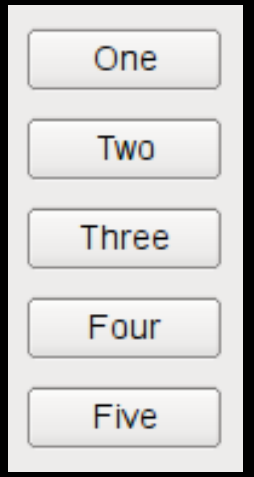
BOX LAYOUT

The `QBoxLayout` class lines up child widgets horizontally or vertically. `QBoxLayout` will take a given space and divide it up into boxes that contain widgets.

- `QHBoxLayout` makes horizontal rows of boxes.



- `QVBoxLayout` makes vertical columns of boxes.



BOX LAYOUT

Once you create a box layout and attach it to a parent widget, the following methods are used to add child widgets and manage the space:

- `addWidget(widget, stretch=0)` to add a widget to the `QBoxLayout` and set the widget's stretch factor.
- `addSpacing(size)` to create an empty (non-stretchable) box with a particular size.
- `addStretch(stretch=0)` to create an empty, stretchable box.
- `addLayout(layout, stretch=0)` to add a box containing another `QLayout` to the row and set that layout's stretch factor.

Stretch factors indicate the relative amount of leftover space that should be allocated to a block.

GRID LAYOUT

The Grid Layout class is the most universal, but we will use both Grid and Box layouts.

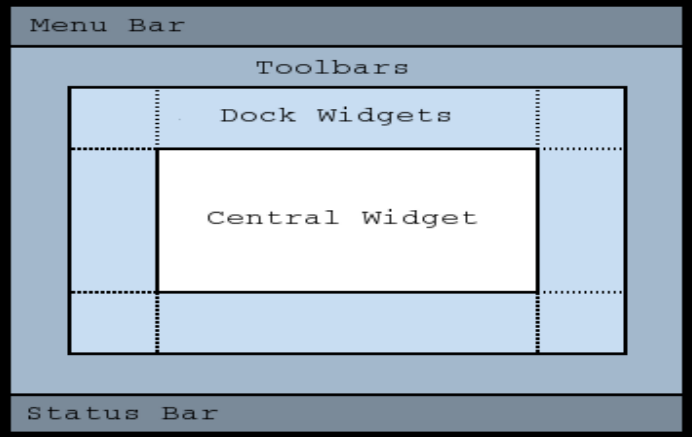
A grid is represented with multiple rows and columns. Widgets can be attached to the grid by indicating the (row, column) space it should fill.

- Create a grid layout with `QtWidgets.QGridLayout()`.
- Attach widgets to the grid with `addWidget(`*`QWidget`*`, `*`row`*`, `*`column`*`)`.

You can also set the number of grid spaces that it should take up.

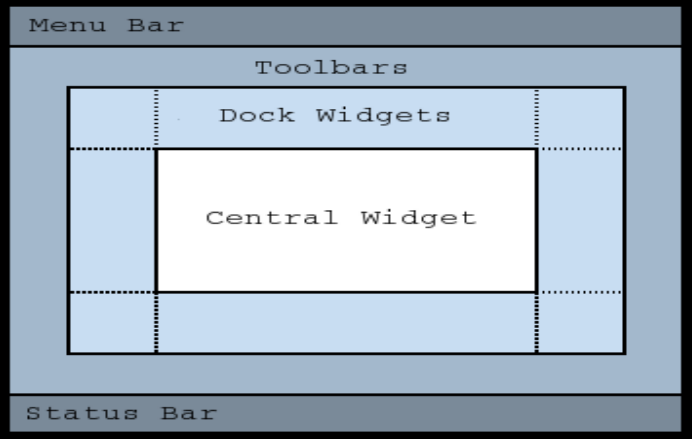
BASIC PYQT

Recall that `QMainWindow` includes a default layout for traditional GUI components like a menu bar, status bar, tool bar, etc. The focal point of the application is stored in the Central Widget of the layout.




```
class PegGameWindow(QtWidgets.QMainWindow):
    def __init__(self):
        ...
    def setup(self):
        ...
        self.central_widget = QtWidgets.QWidget(self)
        self.new_button = StartNewGameBtn(self.central_widget)
        self.quit_button = QuitBtn(self.central_widget)
        self.setCentralWidget(self.central_widget)

        exit_action = QtWidgets.QAction('Exit', self)
        exit_action.triggered.connect(QtWidgets.qApp.quit)

        menu_bar = self.menuBar()
        file_menu = menu_bar.addMenu('File')
        file_menu.addAction(exit_action)

        self.show()
```

BASIC PYQT



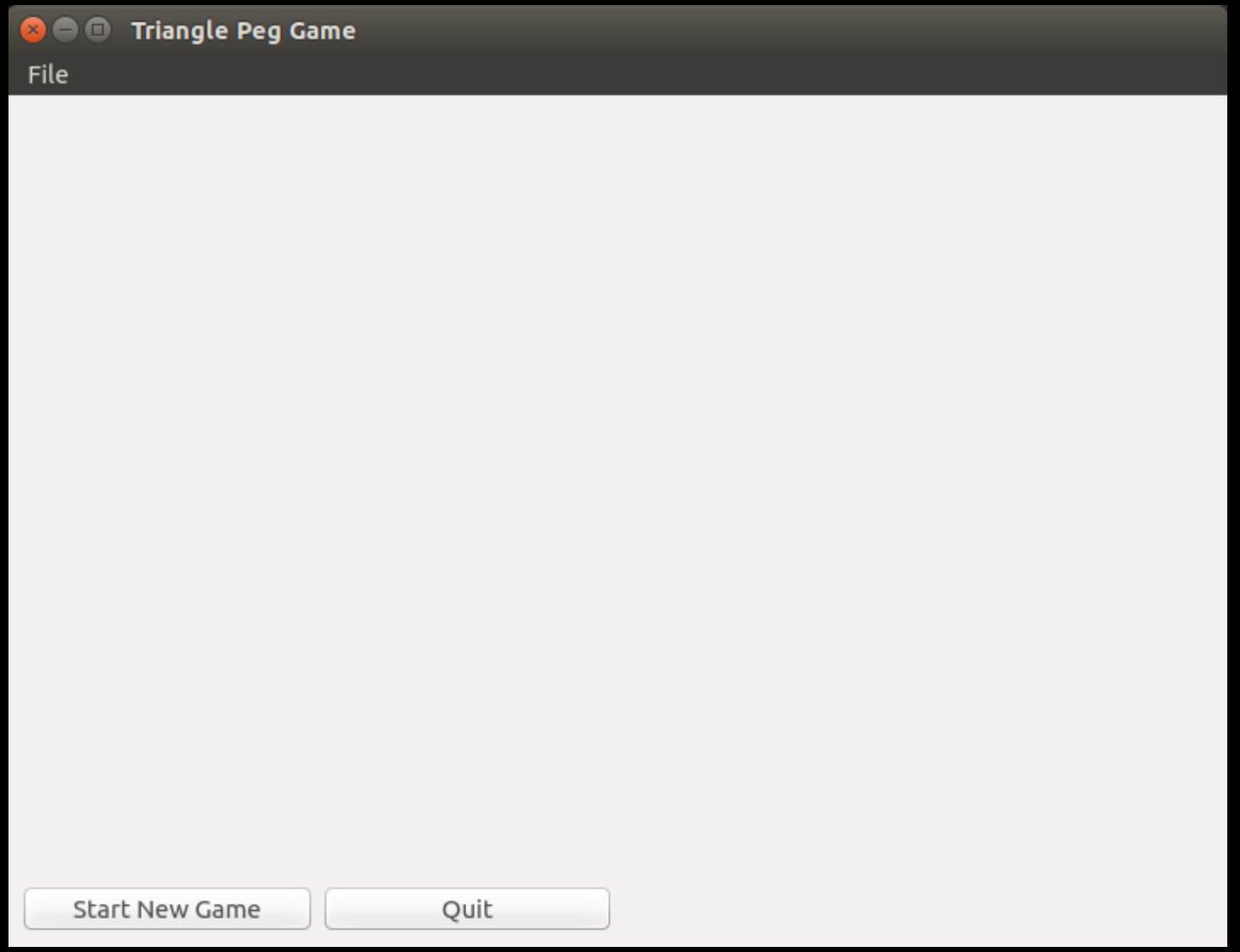

```
class PegGameWindow(QtWidgets.QMainWindow):
    def __init__(self):
        QtWidgets.QMainWindow.__init__(self)
        self.setup()

    def setup(self):
        self.setWindowTitle('Triangle Peg Game')
        self.setToolTip("Play the triangle peg game!")

        self.peg_game = PegGame(self)
        self.setCentralWidget(self.peg_game)
        ...
        self.show()
```

Let's really define what this central widget should look like. We'll create a new class called PegGame.

GRID LAYOUT

So what do we want PegGame to look like?

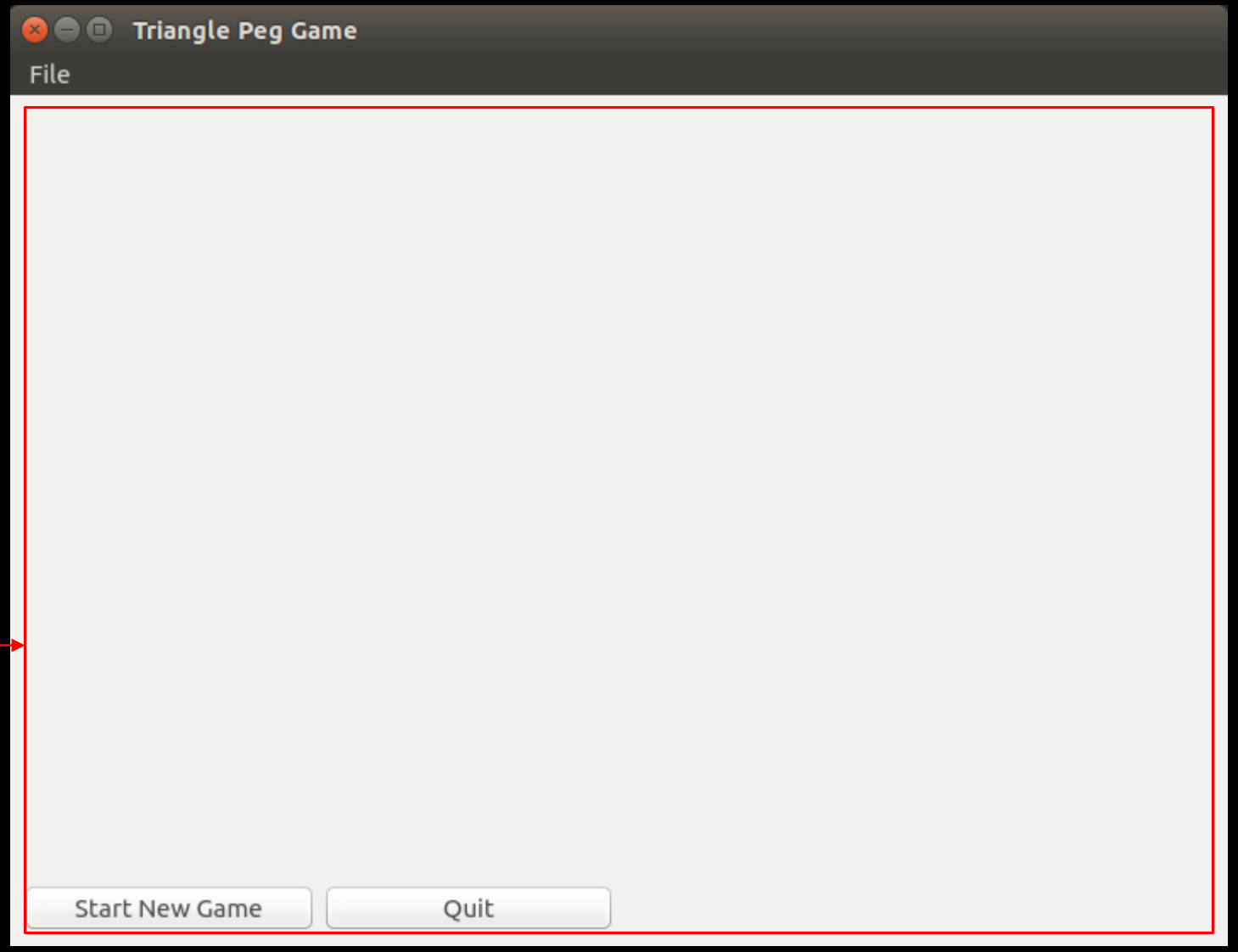
Well it's not too interesting. But let's point out the important details.



