Timers and Animation

Sometimes you want to make events happen on their own – without any user input. Maybe you want a function to run at a specific time delay after the user pushes a button, or a function to run again and again at regular intervals. Unfortunately the event loop of a GUI application is a Diva, hogging all your app's computational cycles. To get around this bottleneck you need to go multi-threaded.

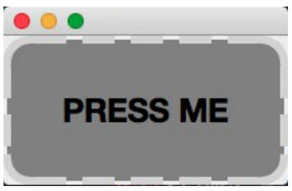
Most GUI toolkits provide a fast and easy interface for making special threads called ***Timers***. In Qt, you have the class `QTimer`. It is found in the `QtCore` module (`PyQt5.QtCore.QTimer`).

QTimer has a method `start(ms)` that you call when you want your timer thread to start running – `ms` is the time in milliseconds before the timer emits a special signal called `timeout` that you can hook up to your own functions just like you would with any other Qt signal. The main difference is that now your function gets to run in its own thread, totally bypassing the computational fascism of the event loop.

One Shot Timer

There are two ways a `QTimer` can behave: ***repeating*** and ***one-shot***. We'll look at the *one-shot* first, and then the more general *repeating* type. The *one-shot* behavior allows you to schedule a function to be run some time in the future. When you call `start()` your QTimer will count down that many milliseconds and emit its `timeout` signal. This will only happen once.

Let's look at an example. This program uses a one-shot timer to make a toggle button that "unchecks" itself 3 seconds after you've pressed it, like a GUI version of Claude Shannon's Ultimate Machine.



... leave me alone ...

Useless-Machine.py

```
from PyQt5.QtWidgets import *
from PyQt5.QtCore import QTimer, Qt
import sys

class MyWindow(QWidget):
    buttonStyle = """
        QPushButton {
            font-size: 24px;
            font-weight: bold;
            background-color: grey;
            border-radius: 20px;
            border: 6px dashed lightgrey;
            max-width: 200px;
            max-height: 100px;
        }
        QPushButton:checked {
            background-color: #a7e1e4;
        }
    """

    def __init__(self):
        super(MyWindow, self).__init__()
        self.timer = QTimer(self)
        self.timer.setSingleShot(True) # give the timer one-shot behavior
        self.timer.timeout.connect(self.unclick) # connect the timeout signal to our unclick method
        self.makeWindow()

    def makeWindow(self):
        self.but = QPushButton("PRESS ME", self)
        self.but.setGeometry(self.rect())
        self.but.setCheckable(True) # make this button behave like a checkbox
        self.but.clicked.connect(self.turnOn)
        self.but.setStyleSheet(self.buttonStyle)
```

```
34         self.setWindowFlags(Qt.FramelessWindowHint) # needed to get a transparent window in Windows
35         self.setAttribute(Qt.WA_TranslucentBackground)
36         self.show()
37 
38     def turnOn(self):
39         self.but.setText("PRESSED")
40         self.timer.start(3000) # 3 second delay
41 
43     def unclick(self):
44         self.but.setText("PRESS ME")
45         self.but.setChecked(False)
46 
47 if __name__ == '__main__':
48     app = QApplication(sys.argv)
49     mywin = MyWindow()
50     sys.exit(app.exec_())
```

A few things to note in this program...

Line 23-25 is where the `QTimer` is created and stored in an instance variable. Then ***line 24*** configures it as a one-shot timer by giving a `True` argument to `setSingleShot()`. ***Line 25*** connects the timer's `timeout` signal to our `unclick` method.

Lines 29-33 create the button widget, resizes it, applies some visual styles and configures it to behave like a checkbox. Most importantly, the button's `clicked` signal is connected to our `turnOn` method.

`turnOn()`, defined on ***lines 38-40*** is where the timer is activated, using `start()`, with a wait time of 3000 milliseconds (3 seconds) before calling `unclick`. The `unclick` method's only job is to change the display text of the button and uncheck it.

Repeating Timer

The more common use of timers is to trigger a function repeatedly, for example to check on a time consuming task like processing a video or waiting for messages to come in on a chat program.

Repeating timers also allow you to create animations & time-based effects, by calling an animation function repeatedly at a steady frame rate. Creating a repeating timer is almost identical in syntax to how we created a single-shot timer in the last example. Except now the millisecond value given to `start()` will be the duration between subsequent timeout signals.

