An Introduction to **qtpaint**

Michael Lawrence

November 11, 2015

1 Overview

The **qtpaint** package supports interactive graphics in R through low-level interaction with the Qt toolkit. It is meant for experienced R users, who are familiar with GUI and graphics programming. It is expected that a number of packages will wrap **qtpaint** to provide a more convenient and user-friendly interface for the most common tasks.

The interface provided by **qtpaint** is intentionally barebones, essentially covering only the functions necessary for efficient drawing. Implementing anything beyond a basic static graphic will require use of the **qtbase** package and its complete, direct interface to the Qt library. Most of the time you will need to load both **qtbase** and **qtpaint** packages.

1.1 Features

An interactive graphic is defined by two types of logic: drawing and handling user input. The latter is handled reasonably well by R graphics devices, though this is more true of some than others. Despite this, handling user events is difficult for graphics drawn through the **grid** package, upon which **ggplot2** and **lattice** are based. **qtpaint** responds to a comprehensive set of input events, from low-level mouse movement to high-level drag and drop actions. It also uses efficient algorithms and data structures for mapping event coordinates back to graphical primitives and the data.

However, the key contribution of **qtpaint** is drawing. The base R graphics system is inflexible and suffers from poor performance. While more flexible drawing is possible with **grid**, it is even slower, as it is bound to the same underlying graphics engine. **qtpaint** preserves the flexible grid-based layout mechanism of **grid**, while delegating to a highly optimized graphics engine. The **qtpaint** strategy for fast graphics may be explained through a physical analogy. If a person wishes to move from point A to point B, the time to reach point B depends on the speed of the person along the route and on the length of the route. Increasing the speed of the person or decreasing the length of the route will result in a shorter time. For drawing, this translates to moving pixels to the screen faster and decreasing the number of pixels that must be drawn.

This draw-less-faster approach is achieved by the following techniques:

- Leveraging the now ubiquitous GPU through OpenGL
- Vectorization of drawing routines at both the R and device level
- Rasterizing a type of glyph only once and blitting each instance, rather than drawing from scratch each time
- More direct access to the graphics hardware by skipping intermediate layers and representations
- Separating the graphic into layers and caching each layer individually, enabling incremental updates

The first point could be implemented through a custom R graphics device, and the next two might be achievable through relatively minor modification of the R graphics system. However, there is simply too

much logic between the R-level plot command and the device, and the last point, cached drawing layers, represents a fairly significant conceptual shift. Hopefully, in time, high-level plotting interfaces will wrap this experiment, and the combination will become a prototype for the next generation of graphics in R itself.

As in R graphics, all drawing operations are defined by an abstract interface implemented by drivers that produce a particular type of output. For example, one driver targets *QPainter* (and all of its implementations, including many off-screen devices), while another implements OpenGL fast-paths on top of *QPainter*. Thus, while our primary use-case is drawing on the screen, the output is just as flexible as in R.

1.2 Layers, Scenes and Views

In **qtpaint**, layers draw everything and handle all input events. A layer is represented by an instance of the C++ *Qanviz::Layer* class, which descends from the Qt*QGraphicsItem* class.

It is possible to draw directly to the scene, but, by default, **qtpaint** asks Qt to cache the drawing at the pixel level. The default cache mode is at the device, so that if the device dimensions change, the layer must be redrawn. An alternative mode will scale the cached image to fit the device, but this is not generally recommended as it results in poor quality. One possible use-case would be to smoothly animate transitions through rasterized scaling, leaving the redraw until after the transition is finished (like Google Maps).

The rationale behind caching pixels, rather than a logical representation, such as a vector path or display list, as in R, is performance. Maintaining a display list is slow, and while display lists are independent of device dimensions, it is not ideal to simply replay a list when the device dimensions change. Intelligently responding to device resizing is a difficult problem, and while R behaves somewhat reasonably, one often wishes to customize the logic at the Rlevel, and **qtpaint** supports this by simply requesting a redraw.

A secondary function of the layer is layout: a layer can have child layers, and the children are laid out on a grid within the region of the parent. The heights of the rows and the widths of the columns need not be regular. Importantly, children can be stacked and thus draw over each other, but due to restrictions in Qt, this is only possible if the layout has a single cell.