GRID LAYOUT

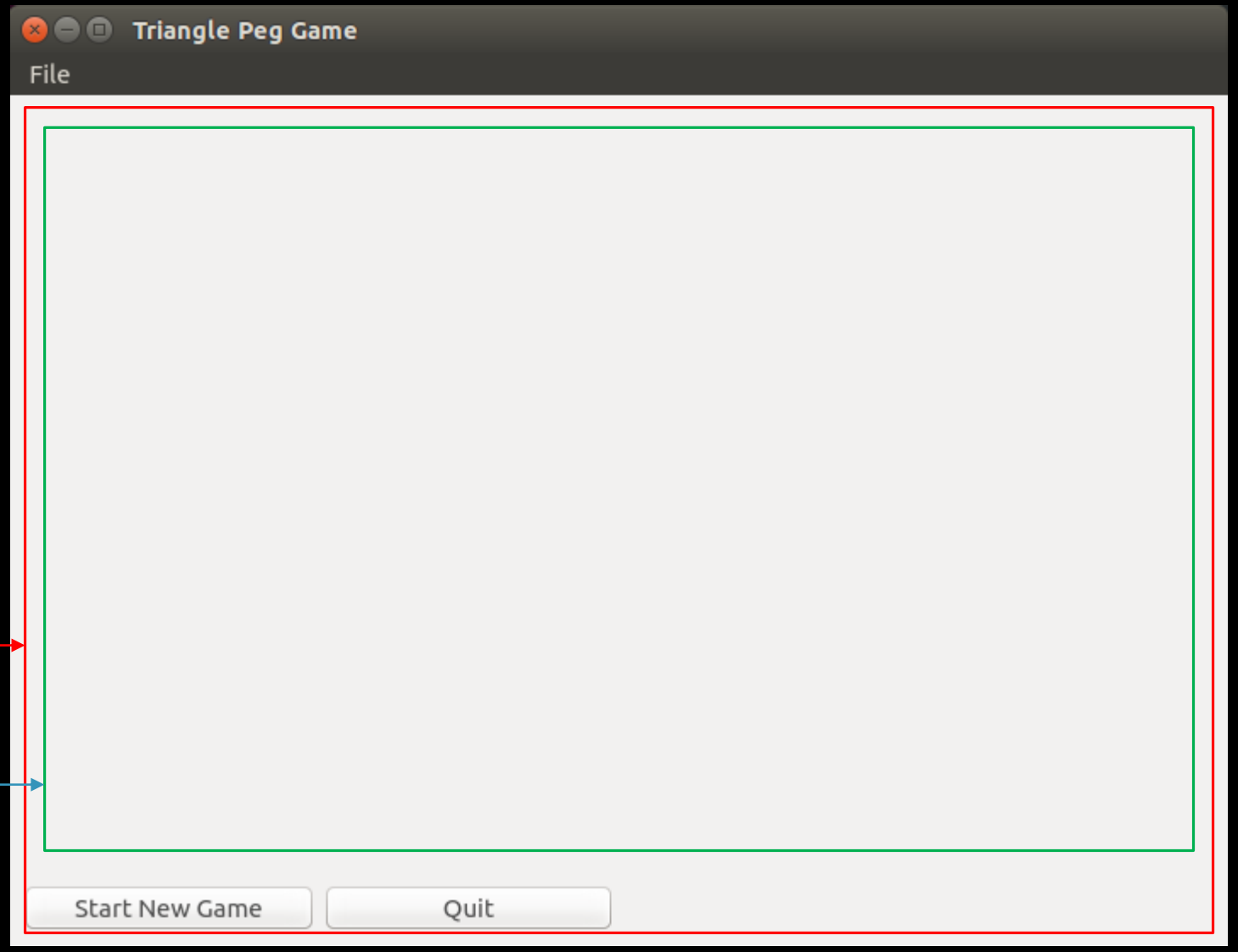
So what do we want PegGame to look like?

Well it's not too interesting. But let's point out the important details.



GRID LAYOUT

PegGame will contain three important components (at least, to start):
- PegBoard, which will house the board, pegs, etc.
- Start New Game Button
- Quit Button



GRID LAYOUT

So how do we define `PegGame`?

`PegGameWindow` has a built-in layout because it inherits from `QMainWindow`.

`PegGame`, however, is a plain widget.
So we can associate a Grid layout with it.

Our three components

```
class PegGame(QtWidgets.QWidget):
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        self.setup()

    def setup(self):
        self.board = PegBoard(self)
        self.new_btn = StartNewGameBtn(self)
        self.quit_btn = QuitBtn(self)
        self.grid = QtWidgets.QGridLayout()
        self.setLayout(self.grid)
        self.grid.addWidget(self.board, 1, 1, 1, 4)
        self.grid.addWidget(self.new_btn, 2, 1, 1, 1)
        self.grid.addWidget(self.quit_btn, 2, 2, 1, 1)
```

GRID LAYOUT

So how do we define `PegGame`?

```
class PegGame(QtWidgets.QWidget):
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        self.setup()

    def setup(self):
        self.board = PegBoard(self)
        self.new_btn = StartNewGameBtn(self)
        self.quit_btn = QuitBtn(self)
        self.grid = QtWidgets.QGridLayout()
        self.setLayout(self.grid)
        self.grid.addWidget(self.board, 1, 1, 1, 4)
        self.grid.addWidget(self.new_btn, 2, 1, 1, 1)
        self.grid.addWidget(self.quit_btn, 2, 2, 1, 1)
```

Creating and setting our grid layout

GRID LAYOUT

So how do we define `PegGame`?

```
class PegGame(QtWidgets.QWidget):
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        self.setup()

    def setup(self):
        self.board = PegBoard(self)
        self.new_btn = StartNewGameBtn(self)
        self.quit_btn = QuitBtn(self)
        self.grid = QtWidgets.QGridLayout()
        self.setLayout(self.grid)
        self.grid.addWidget(self.board, 1, 1, 1, 4)
        self.grid.addWidget(self.new_btn, 2, 1, 1, 1)
        self.grid.addWidget(self.quit_btn, 2, 2, 1, 1)
```

Adding our components to the grid layout.

row, column

GRID LAYOUT

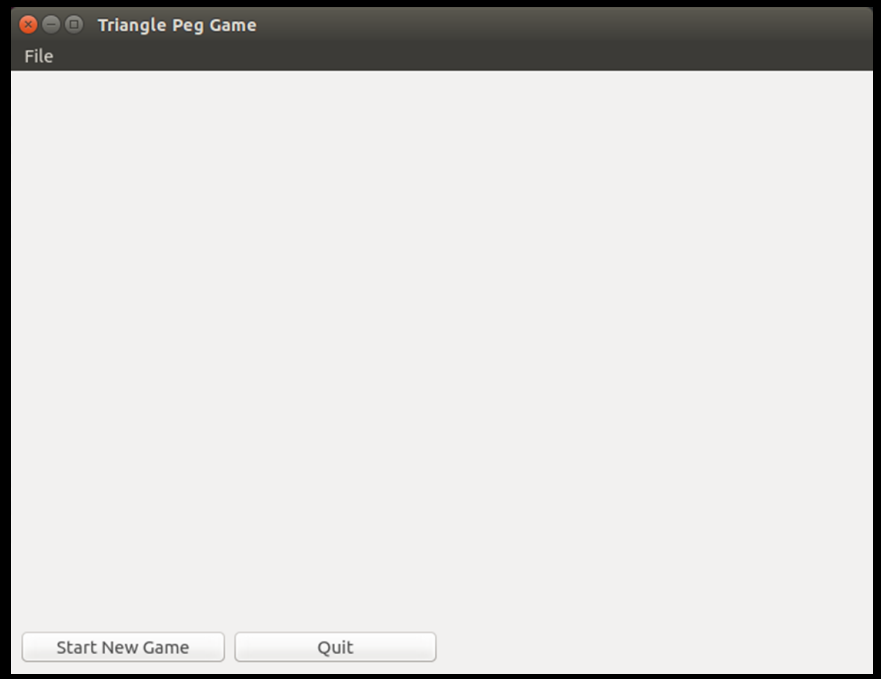
So how do we define `PegGame`?

```
class PegGame(QtWidgets.QWidget):
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        self.setup()

    def setup(self):
        self.board = PegBoard(self)
        self.new_btn = StartNewGameBtn(self)
        self.quit_btn = QuitBtn(self)
        self.grid = QtWidgets.QGridLayout()
        self.setLayout(self.grid)
        self.grid.addWidget(self.board, 1, 1, 1, 4)
        self.grid.addWidget(self.new_btn, 2, 1, 1, 1)
        self.grid.addWidget(self.quit_btn, 2, 2, 1, 1)
```

Adding our components to the grid layout.

height, width

GRID LAYOUT




```
class PegGame(QtWidgets.QWidget):
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        self.setup()

    def setup(self):
        self.board = PegBoard(self)
        self.new_btn = StartNewGameBtn(self)
        self.quit_btn = QuitBtn(self)
        self.grid = QtWidgets.QGridLayout()
        self.setLayout(self.grid)
        self.grid.addWidget(self.board, 1, 1, 1, 4)
        self.grid.addWidget(self.new_btn, 2, 1, 1, 1)
        self.grid.addWidget(self.quit_btn, 2, 2, 1, 1)
```

PEGBOARD

So, what does the `PegBoard` look like so far?

```
class PegBoard(QtWidgets.QWidget):
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        self.setFixedSize(700, 460)
```

The parent we passed in was the `PegGame` instance that holds the `PegBoard`.

The `setFixedSize(`*`width, height`*`)` method gives us a convenient way to not only define a size for the `PegBoard` instance, but also set a lower bound on the size of the parent containers (`PegGame` and `PegGameWindow`).

PEGBOARD

Here's our first goal: let's get a wooden triangle.

To get this, we need to introduce the QPainter object.

QPAINTER

The `QtGui.QPainter` class performs low-level painting on any object that inherits the `QtGui.QPaintDevice` class (e.g. `QWidget`, `QPixmap`, `QPicture`).

```
qp = QtGui.QPainter()
qp.begin(canvas)
...
qp.end()
```

The most basic usage involves creating a QPainter instance, calling the begin method with the QPaintDevice to which we will be painting, and then calling the end method.

QPAINTER

```
qp = QtGui.QPainter()
qp.begin(canvas)
...
qp.end()
```

So what can we do in those ellipses?