Here's an example. This program creates a custom window called GravityWindow that uses a repeating timer to modify its own y position repeatedly so that it accellerates to the bottom of the screen and then bounces a few times before coming to a rest. The animation is done using a velocity and acceleration physics simulation commonly used in 2D games.

This example also shows how you can add sound to your program by using Qt's sound file player, `QSound`.

`Gravity-Windows.py`

```
from PyQt5.QtWidgets import QApplication, qApp, QWidget
from PyQt5.QtCore import QTimer, QUrl
from PyQt5.QtMultimedia import QSound
import sys

class GravityWindow(QWidget):
    def __init__(self, xpos, ypos, title):
        super(GravityWindow, self).__init__()
        self.acceleration = 5.0
        self.velocity = 0.0
        self.screenHeight = qApp.desktop().availableGeometry().height() # height of the screen
        self.sound = QSound("sounds/Sosumi.wav")
        self.makeWin(xpos, ypos, title)

    def makeWin(self, xpos, ypos, title):
        self.setGeometry(xpos, ypos, 200, 100)
        self.setWindowTitle(title)
        self.timer = QTimer(self)
        self.timer.timeout.connect(self.animateFrame)
        self.timer.start(1000 / 12) # animate at 12 frames per second

```

```
22      def animateFrame(self):
23          xpos = self.pos().x()
24          ypos = self.pos().y() + self.velocity # move the ypos a little bit
25          if (ypos + self.height()) > self.screenHeight:   # the window has hit the bottom of the screen
26              ypos = self.screenHeight - self.height()
27              if abs(self.velocity) <= self.acceleration:  # if velocity is very small set it to 0.0
28                  self.velocity = 0.0
29              else:
30                  self.sound.play()
31                  self.velocity *= -0.5 # change direction of movement and half the velocity
32          self.move(xpos, ypos)
33          self.velocity += self.acceleration
34
35
36  if __name__ == '__main__':
37      app = QApplication(sys.argv)
38      app.setStyleSheet("""
39              GravityWindow {
40                  background-color: qlineargradient(x1:0,y1:0,x2:0,y2:1, stop: 0 #ffffff, stop: 1 #333333);
41              }
43          """)
44      win1 = GravityWindow(50, 500, "Humpty")
45      win2 = GravityWindow(400, 0, "Dumpty")
46      win1.show()
47      win2.show()
48      sys.exit(app.exec_())
```

A few things to note in this program...

This program is short and sweet, but there are a few things worth noting. The init method sets up the initial velocity and acceleration values of the window, the higher the acceleration, the more quickly the speed of the window will increase. This code also loads the "bounce" sound into a QSound, and stores the height of the screen for later use. Note on line 11 how I get the dimensions of the user's screen with `qApp.desktop().availableGeometry()`

```
self.screenHeight = qApp.desktop().availableGeometry().height() # height of the screen
```

`qApp` is a useful global variable in the `PyQt5.QWidgets` module that gives you a reference to the object that represents your Qt application (the one created down in ***line 37***). This object is very useful for a number of things, including setting global stylesheets and getting information about the user's desktop and operating system.

Lines 18-20 create the repeating timer that will animate the window. It's `timeout` signal is connected to our `animateFrame` function, and it's set to trigger every 1000 / 12 milliseconds (1 second divided by 12 equals 12 frames per second).

Lines 22-33 is where the new position of the window is calculated for each frame of the animation, based on the current position, velocity and acceleration. The basic idea of this algorithm is:

```
 # 1. Set the y position of the window to be the current position plus the current velocity
ypos = self.pos().y() + self.velocity
 # 2. Check if the new y position would put the window beyond the bottom of the screen
if (ypos + self.height()) > self.screenHeight:
     # 2.1 If true, change the new y position to be exactly at the bottom of the screen
    ypos = self.screenHeight - self.height()
     # 2.2 Check if the absolute value of the velocity is less than the acceleration
    if abs(self.velocity) <= self.acceleration:
         # 2.2.1 If true, stop the window from bouncing by setting its velocity to 0
        self.velocity = 0.0
    else:
         # 2.2.2 If false, make the window bounce. Play the bouncing sound and multiply the velocity
         # by -0.5. This reverses the direction of movement and reduces the speed of movement
        self.sound.play()
        self.velocity *= -0.5 # change direction of movement and half the velocity
# 3. Move the window to the new y position
self.move(xpos, ypos)
 # 4. Increase the velocity by the acceleration
self.velocity += self.acceleration
```