Every layer belongs to a scene, an instance of the Qt *QGraphicsScene*. The scene automatically expands to fit its contents. Usually, one will add a single root layer to the scene, and the root layer arranges everything else on a grid.

A scene is displayed by one or more views. A view displays a particular portion of the scene, at a particular scale. For plotting data, **qtpaint** provides the *PlotView* class, an extension of *QGraphicsView*. *PlotView* adds two key functions: special rescaling behavior when the view is resized, and overlay support.

Normally, when a *QGraphicsView* is resized, it simply shows more of the scene. This is different from the R graphics device, where the plot (the equivalent of the root layer) is rescaled to fit the device. This is the default resize mode for *PlotView*. There are two ways for this rescaling to occur: by changing the geometry of the layers or by changing the zoom factor of the view. Resizing the geometry, the default, affects all views and thus only makes sense when there is a single view of the scene. The advantage of it over the view-specific zooming is that the layout logic is considered. For example, a layout can be packed so that only the plotting area is resized, while the size of the axes does not change.

The overlay of *PlotView* is a special scene with its own layers. The geometry of the overlay scene is fixed so that it always exactly covers the viewport. This facilitates drawing plot annotations that are always displayed in the same way, even as the view is zoomed and panned.

1.3 Event-driven Model

In the **qtpaint** model, both drawing and input handling are implemented in an event-driven manner, which should be familiar to those who have ever developed a GUI. It is easy to see why this is necessary: without notification from the event loop, the program would never know when a graphic needed to be redrawn (e.g., due to a device resize), or when a user had performed a particular action. This represents an inversion of control from the conventional approach to drawing graphics, where the drawing commands are initiated

directly from R, not in response to a request by an external event loop. For example, the code below draws a scatterplot of some normally-distributed values in conventional base graphics fashion:

```
> x <- rnorm(100)
> plot(x)
```

In the event driven model, the `plot(x)` command is not issued immediately by the script; rather, it forms the body of a function which is registered with the external event loop. The R program then yields control to the event loop, and the function, known as a *callback*, is called by the event loop upon demand:

```
> x <- rnorm(100)
> drawScatterplot <- function() { plot(x) }
> # <code that registers drawScatterplot with event loop>
> # control is then passed to event loop
```

The reasons for this will become clear as we discuss each type of logic in the following sections.

2 Drawing

2.1 Basics

Upon creation of a view or whenever a view needs to be updated, the scene needs to be redrawn. The scene asks each layer to draw itself. The order in which the layers are drawn depends on the order in which they are stacked. Drawing proceeds from the bottom up, so that a layer is masked by those above it. The drawing logic is implemented in a callback function provided by the user. Drawing commands are passed to an instance of the C++ class *Painter*.

The *Painter* class defines an interface that is independent of any output type. The methods can be divided into two types, those for state configuration and those for actually drawing the graphical primitives. The state includes options like color, line aesthetics, coordinate transformation, and font. Supported primitives include lines, points, rectangles, circles, polygons, text and plotting glyphs. All functions are vectorized over the primitive coordinates, as well as certain visual attributes, like color.

Each painting callback must accept at least two arguments. The *Layer* instance is passed as the first, and the *Painter* comes second. An optional third argument informs the callback of the rectangular region that must be redrawn. This is expensive to calculate and somewhat bothersome to consider, so it is normally omitted, but it is useful for some optimizations. Below, we define a callback function that will call *Painter* functions to paint a basic scatterplot on a layer:

```
> df <- data.frame(x = rnorm(100, 300, 100),
+                  y = rnorm(100, 200, 80))
> scatterPainter <- function(layer, painter) {
+   qdrawCircle(painter, df$x, df$y, 5)
+ }
```

The `qdrawCircle` function will draw circles on the layer. Like every other `qdraw` function, the *Painter* must be passed as the first argument. This differs from the base R approach, where the device is implicit and managed through functions like `dev.set`. The rest of the arguments pass the (X, Y) positions and the radii of the circles. Note that the recycling rule applies for the radius argument.

Next, we construct a layer that will delegate to our callback for painting. The convenience constructor of *Layer* is `qlayer`:

```
> library(qtpaint)
> library(qtbase)
> scene <- qscene()
> scatterLayer <- qlayer(scene, paintFun = scatterPainter)
```

In the above, we first create a scene for the layer and then create the layer. The first argument to `qlayer` is for the "parent," which can be either the scene (for a root layer) or another layer. It is not necessary to provide a parent at construction time, but it is usually most convenient.