- Manipulate settings
  - `font()`, `brush()`, `pen()` give you access to the tools used to draw. Return `QFont`, `QBrush`, and `QPen` objects.
  - Also `setFont(`*`font`*`)`, `setBrush(`*`brush`*`)`, `setPen(`*`pen`*`)`.
- Draw
  - `drawEllipse()`, `drawPolygon()`, `drawImage()`, `drawLine()`, `drawPath()`, `drawPicture()`, `drawPie()`, etc…
  - If you can dream it, you can draw it. ☺

QPAINTER

First thing to notice is the method name we're using: paintEvent.

paintEvent is a method called automatically whenever the widget's appearance is updated. This includes when we first show the widget, but also when we request an update.

```
class PegBoard(QtWidgets.QWidget):
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        self.setFixedSize(700, 460)

    def paintEvent(self, event):
        points_list = [QtCore.QPoint(50,455),
                       QtCore.QPoint(650,455),
                       QtCore.QPoint(350,5)]
        triangle = QtGui.QPolygon(points_list)
        qp = QtGui.QPainter()
        qp.begin(self)
        qp.drawPolygon(triangle)
        qp.end()
```

QPAINTER

We create a list of QPoint objects representing the vertices of our triangle.

```
class PegBoard(QtWidgets.QWidget):
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        self.setFixedSize(700, 460)

    def paintEvent(self, event):
        points_list = [QtCore.QPoint(50,455),
                       QtCore.QPoint(650,455),
                       QtCore.QPoint(350,5)]
        triangle = QtGui.QPolygon(points_list)
        qp = QtGui.QPainter()
        qp.begin(self)
        qp.drawPolygon(triangle)
        qp.end()
```

QPAINTER

```
class PegBoard(QtWidgets.QWidget):
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        self.setFixedSize(700, 460)

    def paintEvent(self, event):
        points_list = [QtCore.QPoint(50,455),
                       QtCore.QPoint(650,455),
                       QtCore.QPoint(350,5)]
        triangle = QtGui.QPolygon(points_list)
        qp = QtGui.QPainter()
        qp.begin(self)
        qp.drawPolygon(triangle)
        qp.end()
```

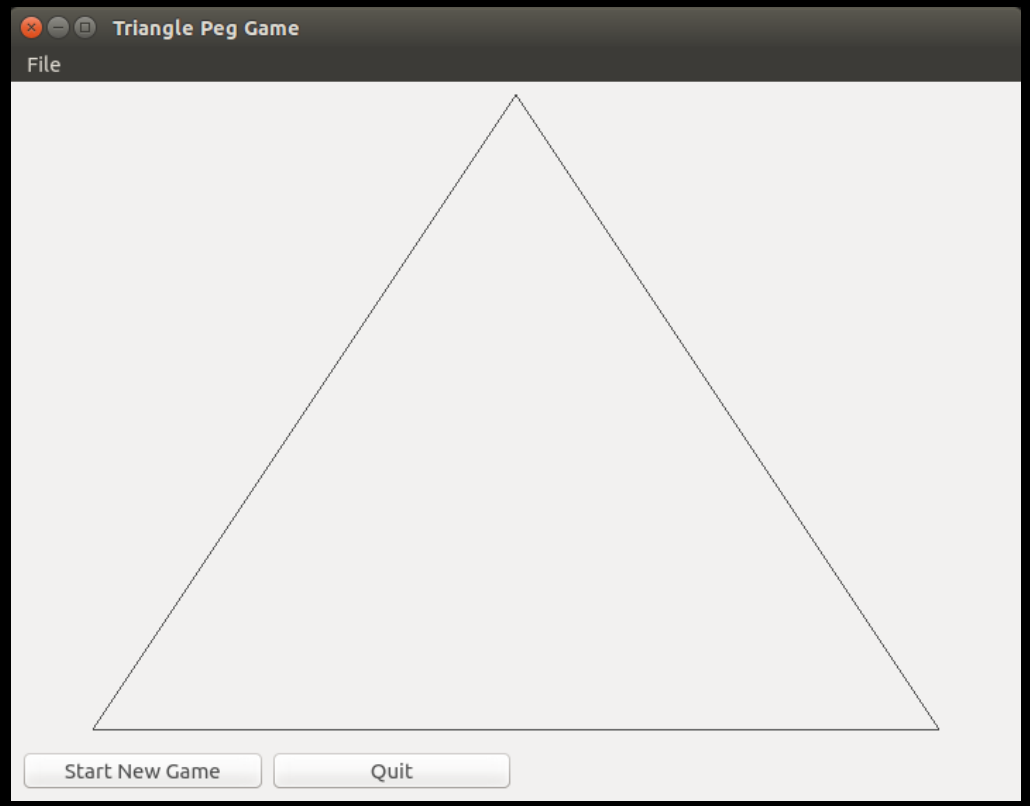
Using these points, we initialize a QPolygon object representing our triangle.

QPAINTER

```
class PegBoard(QtWidgets.QWidget):
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        self.setFixedSize(700, 460)

    def paintEvent(self, event):
        points_list = [QtCore.QPoint(50,455),
                       QtCore.QPoint(650,455),
                       QtCore.QPoint(350,5)]
        triangle = QtGui.QPolygon(points_list)
        qp = QtGui.QPainter()
        qp.begin(self)
        qp.drawPolygon(triangle)
        qp.end()
```

Finally, we create our QPainter, make the PegBoard instance our QPaintDevice and draw the triangle!

QPAINTER




```
class PegBoard(QtWidgets.QWidget):
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        self.setFixedSize(700, 460)

    def paintEvent(self, event):
        points_list = [QtCore.QPoint(50,455),
                       QtCore.QPoint(650,455),
                       QtCore.QPoint(350,5)]
        triangle = QtGui.QPolygon(points_list)
        qp = QtGui.QPainter()
        qp.begin(self)
        qp.drawPolygon(triangle)
        qp.end()
```

QPAINTER

We want to make two changes: get rid of the black outline and apply a wood grain fill.

Grab the curren QPen object, set the color to transparent, and reset Qpen object.

Create a new QBrush object, set the texture image to a local picture of some wood. Set the QBrush object.

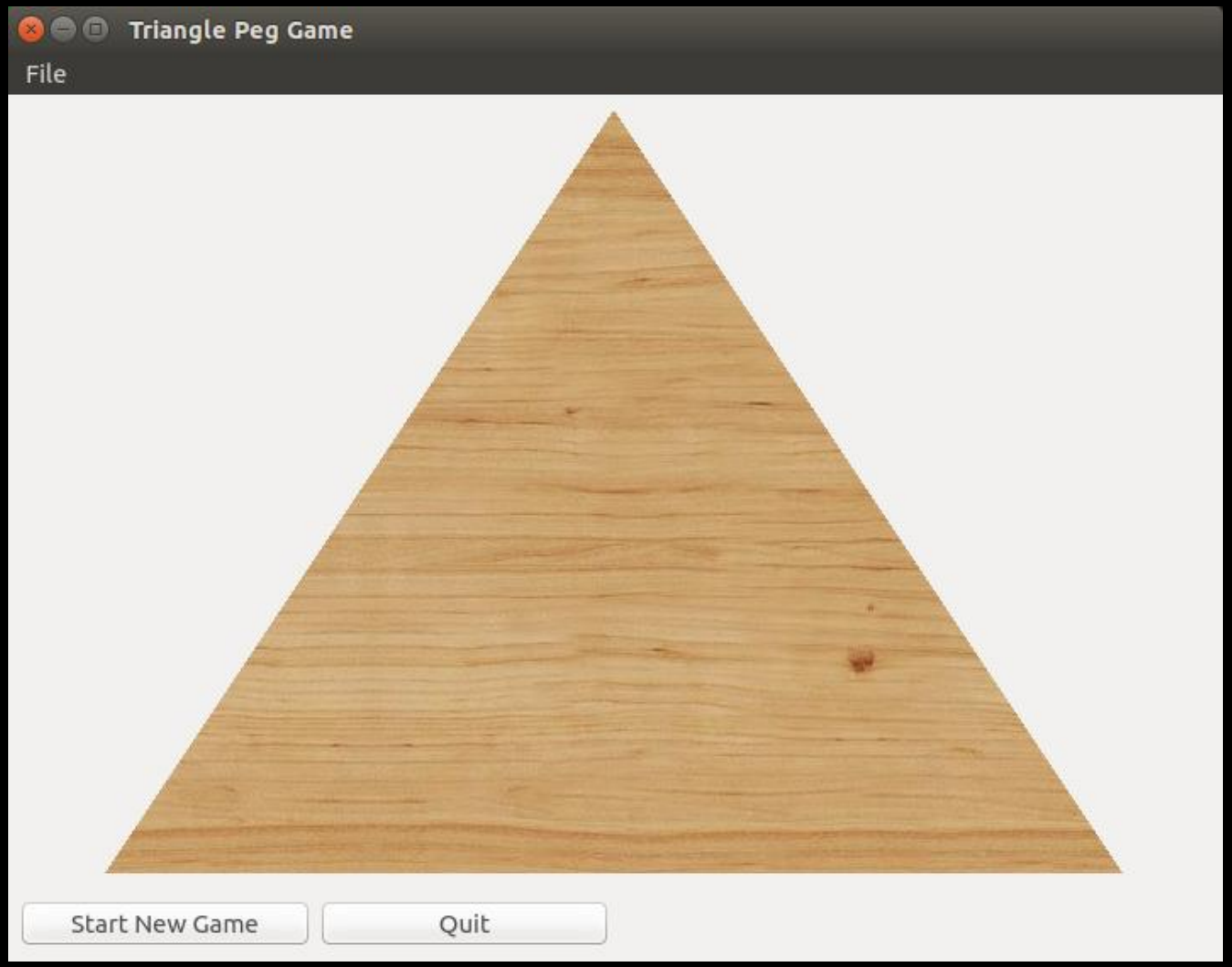
```
qp = QtGui.QPainter()
qp.begin(self)
pen = qp.pen()
pen.setColor(QtCore.Qt.transparent)
qp.setPen(pen)
brush = QtGui.QBrush()
brush.setTextureImage(QtGui.QImage('wood.jpg'))
qp.setBrush(brush)
qp.drawPolygon(triangle)
qp.end()
```

QPAINTER

Ok, so what if we want a black background?

Rather than drawing *on* the widget, that requires us to change the way the widget *draws itself*.

Specifically, we'll need to change the way the PegBoard widget displays.



QPALETTE

To do this, we need to introduce the idea of a `QPalette`.

The `QPalette` class contains color groups for each widget state. It describes how the widget should render itself on the screen for each state.

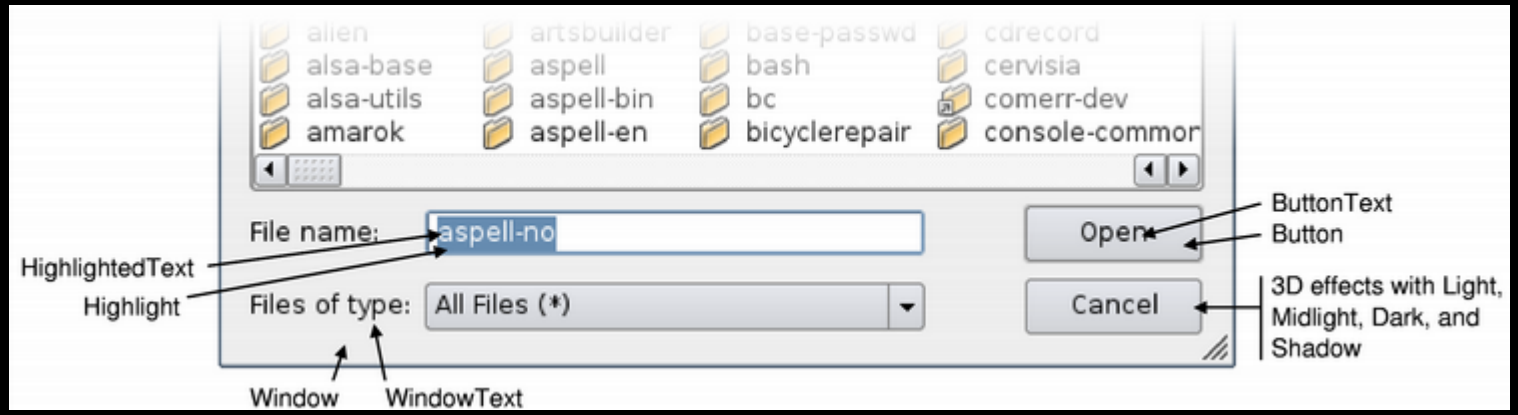
A palette consists of three color groups: *Active (keyboard focus)*, *Disabled (not in focus)*, and *Inactive (disabled)*. All widgets in Qt contain a palette and use their palette to draw themselves.

`QWidget`'s `palette()` method returns the currently used `QPalette` object and `setPalette(`*`palette`*`)` allows you to reassign the `QPalette` object being used.

QPALETTE

For each state of a widget, there are many roles of which we need to describe the look and feel. Each role has an assigned color and brush.

- Window
- Background
- WindowText
- Foreground
- Button
- Etc.



QPALETTE

Colors and brushes can be set for particular roles in any of a palette's color groups with `setColor()` for color and `setBrush()` for color, style, and texture.

Calling, for example, `backgroundRole()` on a widget will return the current brush from the widget's palette that is being used to draw the role. You can also get this, and any other brush, as `palette.Background`.