The `timerEvent()` Method

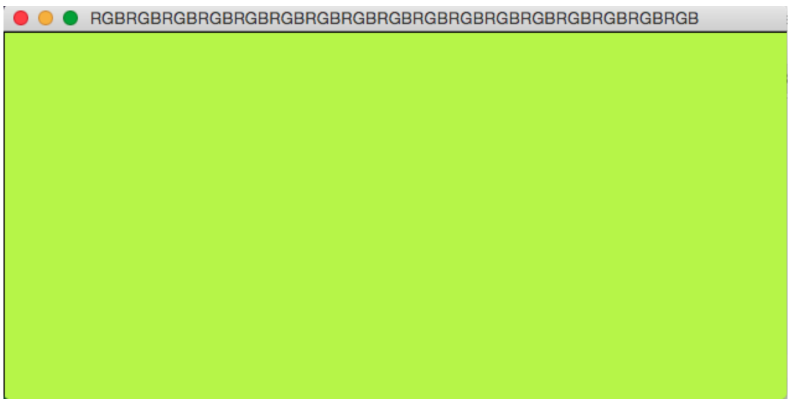
All QObject subclasses (including QWidget) provides built-in mechanisms for making repeating timers. This mechanism is often used as a shortcut when a timer should be associated with a specific object or widget, the case of an animated widget is a good example.

To use the built-in timers of QObject, you use the following three methods:

`startTimer(ms)`	Starts a new timer with timeout interval `ms`, returns a unique timer id number
`killTimer(id)`	Kills the timer with the given id number
`timerEvent()`	Method called on an object whenever one of its built-in timers emits a timeout signal

Let's take a look at an example that uses the built-in timers of a custom widget to control the speed of an animation animation. This program creates a widget whose canvas is filled in with a solid color that is animated to create a glowing or flickering effect, depending on how fast the color is changing. The color is updated at a specific rate specified by a repeating timer. The widget has a method, `setSpeed()`, that can be used to change the speed by killing the current timer and starting a new one with a different interval time.

By clicking and dragging inside the widget, the user can change the flicker speed. The user can also toggle between full screen mode and windowed mode using the ***F*** key.



A flickering color box, use the F key to go full screen, click and drag to change the flicker rate

GlowBox.py

```python
from PyQt5.QtWidgets import *
from PyQt5.QtCore import Qt, QSize
from PyQt5.QtGui import QColor, QBrush, QPainter, QKeySequence
from random import randint
import sys

class GlowBox(QWidget):
    def __init__(self, parent=None):
        super(GlowBox, self).__init__(parent)
        self.color = [randint(0,255), randint(0,255), randint(0,255)] # initial red green blue values
        self.changeBy = [-7, 5, 12] # red green blue change per frame
        self.frameDelayMillis = 1000 / 60  # 60 frames per second, initial interval time
        self.timerID = self.startTimer(self.frameDelayMillis) # start the built-in timer

    def paintEvent(self, event):
        qp = QPainter(self) # QPainter has all the drawing commands
        rect = self.rect()
        color = QColor(self.color[0], self.color[1], self.color[2])
        brush = QBrush(color, Qt.SolidPattern)
        qp.setBrush(brush)
        qp.setPen(Qt.NoPen)
        qp.drawRect(rect)