To display the layer, there is one final step: creating a view of the scene and "printing" it to the screen.

```
> view <- qplotView(scene = scene)
> print(view)
```

The circles in our scatterplot are drawn with an efficient algorithm, but each circle is drawn from scatch. Since they all look the same, we should be able to render a single circle and copy its pixels wherever we need a circle in our plot. This would be much faster. **qtpaint** provides `qdrawGlyph` for this purpose. The first step is to define our circle glyph, optimally outside of the paint callback.

```
> circle <- qglyphCircle()
```

The `qglyphCircle` function creates a vector representation of a (by default) radius 5 circle. Other glyph shapes are included; see *QPainterPath* for creating a custom glyph.

To use the glyph, we redefine our paint callback:

```
> scatterPainter <- function(layer, painter) {
+   qdrawGlyph(painter, circle, df$x, df$y)
+ }
```

The plot should appear the same, but when there are many glyphs, it will be drawn much more quickly.

Thus, we have created a scatterplot. However, one might notice that we are plotting in pixels, which is not very convenient for visualizing data. **qtpaint** facilitates the mapping from data space to device space, and we introduce this in the next section.

2.2 Coordinate Systems

As **qtpaint** is based on the Qt Graphics View framework, it inherits the same treatment of coordinate systems. There are four basic coordinate systems: item (layer), scene, view and viewport (device). Between each of the systems, there is a transformation expressed as a *QTransform* object, essentialy a 3x3 matrix encoding translation, scaling, rotation and shear. Initially, the transformations are all set to identity, so data is painted in device space, as above.

From the perspective of data plotting, each system has a distinct and important role:

Layer Data space; the coordinates in which the layer paints itself.

Scene Figure space; the coordinates in which each layer is laid out.

View View space; result of view-specific zoom.

Viewport Device space; scrollbar-mediated translation of view coordinates.

Usually, only the transformation from layer to scene is necessary, which would convert data coordinates to device coordinates. This is the only transformation performed by base R graphics. Supporting multiple views requires transformations between the scene and each view. The final transformation, from view to viewport, occurs when the view widget is not large enough to contain the entire scene and thus allows the user to adjust the viewport via scrollbars. Note that this is not possible in the default mode of *PlotView*, where the geometry of the layers is constrained (and layer to scene transforms adjusted accordingly) so that the view always fits in the widget.

Configuring the layer to scene transform for plotting data is simple. The limits of the data need to be passed as a rectangle to the `qlayer` constructor. It is also possible to modify this after construction. Below, we recreate our layer with limits:

```
> scene <- qscene()
> scatterLayer <- qlayer(scene, paintFun = scatterPainter,
+                        limits = qrect(0, 0, 600, 400))
> print(qplotView(scene = scene))
```

Although the plot will look the same initially, the plot now scales as the widget is resized.

When moving beyond non-trivial graphics, it becomes necessary to take into account pixels when drawing. An example is labeling a point. The `qdeviceTransform` function retrieves the *QTransform* used by *Painter* to convert data to device coordinates. Vectorized mapping of data points through the *QTransform* is performed by the `qvmap` function. The *QTransform* can be queried, modified, inverted, etc, using the **qtbase** package; see the `Qt` documentation for more details.

2.3 Layout

In the previous section, we demonstrated a simple scatterplot, without axes or other plot decorations. Such simplicity often works well for exploring data through interactive graphics. However, for presentation, decorations and guides become a necessity. Such complex plots consist of multiple components, almost always laid out on a grid, as supported by the **grid** package.

The layout strategy in **qtpaint** is also grid-based, and is managed by an instance of the *QGraphicsGridLayout* class, not the *Layer* itself. The layout object defines the bounds of each layer in scene (or "figure") coordinates. These bounds are called the *geometry* of the layer. Since the layout controls the geometry of its child layers, it really only makes sense for the user to set the geometry of the root layer. Normally, the root layer geometry defines the extents of the scene, which, by default, matches the extents, in pixels, of the view widget.