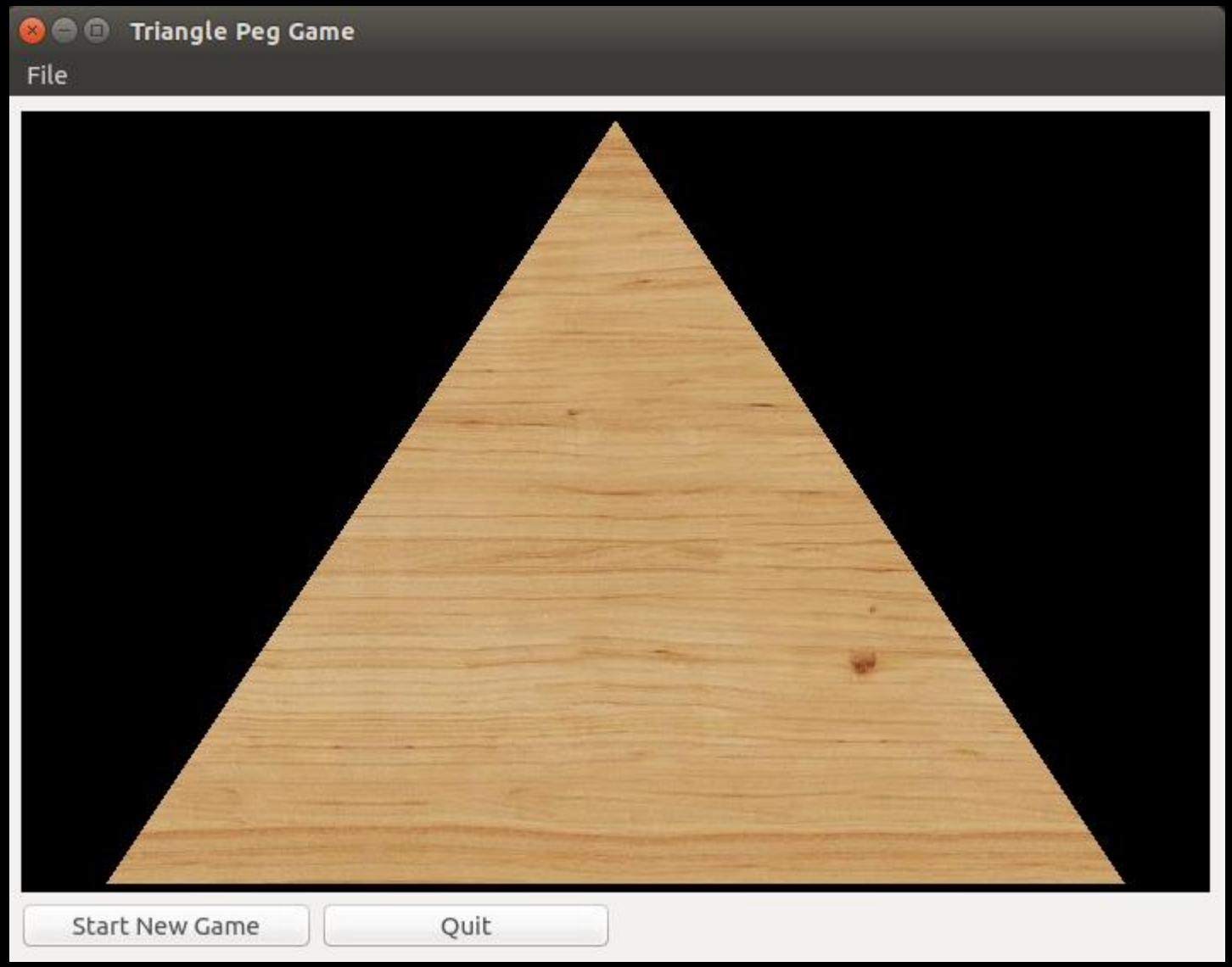
QPALETTE

Calling `palette()` on a `QWidget` object will return its currently associated `QPalette`. `QPalette` objects have a method `setColor()` which allows a `ColorRole` to be associated with a `QColor`.

`setAutoFillBackground` toggles the filling in of the background, which is transparent by default.

```
class PegBoard(QtWidgets.QWidget):
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        self.setFixedSize(700, 460)
        p = self.palette()
        p.setColor(self.backgroundRole(), QtGui.QColor(0, 0, 0, 255))
        self.setPalette(p)
        self.setAutoFillBackground(True)
```

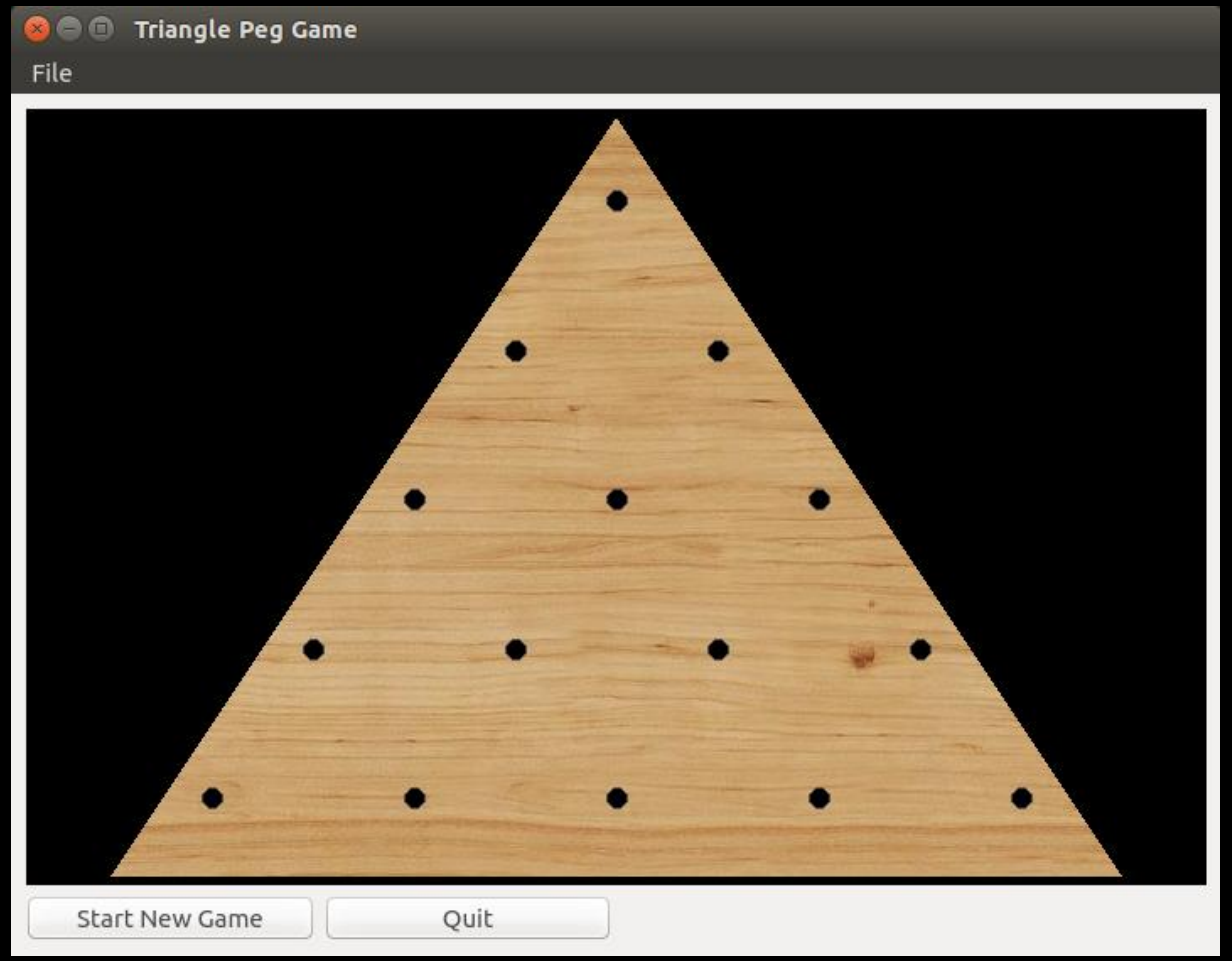
QPALETTE



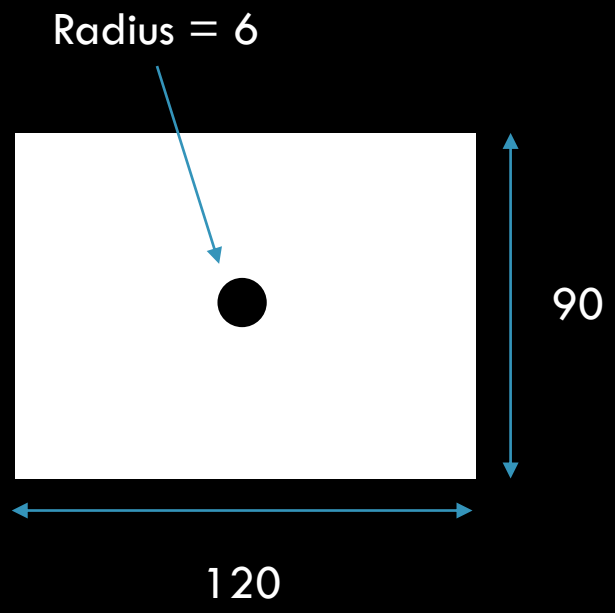
ADDING TO THE PEG BOARD

Next is to add the peg holes to our peg board.

We won't merely draw some dark circles on the board – we will create PegHole objects that can contain Peg objects. But let's just start with the PegHole definition.



PEG HOLES



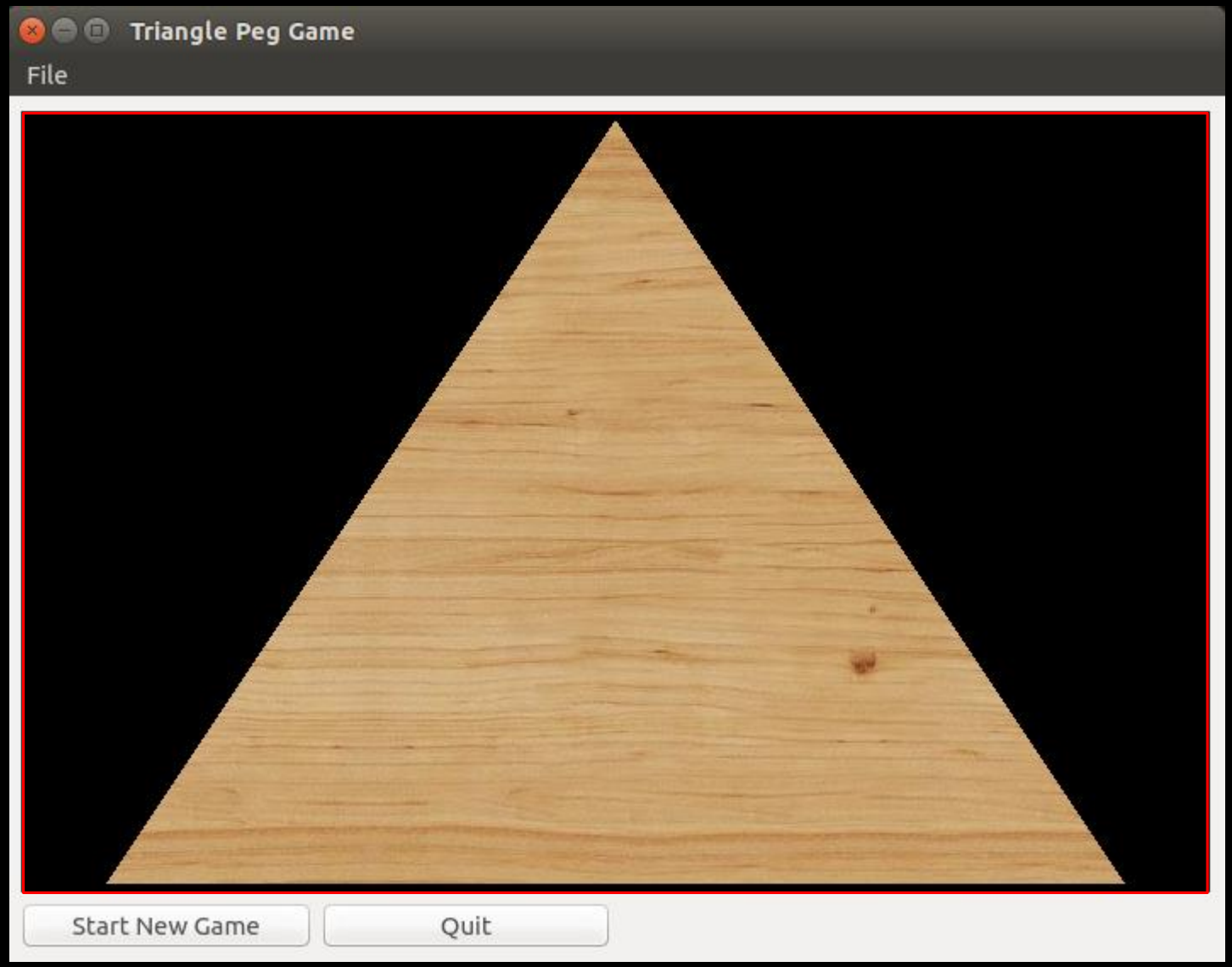
Note that our background is actually transparent!

```
class PegHole(QtWidgets.QWidget):
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        self.grid = QtWidgets.QGridLayout()
        self.setLayout(self.grid)
        self.peg = None

    def paintEvent(self, event):
        qp = QtGui.QPainter()
        qp.begin(self)
        brush = QtGui.QBrush(QtCore.Qt.SolidPattern)
        qp.setBrush(brush)
        qp.drawEllipse(QtCore.QPointF(60, 45), 6, 6)
        qp.end()

    def minimumSizeHint(self):
        return QtCore.QSize(120, 90)
```