```

```
24     def timerEvent(self, event):
25         self.updateColor()
26         self.update() # force the widget to redraw itself
27 
28     def updateColor(self):
29         for i in [0,1,2]:
30             self.color[i] = self.color[i] + self.changeBy[i]
31             if self.color[i] < 0:
32                 self.color[i] = 0
33                 self.changeBy[i] *= -1 # reverse direction
34             elif self.color[i] > 255:
35                 self.color[i] = 255
36                 self.changeBy[i] *= -1 # reverse direction
37 
38     def setSpeed(self, frameDelay):
39         self.killTimer(self.timerID)
40         self.frameDelayMillis = frameDelay
41         self.timerID = self.startTimer(self.frameDelayMillis)
43 
44 class MyWin(QWidget):
45     def __init__(self):
46         super(MyWin, self).__init__()
47         self.makeWindow()
48 
49     def makeWindow(self):
50         self.setWindowTitle("RGB")
51         h = qApp.desktop().screenGeometry().height() - 100
52         w = qApp.desktop().screenGeometry().width() - 100
53         self.speedFactor = w / 50
54         self.setGeometry(20, 20, w, h)
55         self.box = GlowBox(self)
56         self.box.resize(w,h)
57         self.show()
58 
59     def resizeEvent(self, event):
60         self.box.resize(self.rect().width(), self.rect().height())
61 
62     def mouseMoveEvent(self, event):
63         xpos = abs(event.pos().x())
64         print "X Pos: ", xpos
65         self.box.setSpeed(max(xpos / float(self.speedFactor), 1))
66 
67     def keyPressEvent(self, event):
68         if event.key() == Qt.Key_F:
69             self.toggleFS()
70 
71     def toggleFS(self):
72         if self.isFullScreen():
73             self.showNormal()
74         else:
75             self.showFullScreen()
76 
77 if __name__ == '__main__':
78     app = QApplication(sys.argv)
79     mywin = MyWin()
80     sys.exit(app.exec_())
```

A few things to note in this program...

There's a few new things going on in this program. Besides the use of `QObject`'s built-in timers for the animation of `GlowBox`, this program also is probably your first encounter with Qt's full-screen functions, and with a new event handler, `keyPressEvent`, that can be used to respond to keyboard input.

Inside the init method of `GlowBox`, ***Line 13*** starts a built-in timer that will be used to animate the color of the box. We save the unique ID returned by `startTimer` in the instance variable `timerID`. We'll use this ID later to kill the timer.

Lines 24-26 define the overriden function timerEvent(). This method is called every time one of GlowBox's built-in timers emits a timeout signal. It calls updateColor, which increments the RGB values drawn to the screen by one step, then tells the widget to redraw itself.

Lines 38-41 implement the setSpeed method. First the current timer is killed using the ID we stored earlier. Then a new timer is started using a different frame delay.

```
        self.killTimer(self.timerID)
        self.frameDelayMillis = frameDelay
        self.timerID = self.startTimer(self.frameDelayMillis)
```

Inside the application window (`MyWin`) there are a few new event responders worth noting.

`resizeEvent(self, event)` On ***lines 59 and 60***, this is the method of `QWidget` that you can override to respond to the user resizing the window. In this case we use it to make sure that the `GlowBox` widget grows with the window dimensions. Whenever the window is resized, the glowbox is also resized to fill the whole window. event is a `QResizeEvent`, which includes information about the old and new sizes of the window.

`keyPressEvent(self, event)` On ***lines 67-69***, is the method you override when you want to respond to keyboard input. The event is a `QKeyEvent` that contains information about the key that was pressed and any modifier keys that were down when the key was pressed. The most useful method is `key()`, which gives you the key that was pressed as one of Qt's key value constants. In this program we test if the key that was pressed is F (`Qt.Key_F`). If true, we toggle full-screen mode by calling `toggleFS()`.

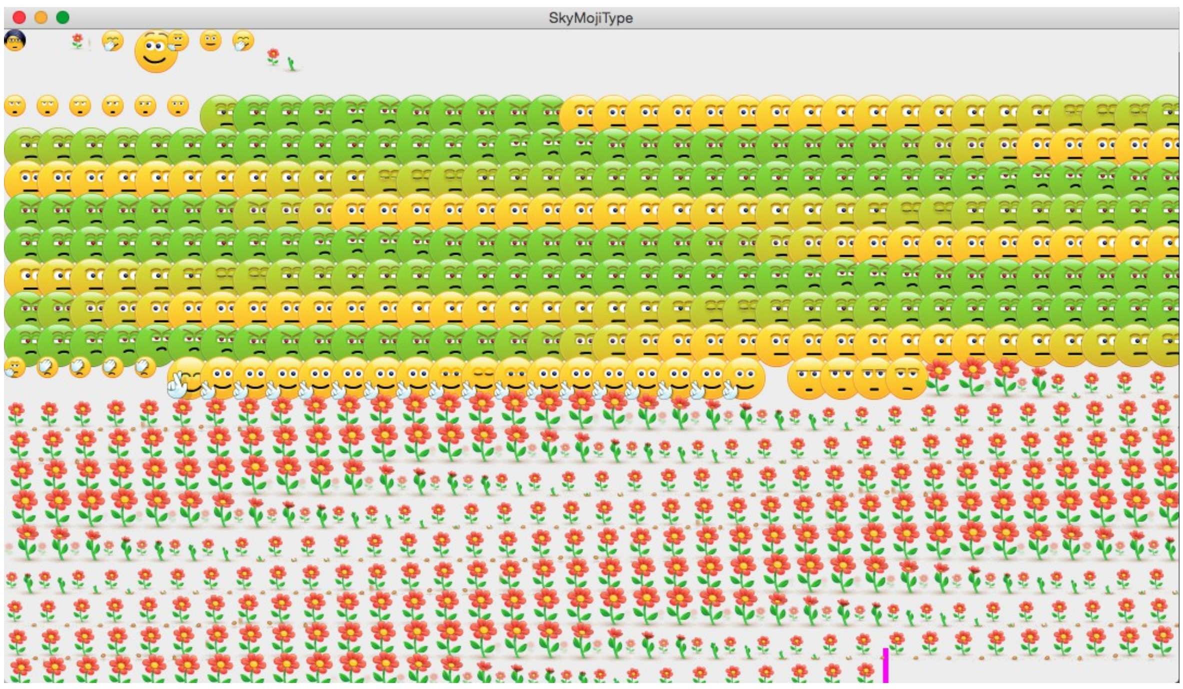
Lines 71-75 show some of Qt's full-screen app functions at work. These are methods of QWidget that can be used whenever a QWidget is functioning as a window. The methods are:

`isFullScreen()`	Returns a boolean true or false whether the window is currently running in full-screen mode.
`showNormal()`	Run the window in normal (windowed) mode
`showFullScreen()`	Run the window in full-screen mode

Emoji Ciphers

The next two examples will work through the creation of a slightly more ambitious program. We're going to build a typing app that translates user key strokes into Skype emoticons.

First we'll go through how to find the application resources (images, emojis, etc..) that are used by application developers to build the interface of common softwares. We'll be focusing on using the emoji image files in Skype, so in the *second part* we'll examine how those animations are stored and build a custom widget that will just read in one of those files and handle the animation for us. *Finally*, we'll build a typing console that translates each key the user types into a unique emoji, represented by our custom widget.

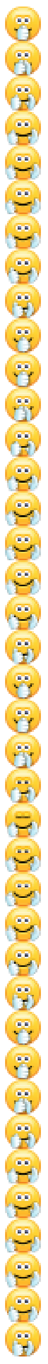


1 - Pilfering Resources

You can find application resources in a number of places. For web-based applications it's usually a matter of searching through the website's source code and looking for image formats. Most browsers have a "View Source" option or "Developer Mode" that allows you to sniff through the source code of a site. Although sometimes it's as easy as right-clicking an icon you want to use and doing a "Save Image As".

Accessing resources from Desktop apps can be slightly more difficult, as desktop applications are usually distributed in as black-boxed a way as possible by their producers.

On OSX all applications are really folders with a (usually hidden) `.app` extension. If you right-click an application, you'll have the option to "Show Contents". This will open up the application folder and allow you to dig around for resources. Usually you'll find all the image, sound and video resources in the `Contents/Resources` directory inside the application folder.



On Windows the application resources are significantly more black-boxed than on OS X. Most developers package their resources inside dynamically loaded libraries (`dll` files), or even inside the compiled application file (the `exe` file). These can be hard to crack open, but some google searching can turn up a few utilities for getting the resources out of these files. I found a few freeware programs by NirSoft that do the trick.

2 - Our Animated Emoticon Widget

I cracked open Skype on OS X and started examining some of the image resources inside. All of these are in the github repository for this worksheet in the directory `skype/img` - be sure to have this folder in your working directory before continuing with this part of the worksheet!

Open up any of the files with "anim" in their name and you'll see a long vertical image like the one shown to the left (this one is clap_anim@2x.png). Skype stores animated icons as single images with each frame of the animation following the last in a long vertical column. All of the animated icons are also perfect squares.