PEG HOLES

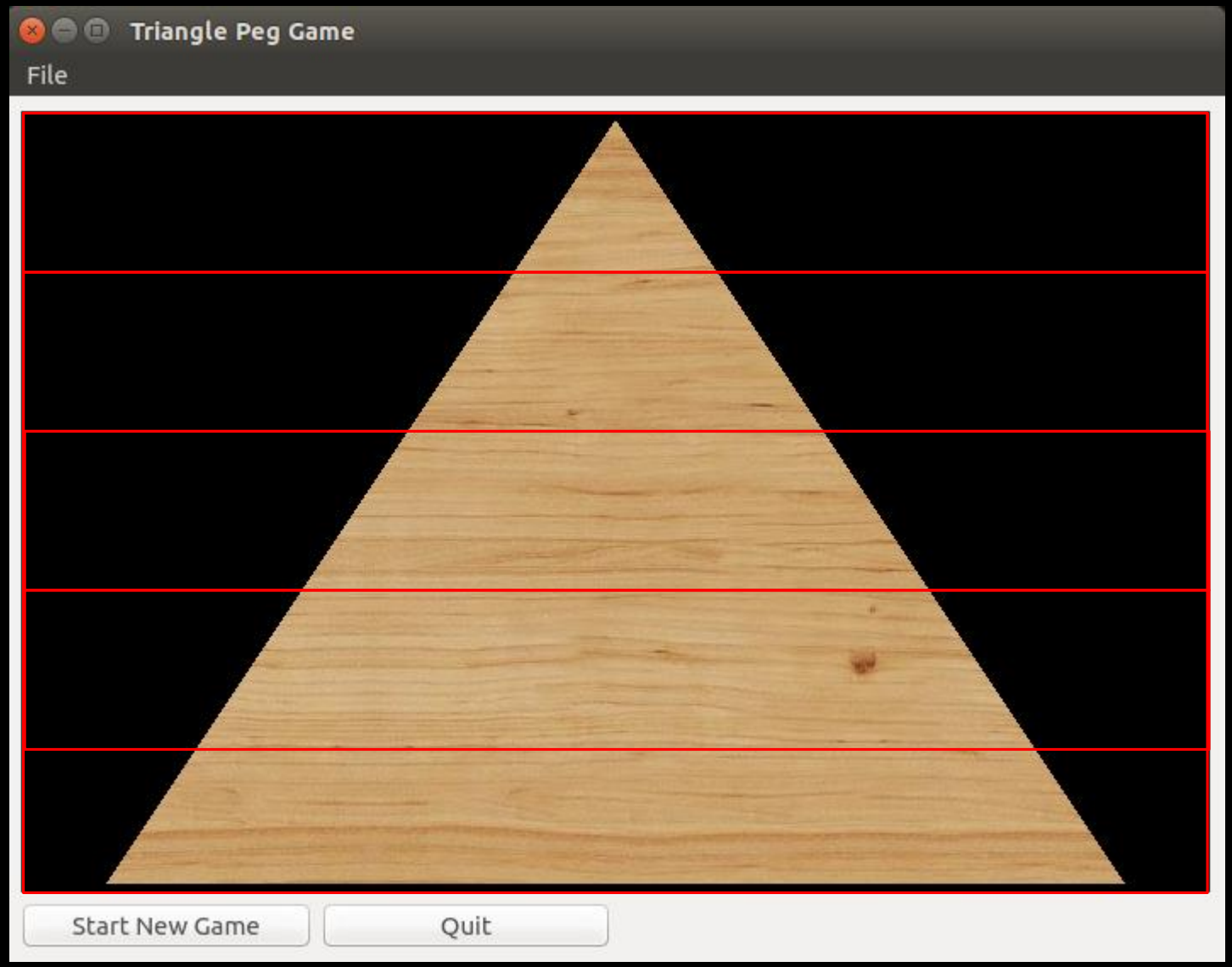
Now, how do we place these things on our board?

Start by creating a vertical box layout on the PegBoard.



PEG HOLES

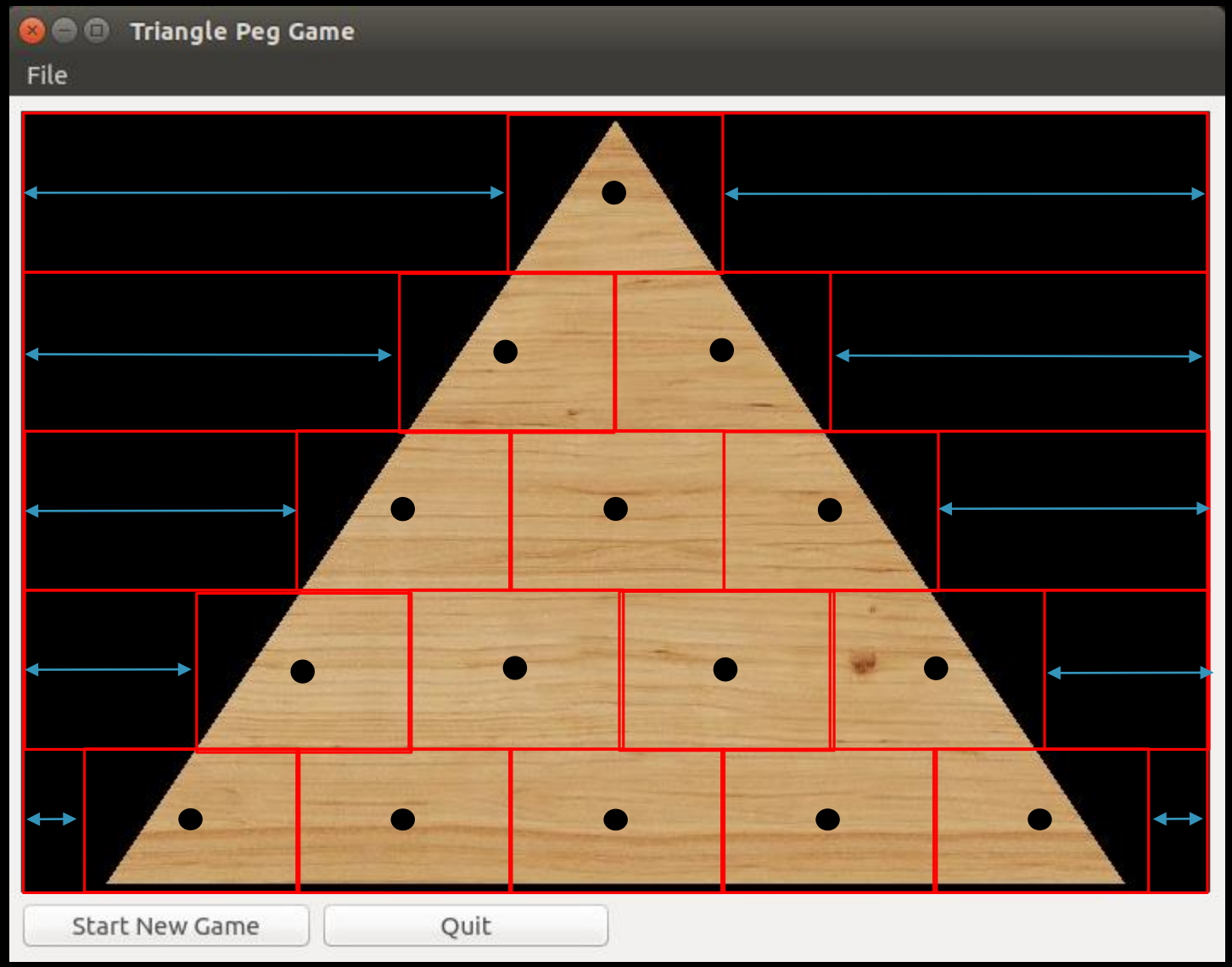
Now, add 5 horizontal box layouts to the vertical box layout.



PEG HOLES

In each horizontal box layout, add:

- a very stretchy spacer.
- the PegHole objects.
- another very stretchy spacer.



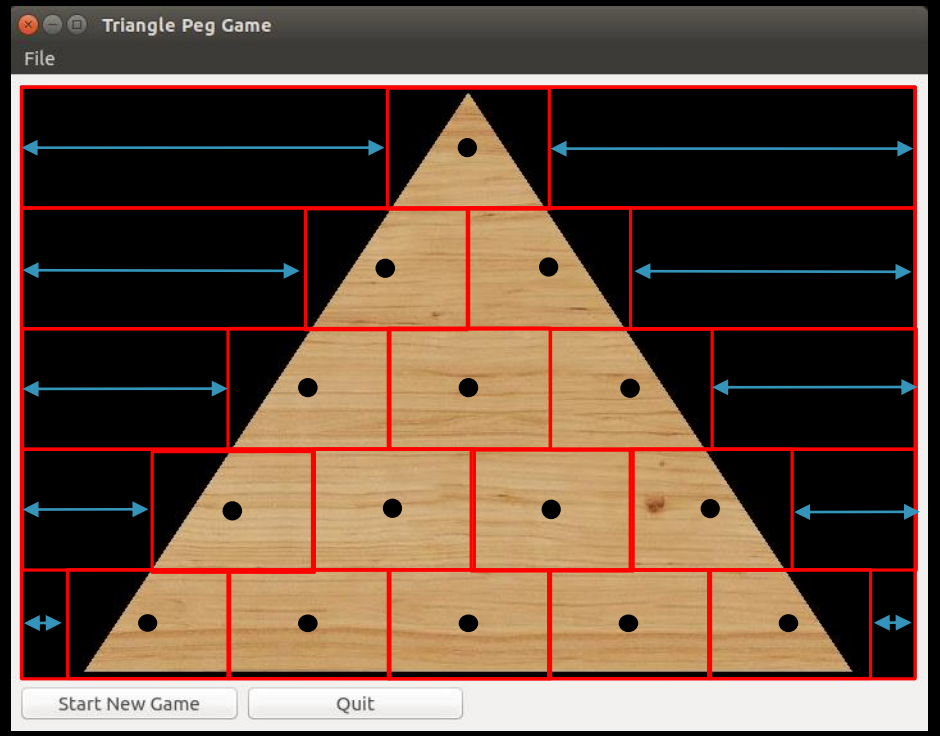
PEG HOLES


```
class PegBoard(QtWidgets.QWidget):
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        self.setFixedSize(700, 460)
        p = self.palette()
        p.setColor(self.backgroundRole(), QtGui.QColor(0, 0, 0, 255))
        self.setPalette(p)
        self.setAutoFillBackground(True)
        self.vbox = QtWidgets.QVBoxLayout()
        self.setLayout(self.vbox)
        self.vbox.setSpacing(0)
        self.place_holes()
```

Create a vertical box layout for the PegBoard instance. The `setSpacing` method allows us to manipulate the margins between widgets in the layout.