Let's start by building a widget that can read in these image files and animate them. To turn this long image into an animation we need to go through it frame by frame, each time drawing only a small portion of the image. Our widget will have to keep track of how many frames there are in the animation and which frame it's currently on. When the algorithm reaches the last frame of the image, it will need to start again from the beginning so that the animation loops the way you would expect with an emoticon.

SkyMoji.py

```
from PyQt5.QtWidgets import QWidget, QApplication
from PyQt5.QtGui import QPixmap, QPainter
from PyQt5.QtCore import QRect
import sys

class SkyMoji(QWidget):
    def __init__(self, imgpath, parent=None):
        super(SkyMoji, self).__init__(parent)
        self.currentFrame = 0
        self.setPixmap(QPixmap(imgpath)) # load image file as a Pixmap
        self.startTimer(1000 / 15) # start the animation timer at 15 frames per second

    def setPixmap(self, pix):
        self.pixmap = pix
        self.frameWidth = pix.rect().width()
        self.frameHeight = self.frameWidth # Skype emoticons are perfect squares
        self.numFrames = pix.rect().height() / self.frameHeight # Calculate number of frames
        self.resize(self.frameWidth, self.frameHeight)

    def paintEvent(self, event):
        qp = QPainter(self)
        ypos = self.frameHeight * self.currentFrame # which part of the image to display
        sourceRect = QRect(0, ypos, self.frameWidth, self.frameHeight)
        # copy the part of the image in sourceRect to the canvas, scaling if necessary
        qp.drawPixmap(self.rect(), self.pixmap, sourceRect)
```

```
26
27     def timerEvent(self, event):
28         self.currentFrame += 1
29         if self.currentFrame == self.numFrames:
30             self.currentFrame = 0 # start the animation back at frame 0
31         self.update() # force the widget to redraw itself
32
33 if __name__ == '__main__':
34     app = QApplication(sys.argv)
35     filepath = "skype/img/fingerscrossed_anim@2x.png"
36     mywidget = SkyMoji(filepath)
37     mywidget.setGeometry(50, 50, 200, 200)
38     mywidget.setWindowTitle("Fingers Crossed")
39     mywidget.show()
40     sys.exit(app.exec_())
```

A few things to note in this program...

On ***Line 6*** we create the custom `SkyMoji` widget that will load its own image and handle its own animation.

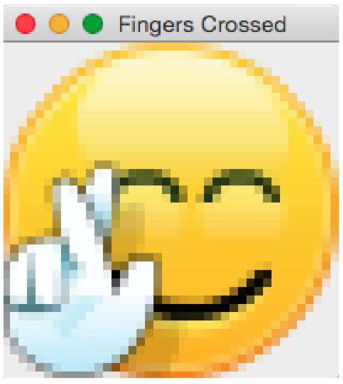
The `init` method on ***Lines 7-12*** takes two arguments besides `self`: `imgpath` is the path to the image file that will be loaded into this widget, and `parent` is the parent widget/window. An instance variable for the current frame is created and initialized to frame 0. The image file is loaded into a `QPixmap` and stored for use in the animation. And the animation timer is started at 15 frames per second.

Lines 13-18 defines the method setPixmap, which is used mainly in the init method to analyze the dimensions of the loaded image and set up the widget's instance variables for animation. First we figure out the dimensions of each animation frame by getting the width of the pixmap (***line 15***), and assuming that the height will be the same, since skype emoticons are perfect squares (***line 16***).

Then we calculate the number of animation frames in the image by dividing its height by the height of a single frame (***line 17***). Finally, we set the dimensions of the widget to match the dimensions of the emoticon (***line 18***). This is everything we need to do the animation.

The `paintEvent()` method, defined on ***Lines 20-25***, only needs to stupidly draw the current frame of the animation to the screen based on what the variables tell it to do. To do this we're going to use one version of `QPainter`'s `drawPixmap()` method. `drawPixmap()` is the QPainter method used for drawing pixmaps to the widget's canvas. It is an overloaded method, meaning that there are many possible combinations of arguments that you can use with it. The version we're interested in is the one that lets us draw a specific section of a pixmap to the widget's canvas.

First we calculate the y position of the animation frame we want to draw (**line 22**). Then on **line 23** we create a `QRect` that defines the section of the image that contains the frame we want to draw. Finally on **line 24** we tell the `QPainter` to draw from this section into the boundary rectangle of the widget. Something nice about this version of drawPixmap() is that it will automatically scale the source image to the dimensions of the destination rectangle, this means if you resize the widget the emoticon will scale to fit.



`drawPixmap()` *will scale the source image to the dimensions of the destination rectangle*

Lines 27-31 define the `timerEvent()` method that gets called by our timer 15 timers per second. All that this method needs to do is update the state variables for the animation and tell the widget to redraw itself. This means incrementing the current frame, and looping it back to frame 0 if necessary.

3 - The SkyMojiType Machine

Now that we have the `SkyMoji` widget, we can create the typing app. In this program we'll have a console window where the user can type and have their key input translated into emoticons from the Skype resource directory. We're going to import the `SkyMoji` class into this program, so make sure `SkyMoji.py` is in your project directory. Also be sure you have all the Skype image resources in the `skype/img/` directory of your project.

We're going to use python's `glob` module to make a list of all the animated icons inside the Skype image resource directory. Then we'll use a little simple math to convert input key numbers to indexes of this list. We want the user to be able to use the spacebar, tab and return keys as expected, so we'll add a little extra logic to handle these special cases. All of this happens inside the `keyPressEvent()` method of our window class.

SkyMojiType.py

```python
from PyQt5.QtWidgets import QWidget, QApplication, qApp
from PyQt5.QtCore import Qt
from PyQt5.QtGui import QPainter, QPen
import sys
import glob
from SkyMoji import SkyMoji

class TypingWindow(QWidget):
    charSpace = 30
    tabSize = 4

    def __init__(self):
        super(TypingWindow, self).__init__()
        self.setWindowTitle("SkyMojiType")
        h = qApp.desktop().screenGeometry().height() - 200
        w = qApp.desktop().screenGeometry().width() - 200
        self.setGeometry(w/10, h/10, w, h)
        self.lineLength = w / self.charSpace
        self.charCount = 0
        self.cursorVisible = True
        self.emojis = []
        self.allImages = glob.glob("skype/img/*anim*.png") # file names containing 'anim' and ending in .png
        print "Total Animated Files Found: ", len(self.allImages)
        self.startTimer(200) # Animation timer for cursor, don't save the ID as we don't plan to kill it
        self.show()

    def timerEvent(self, event):
        if self.cursorVisible:
            self.cursorVisible = False
        else:
            self.cursorVisible = True
        self.update() # redraw yourself!

    def paintEvent(self, event):
        qp = QPainter(self)
        if self.cursorVisible:
            pen = QPen(Qt.magenta)
            pen.setWidth(5)
            qp.setPen(pen)
            xpos = (self.charCount % self.lineLength) * self.charSpace
            ypos = (self.charCount / self.lineLength) * self.charSpace
            qp.drawLine(xpos, ypos, xpos, ypos + self.charSpace)

    def keyPressEvent(self, event):
        key = event.key()
        if key == Qt.Key_Space:
            self.charCount += 1
        elif key == Qt.Key_Return or key == Qt.Key_Enter:
            self.charCount = ((self.charCount / self.lineLength) + 1) * self.lineLength
        elif key == Qt.Key_Tab:
            self.charCount += self.tabSize
        elif key >= 0x21 and key <= 0x5A:
            # Decode Key
            filepath = self.allImages[key - 0x21]
            xpos = (self.charCount % self.lineLength) * self.charSpace
            ypos = (self.charCount / self.lineLength) * self.charSpace
            emo = SkyMoji(filepath, self)
            self.emojis.append(emo)
            emo.move(xpos, ypos)
            emo.show() # you need to explicitly show() widgets created after their parent window is shown
            self.charCount += 1

if __name__ == '__main__':
    app = QApplication(sys.argv)
    mywin = TypingWindow()
    sys.exit(app.exec_())

```

A few things to note in this program...

Take some time to study this program. There really isn't much new here. Moreso it's an integration of all the information from the past few worksheets on Qt.

On ***line 9 and 10*** we create two class constants that define the number of pixels represented by a "character" or space, and the number of characters represented by a tab stop.