PEG HOLES

```
def place_holes(self):
    for row in range(0,5):
        rowLayout = QtWidgets.QHBoxLayout()
        self.vbox.addLayout(rowLayout)
        rowLayout.addStretch(1)
        for col in range(0,row+1):
            hole = PegHole(self)
            rowLayout.addWidget(hole, 0)
        rowLayout.addStretch(1)
```



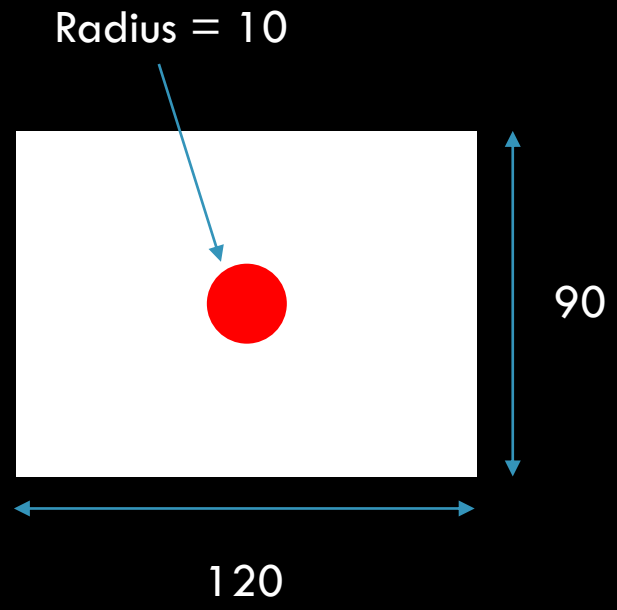
PEGS

So, the next step is to create a Peg object.

Keep in mind that PegHole objects are designed to house Peg objects and control their manipulation on the board.

```
class PegHole(QtWidgets.QWidget):
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        self.grid = QtWidgets.QGridLayout()
        self.setLayout(self.grid)
        self.peg = None
```

PEGS




```
class Peg(QtWidgets.QWidget):
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        self.resize(parent.size())

    def paintEvent(self, event):
        qp = QtGui.QPainter()
        qp.begin(self)
        brush = QtGui.QBrush(QtCore.Qt.SolidPattern)
        brush.setColor(QtCore.Qt.red)
        qp.setBrush(brush)
        qp.drawEllipse(QtCore.QPointF(self.width()/2,self.height()/2),10,10)
        qp.end()
```

Size is changed to match PegHole size.

PEGS

Now, we add a method to our PegHole class which allows us to associate Peg instances with PegHole objects.

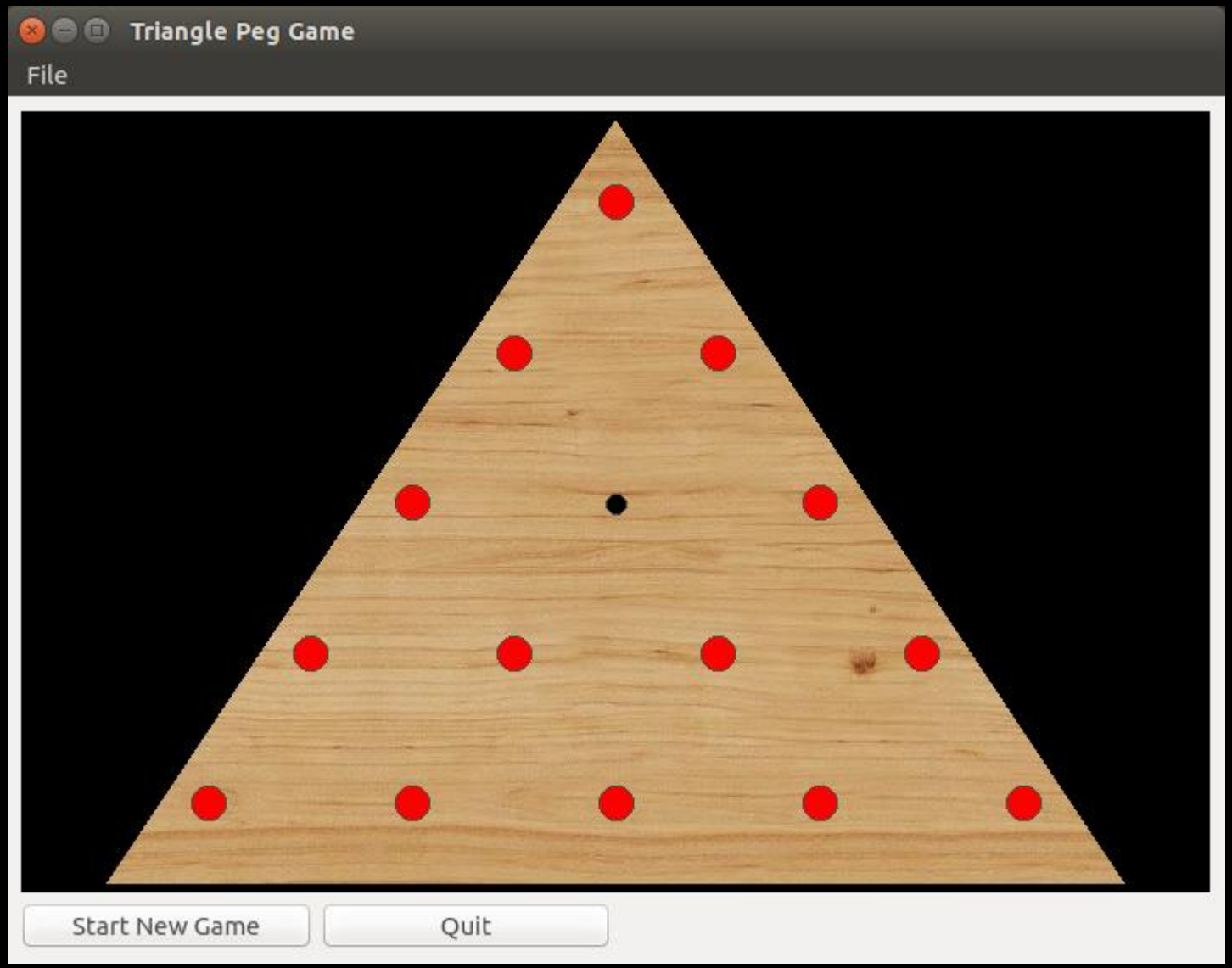
```
class PegHole(QtWidgets.QWidget):
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        self.grid = QtWidgets.QGridLayout()
        self.setLayout(self.grid)
        self.peg = None
    ...
    def addPeg(self):
        self.peg = Peg(self)
        self.grid.addWidget(self.peg)
```

PEGS

And finally, we extend our PegBoard.place_holes method to associate Pegs with PegHoles.

```
def place_holes(self):
    for row in range(0,5):
        rowLayout = QtWidgets.QHBoxLayout()
        self.vbox.addLayout(rowLayout)
        rowLayout.addStretch(1)
        for col in range(0, row+1):
            hole = PegHole(self)
            rowLayout.addWidget(hole, 0)
            if (row, col) != (2,1):
                hole.addPeg()
        rowLayout.addStretch(1)
```

PEGS



PEG MOVEMENT

Now, the hard part. How do we describe the mechanics of moving the pegs? How does a user interact with the game board?

We aren't going to concern ourselves with keeping score or enforcing legal movements yet. We just want to figure out *how* to move the pegs around.

The most natural mechanism is probably drag-and-drop. We should be able to "pick up" pegs, move them with the mouse, and "drop" them into an empty hole.

DRAG AND DROP

Drag and drop, a commonly implemented GUI interaction mechanism, centers around the `QDrag` object.

The `QDrag` class supports MIME-based drag and drop transfer.

MIME (Multiple Internet Mail Extensions) is a system originally designed to package arbitrary attachments to email. But this system is also commonly used to package arbitrary data with a `QDrag` object so we can send it anywhere in the application.

DRAG AND DROP

There are two parts to a drag and drop action to be considered:

- The drag source.

- The drop target.

Let’s start with our drag source, the PegHole object.

DRAG SOURCE

A drag source is any object that creates a `QDrag` object when we interact with it. What does it mean to "interact"? Well, we could define it a couple of ways – here are some methods we could override to start a drag:

- `mousePressEvent(self, event)`
  - Called when the mouse is pressed down.
- `mouseReleaseEvent(self, event)`
  - Called when the mouse is released.
- `mouseMoveEvent(self, event)`
  - Called when the mouse is pressed down and moved.

DRAG SOURCE

```
class PegHole(QtWidgets.QWidget):
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        self.grid = QtWidgets.QGridLayout()
        self.setLayout(self.grid)
        self.peg = None

    def mousePressEvent(self, event):
        if not self.peg:
            QtWidgets.QWidget.mousePressEvent(self, event)
            return

        drag = QtGui.QDrag(self)
        drag.setMimeData(QtCore.QMimeData())
        dropAction = drag.exec_(QtCore.Qt.MoveAction)
```

Now, when we click on a PegHole object, we'll start the process of dragging… but only if there is a Peg object associated!

DRAG SOURCE

```
class PegHole(QtWidgets.QWidget):
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        self.grid = QtWidgets.QGridLayout()
        self.setLayout(self.grid)
        self.peg = None

    def mousePressEvent(self, event):
        if not self.peg:
            QtWidgets.QWidget.mousePressEvent(self, event)
            return

        drag = QtGui.QDrag(self)
        drag.setMimeData(QtCore.QMimeData())
        dropAction = drag.exec_(QtCore.Qt.MoveAction)
```

If there is no associated Peg object, simply call the basic mousePressEvent method and return.

DRAG SOURCE

```
class PegHole(QtWidgets.QWidget):
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        self.grid = QtWidgets.QGridLayout()
        self.setLayout(self.grid)
        self.peg = None

    def mousePressEvent(self, event):
        if not self.peg:
            QtWidgets.QWidget.mousePressEvent(self, event)
            return

        drag = QtGui.QDrag(self)
        drag.setMimeData(QtCore.QMimeData())
        dropAction = drag.exec_(QtCore.Qt.MoveAction)
```

If there is a Peg object, then start a drag by instantiating a new QDrag object.

DRAG SOURCE

```
class PegHole(QtWidgets.QWidget):
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        self.grid = QtWidgets.QGridLayout()
        self.setLayout(self.grid)
        self.peg = None

    def mousePressEvent(self, event):
        if not self.peg:
            QtWidgets.QWidget.mousePressEvent(self, event)
            return

        drag = QtGui.QDrag(self)
        drag.setMimeData(QtCore.QMimeData())
        dropAction = drag.exec_(QtCore.Qt.MoveAction)
```

Associate a QMimeData object. This is required, but your QMimeData object is not required to hold anything significant.

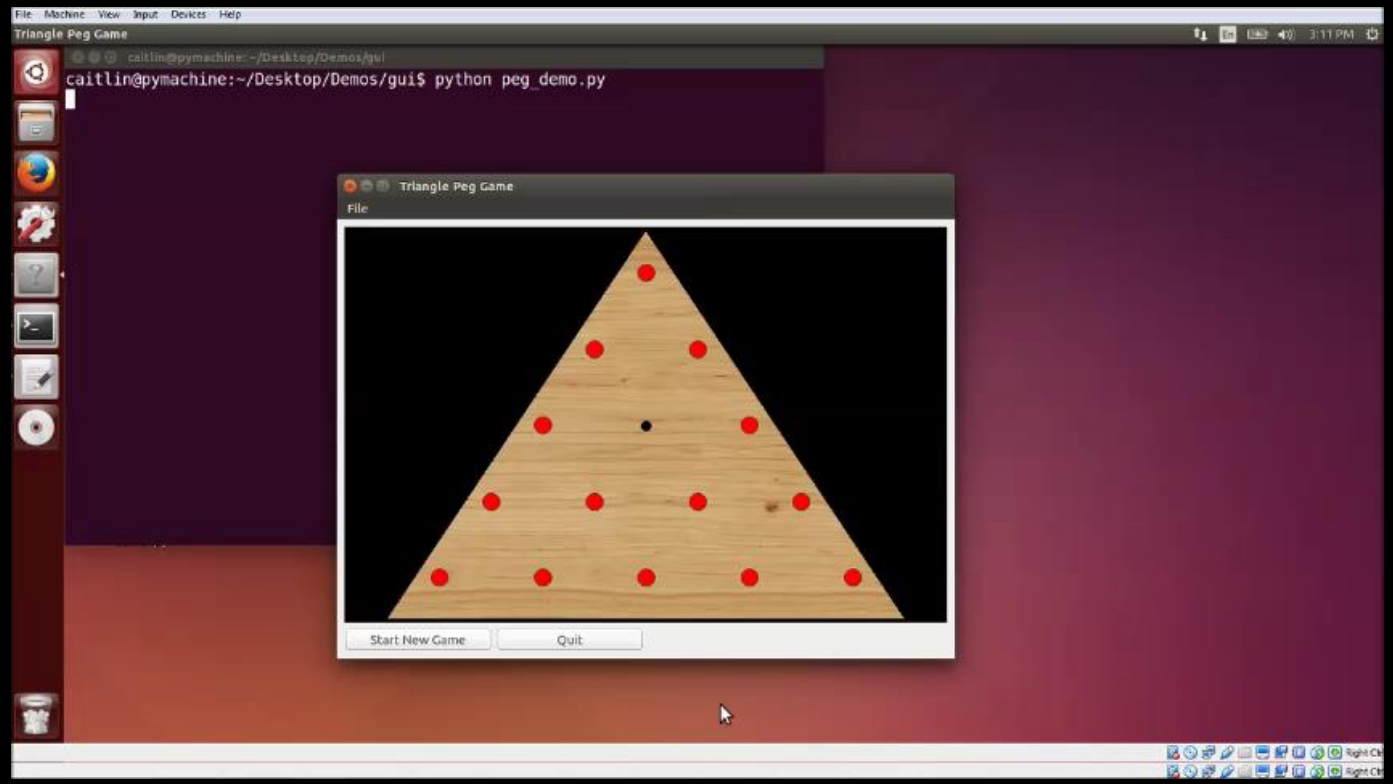
DRAG SOURCE

```
class PegHole(QtWidgets.QWidget):
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        self.grid = QtWidgets.QGridLayout()
        self.setLayout(self.grid)
        self.peg = None

    def mousePressEvent(self, event):
        if not self.peg:
            QtWidgets.QWidget.mousePressEvent(self, event)
            return

        drag = QtGui.QDrag(self)
        drag.setMimeData(QtCore.QMimeData())
        dropAction = drag.exec_(QtCore.Qt.MoveAction)
```

Call the exec_() method on the QDrag object to start the drag and drop operation. This method returns a value indicating the drop action taken.

DRAG SOURCE

So, what happens now?



Well, not much. We haven't defined what it looks like to drag the Peg and we haven't defined any drop targets. The only thing we can see for sure is that a drag event starts when the PegHole has an associated Peg, otherwise nothing happens.

DROP TARGET

Now, let's define a drop target. Since a PegHole is conceptually both the source of the Peg as well as the eventual destination of the Peg, we will also use a PegHole as our drop target. Drop targets have the following properties:

- Sets `self.setAcceptDrops(True)`.
- Implements the `dragEnterEvent(self, event)` method.
  - Called when a drag enters the target's area. Typically defined to either accept or reject the drag.
- Implements the `dropEvent(self, event)` method.
  - Called when the user releases the mouse button, "dropping" on the target.

DROP TARGET

Our PegHole now accepts drop requests

Allow the drag to enter the widget space and set the drop action to be whatever the QDrag object proposes (i.e. MoveAction).

```
class PegHole(QtWidgets.QWidget):
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        self.setAcceptDrops(True)
        self.grid = QtWidgets.QGridLayout()
        self.setLayout(self.grid)
        self.peg = None

    def dragEnterEvent(self, event):
        event.acceptProposedAction()

    def dropEvent(self, event):
        if not self.peg:
            self.addPeg()
            event.accept()
        else:
            event.ignore()
```

DROP TARGET

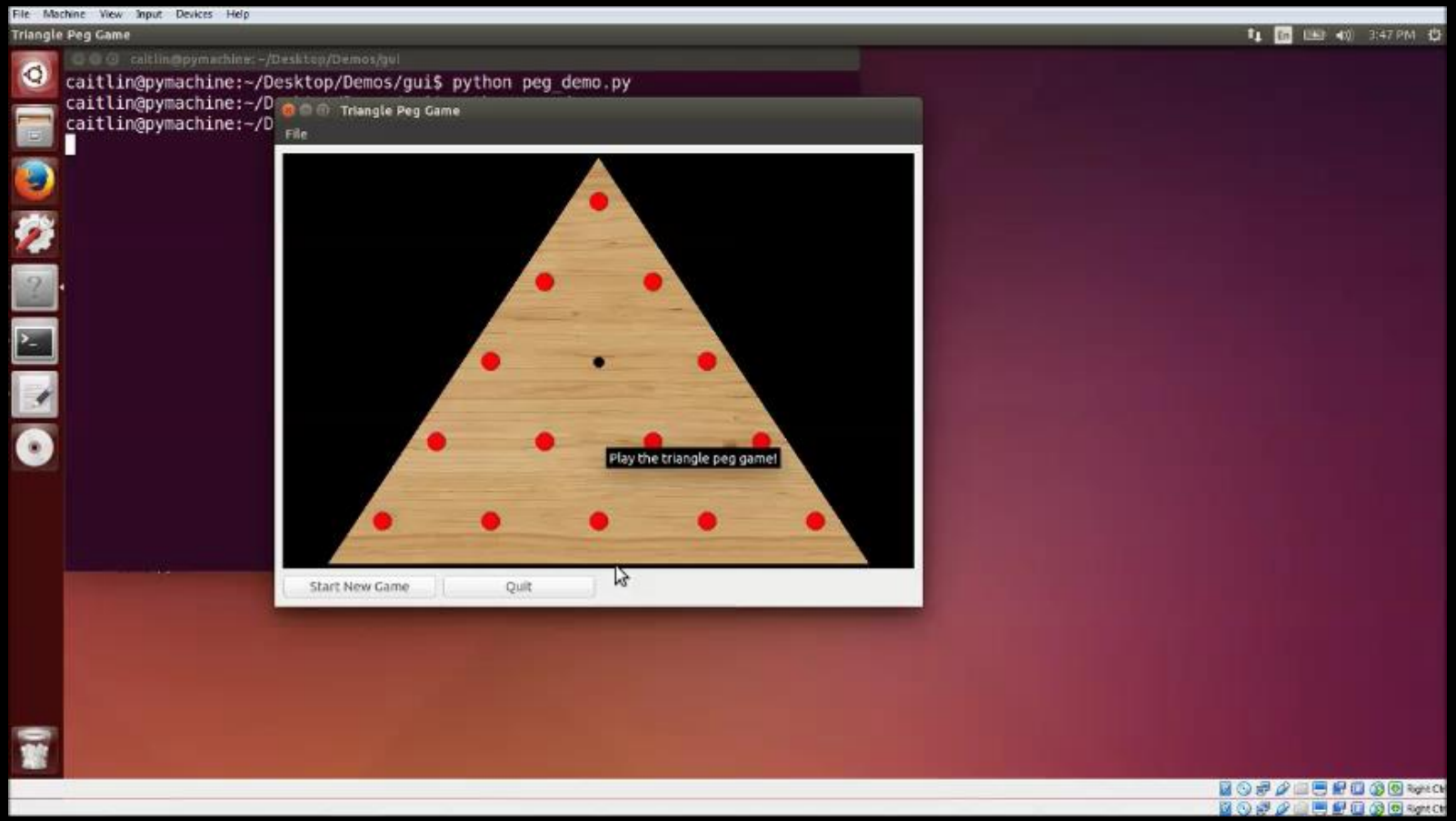
```
class PegHole(QtWidgets.QWidget):
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        self.setAcceptDrops(True)
        self.grid = QtWidgets.QGridLayout()
        self.setLayout(self.grid)
        self.peg = None

    def dragEnterEvent(self, event):
            event.acceptProposedAction()

    def dropEvent(self, event):
        if not self.peg:
            self.addPeg()
            event.accept()
        else:
            event.ignore()
```

Accept the drop and add a new Peg object if there is not already a Peg object associated.

DROP TARGET

We can at least force a new Peg object to appear.

But that does not delete the old Peg object nor describe what it means to drag from PegHole instance to PegHole instance.



DRAG SOURCE

Back to the drag source….

```
def mousePressEvent(self, event):
    if not self.peg:
        QtWidgets.QWidget.mousePressEvent(self, event)
        return
```

Hide the peg temporarily →

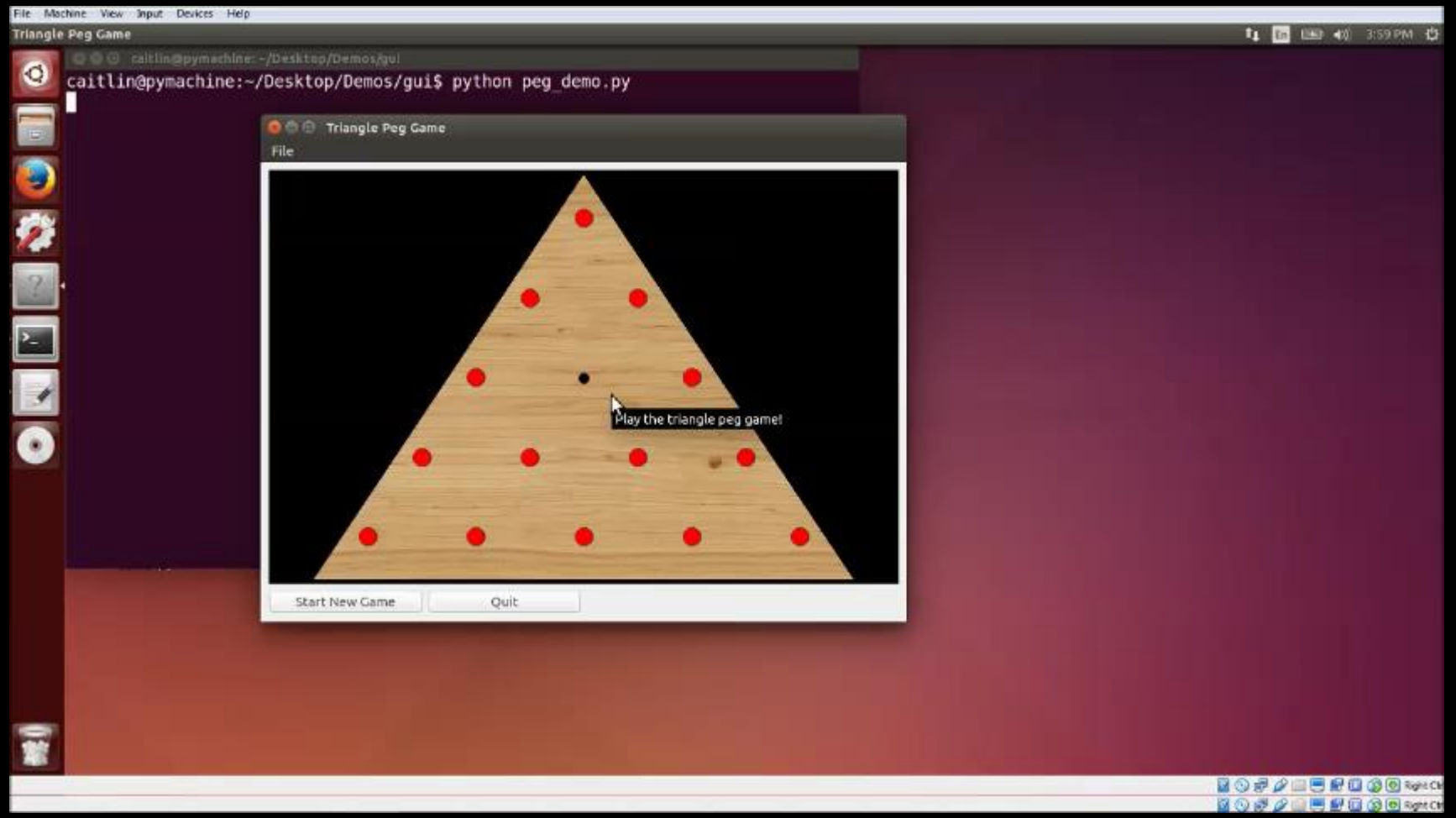
```
    self.peg.hide()
    drag = QtGui.QDrag(self)
    drag.setMimeData(QtCore.QMimeData())
    dropAction = drag.exec_(QtCore.Qt.MoveAction)
```

When “drop” happens, check whether drop was accepted or not. If so, delete Peg. Otherwise, show the Peg again.

```
    if dropAction:
        del(self.peg)
        self.peg = None
    else:
        self.peg.show()
```

DRAG AND DROP

Ok, once again!