Inside the init method on ***lines 12-25*** we set up all the instance variables for the typing app. The geometry of the window is calculated, and from that we calculate the number of emojis per line (`lineLength`). We create an instance variable, `charCount`, that will keep track of the total number of emojis (or spaces) that have been typed. This is important for calculating when it's time to move to the next line.

To make the cursor blink we'll have an instance variable, `cursorVisible`, that works with a timer to tell the `paintEvent` method when to draw the cursor and when not to. We have a list, `emojis`, that will be used during the typing process to store new `SkyMoji` objects as they are created. And another list, `allImages`, that will store all the animated image filepaths from our skype resource directory. We get this list of files using the `glob` module. You can read more about using glob here.

Finally, on ***lines 24 and 25***, we start the cursor animation timer and show the window to the user.

The `paintEvent` method is responsible for drawing the blinking cursor. It's just drawing a vertical line if `self.cursorVisible` is true. Worth taking a closer look at are ***lines 40 and 41***.

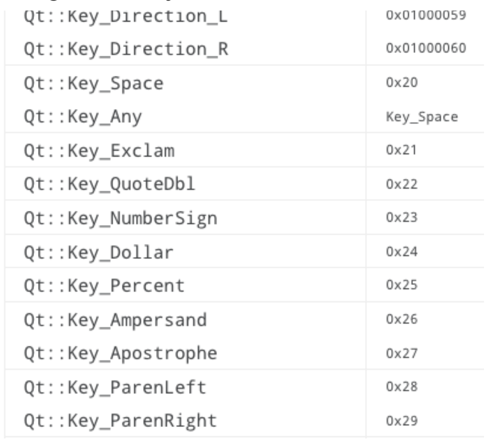
```
xpos = (self.charCount % self.lineLength) * self.charSpace
ypos = (self.charCount / self.lineLength) * self.charSpace
```

These two lines calculate the current x and y position of the cursor based on the number of emojis (and spaces) that have been typed so far. The current y position is the number of emojis divided by the emojis per line (multiplied by the number of pixels per emoji). The current x position is the *remainder* of that same division. In graphics programming breaking a list of objects into lines is a common thing you want to do, so it's good that you understand how this calculation works!

The keyPressEvent method on ***lines 45-62*** deserves some explaining. Remember from the GlowBox program that the event argument here is an instance of `QKeyEvent`. It's `key()` method will return the key that was pressed in the form of one of Qt's key value constants. These constants are really just integer variables.

Lines 53-62 look for keys in the range of 0x21 -0x51, and subtract 0x21 from this range of numbers so that it starts at 0. Then we can use these numbers directly as indexes for our list of emoticon file paths (***line 55***). I picked this range of key constants because they all represent printable characters that you would normally type from your keyboard (but I am leaving out a few!).

Now that we've decoded the key to a specific emoticon image, we need to create a new SkyMoji widget and position it at the cursor. On ***lines 56 & 57*** I calculate the current cursor position. Then on lines ***58-61*** I create the widget, add it to the emojis list, move it to the cursor location, and make it visible. This last part is necessary because widgets created after a window has been shown to the user are by default hidden.



Qt::Key_Direction_L	0x01000059
Qt::Key_Direction_R	0x01000060
Qt::Key_Space	0x20
Qt::Key_Any	Key_Space
Qt::Key_Exclam	0x21
Qt::Key_QuoteDbl	0x22
Qt::Key_NumberSign	0x23
Qt::Key_Dollar	0x24
Qt::Key_Percent	0x25
Qt::Key_Ampersand	0x26
Qt::Key_Apostrophe	0x27
Qt::Key_ParenLeft	0x28
Qt::Key_ParenRight	0x29

An excerpt of Qt's keyboard constants. Their integer values are written in either binary or hexadecimal.

Finally, on ***line 62*** I increment `charCount`, moving the cursor one position forward.

The special cases for tabs, spaces and returns on ***lines 47-52*** follow a similar logic. The key task being to increment `charCount` by the right amount so that it creates a space, tab or line return.

Wrap-up

Take some time to study and understand the above examples. It's a lot of material, but if you can understand how these programs work then you've got a lot of power at your fingertips to make your interfaces act and react in imaginative ways. Consider how you can take the resources from everyday pieces of software and estrange them, play with expectations and the aesthetics we have normalized.