DRAG AND DROP

But wouldn't it be so nice to see the Peg move?

We'll create an icon of the Peg and associate it with the `QDrag` object.

```
def mousePressEvent(self, event):
    if not self.peg:
        QtWidgets.QWidget.mousePressEvent(self, event)
        return

    self.peg.hide()
    drag = QtGui.QDrag(self)
    drag.setMimeData(QtCore.QMimeData())
    drag.setPixmap(self.peg_icon)
    drag.setHotSpot(self.peg_icon.rect().topLeft())
    dropAction = drag.exec_(QtCore.Qt.MoveAction)

    if dropAction:
        del(self.peg)
        self.peg = None
    else:
        self.peg.show()
```

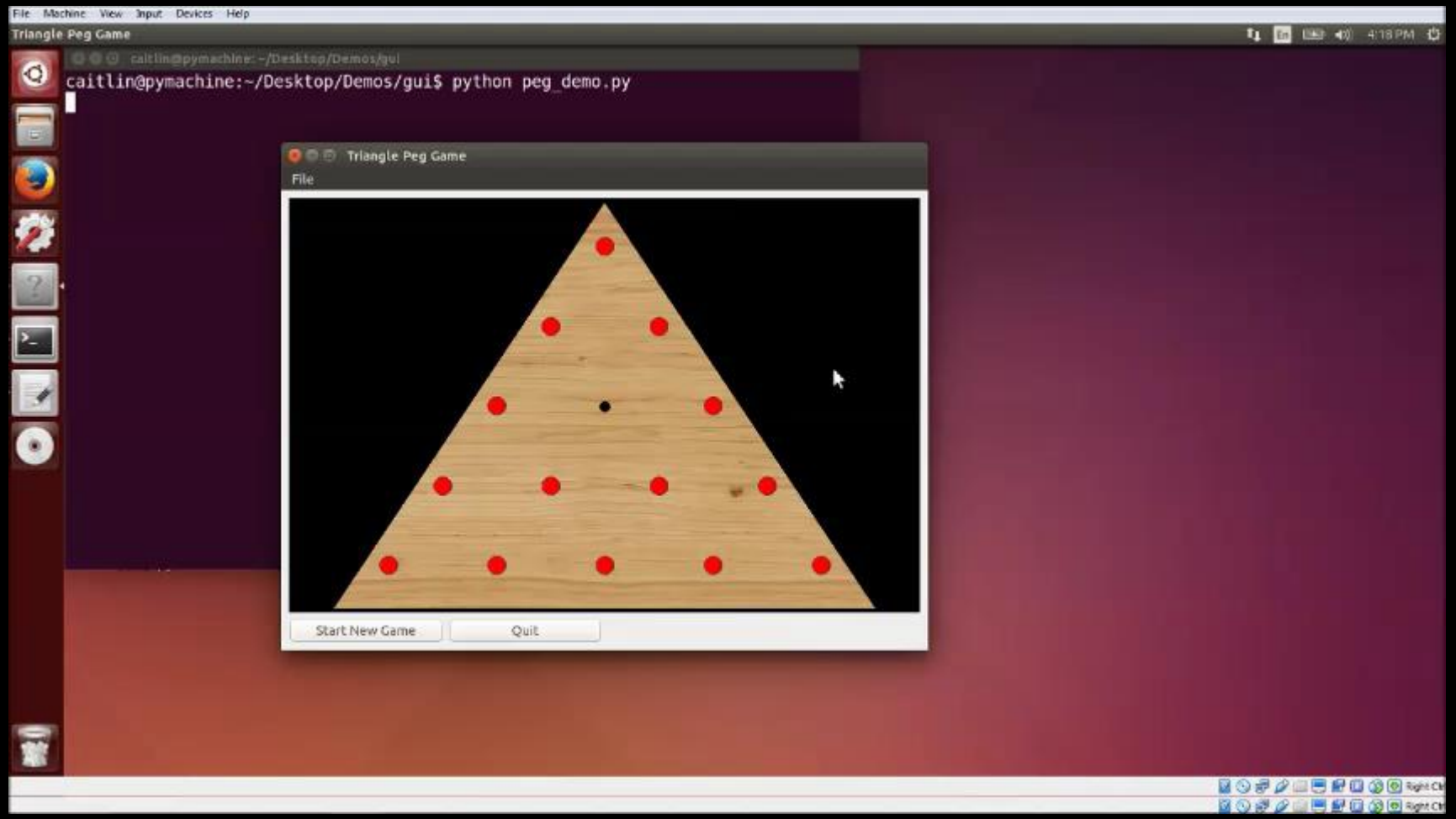
DRAG AND DROP

Now we have an icon that represents what a Peg looks like visually.

This is the visual we'll associate with the drag so that our user feels like they're moving the object in space.

```
class PegHole(QtWidgets.QWidget):
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        self.setAcceptDrops(True)
        self.grid = QtWidgets.QGridLayout()
        self.setLayout(self.grid)
        self.peg = None
        self.create_icon()

    def create_icon(self):
        self.peg_icon = QtGui.QPixmap(22, 22)
        self.peg_icon.fill(QtCore.Qt.transparent)
        qp = QtGui.QPainter()
        qp.begin(self.peg_icon)
        brush = QtGui.QBrush(QtCore.Qt.SolidPattern)
        brush.setColor(QtCore.Qt.red)
        qp.setBrush(brush)
        qp.drawEllipse(0, 0, 20, 20)
        qp.end()
```

DRAG AND DROP



ENFORCING RULES

So, we have the mechanics of movement. Now, all we need to do is enforce some rules. This comes down to two checks:

- Make sure destination is valid.
- Make sure “hopped” hole contains a Peg to be removed.

We’re going to use the MIME mechanism to send information about our drag source to our drop target – this will help us decide whether to accept or not.

CHECKING DESTINATION

We first extend the PegHole class definition to accept its position as an argument. We pass in the row and column where it is placed on the board.

Furthermore, the PegBoard now maintains a list of its child widgets and an associated counter

```
class PegBoard(QtWidgets.QWidget):
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        self.holes = []
        ...
        self.place_holes()
        self.peg_count = len(self.holes)- 1

    def place_holes(self):
        for row in range(0,5):
            row_list = []
            ...
            for col in range(0,row+1):
                hole = PegHole(self, row, col)
                ...
            rowLayout.addStretch(1)
            self.holes.append(row_list[:])
```

CHECKING DESTINATION

```
def mousePressEvent(self, event):
    if not self.peg:
        QtWidgets.QWidget.mousePressEvent(self, event)
        return

    self.peg_icon = self.create_icon()
    self.peg.hide()
    drag = QtGui.QDrag(self)
    data = QtCore.QMimeData()
    data.setText(pickle.dumps((self.row, self.col)))
    drag.setMimeData(data)
    drag.setPixmap(self.peg_icon)
    drag.setHotSpot(self.peg_icon.rect().topLeft())
    dropAction = drag.exec_(QtCore.Qt.MoveAction)

    ...
```

The `QMimeData` object associated with the `QDrag` object will now house some information – a pickled string holding the source's coordinates.

CHECKING DESTINATION

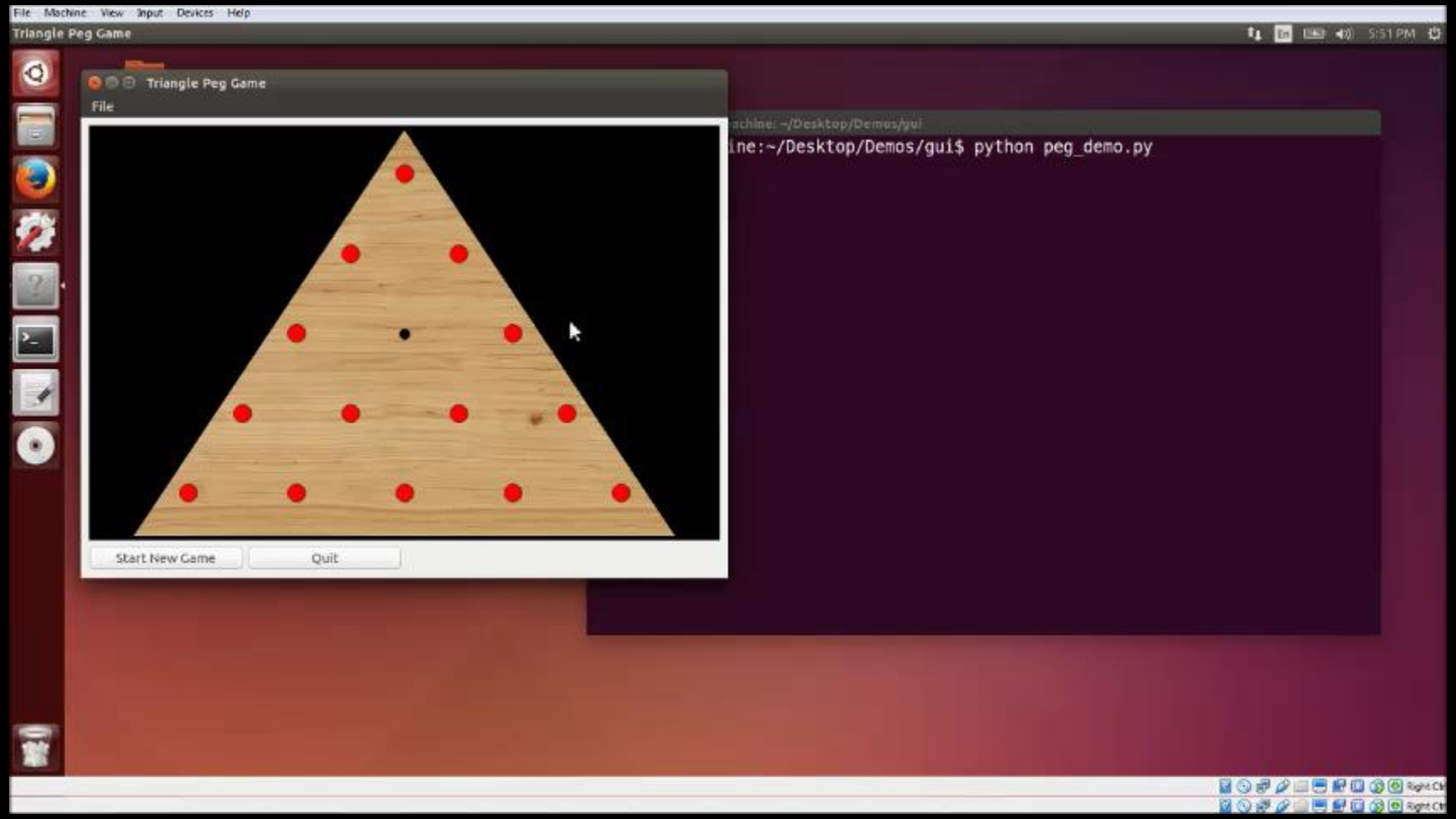
Only accept the drop request if the destination does not have a Peg *and* the source of the drag is valid.

```
def dropEvent(self, event):
    if not self.peg:
        row, col = pickle.loads(event.mimeData().text())
        if(self.check_valid(row, col)):
            self.addPeg()
            event.accept()
        else:
            event.ignore()
    else:
        event.ignore()
```

ENFORCING RULES

```
def check_valid(self, row, col):
    hopped = None
    # Calculate coordinates of skipped peg
    # and store as hopped
    if not hopped:
        return False

    if(self.parent.holes[hopped[0]][hopped[1]].peg):
        self.parent.holes[hopped[0]][hopped[1]].deletePeg()
        self.parent.peg_count -= 1
        return True
    else:
        return False
```

ENFORCING RULES



FURTHER READING

Check out this link for a little tutorial on custom widgets.

Check out this link for a small tetris game demo.

Very thorough PyQt source here.