Specifying a layout requires three steps. The first is to add each component layer to a grid cell of the root layer, which contains the entire figure. We add child layers to the root layer upon their construction. The `qlayer` constructor takes the zero-based row and column grid coordinates as arguments. We create a plot with a title, X axis, Y axis and plotting area:

```
> scene <- qscene()
> figLayer <- qlayer(scene)
> titleLayer <- qlayer(figLayer, titlePainter, rowSpan = 2)
> yaxis <- qlayer(figLayer, yAxisPainter, row = 1)
> plotLayer <- qlayer(figLayer, plotPainter, row = 1, col = 1)
> xaxis <- qlayer(figLayer, xAxisPainter, row = 2, col = 1)
```

We skip implementation of the paint callback functions above, as we are only illustrating the configuration of the layout. The `row` and `col` arguments specify the zero-based row and column coordinates, respectively, with the origin at the top-left of the parent. They default to zero. The `rowSpan` and `colSpan` arguments default to one and indicate how many rows or columns are spanned by the layer, where `row` and `col` represent the top-left coordinate of the rectangle.

It is also possible to specify the layout after constructing the layers, using the familiar `[<-` function. We construct the same layout as before with existing layers:

```
> figLayer[0, 0, rowSpan=2] <- titleLayer
> figLayer[1, 0] <- yaxis
> figLayer[1, 1] <- plotLayer
> figLayer[2, 1] <- xaxis
```

The second step of layout specification is telling the layout object our preferred (default) height and width (in scene units) for each row and column, respectively:

```
> layout <- figLayer$gridLayout()
> layout$setRowPreferredHeight(0, 75)
```

```
> layout$setRowPreferredHeight(1, 400)
> layout$setRowPreferredHeight(2, 75)
> layout$setColumnPreferredWidth(0, 75)
> layout$setColumnPreferredWidth(1, 600)
```

It is also possible to set the maximum and mininum heights and widths by simply replacing `Preferred` with `Maximum` or `Minimum` in the calls above.

An alternative to the above is to allow the layout object to derive the row heights and column widths automatically from the size requests of the layers. The layout will attempt to satisfy the preferred width and height of every layer (set with the `setPreferredWidth/Height/Size` methods). However, such requests may not be fulfilled, as the layout is given a finite amount of space to distribute to the layers. This is a problem, for example, if a layer is displaying text. The size of the text (calculated using `qtextExtents`) depends on the font parameters and thus is not controlled by the layout. The solution is to set a minimum size on the layer (with the `setMinimumWidth/Height/Size` methods). To satisfy the minimum size of every layer, the layout will request the necessary space from its parent. This request will be satisfied unless a maximum size constraint is violated.

We are now two thirds of the way to having a reasonable layout for our figure. The final step is to configure the resizing behavior of each row and column in the grid. Quite often, we want most components to remain the same size, even if the size of the view increases. Only the plot area should scale with the view. To achieve this, we need to set the row and column stretch parameters on the layout object:

```
> layout$setRowStretchFactor(0, 0)
> layout$setRowStretchFactor(2, 0)
> layout$setColumnStretchFactor(0, 0)
```

The stretch factor indicates the proportion of the newly available space that should be allocated to the row or column relative to the others. For example, if a column has a stretch factor of two, then it will receive twice as much of the available space compared to a column with a factor of one. As one might expect, a stretch factor of zero prevents the layout from expanding the row or column when more space becomes available. The default stretch factor is one.

There is an important caveat when overlaying/stacking layers (a useful technique for efficient dynamic graphics). Stacking requires multiple layers to coexist with the same grid cell of the layout. This only works for layers with a single cell, and even then Qt will emit an annoying error message (but everything still works, apparently). Thus, stacking layers in a multi-cell layout will require some creative nesting of layers.

2.4 Performance Tips

Although **qtpaint** is well optimized, the low-level interface leaves many performance considerations to the user. Here are some useful tips.

Keep drawing simple The fast paths in **qtpaint** rely on certain properties of the drawing being easy to draw. Whenever possible, use zero (one pixel) line width, avoid line dashing, and avoid different stroke and fill colors. If the stroke and fill colors are the same, it is best to eliminate the stroke. For some operations, these rules may make no difference; for others, the performance will change by a factor of 100.

Sort primitives by color If possible, sort primitives by their color, as many operations are performed per color and may involve significant constant overhead. One operation were this is *not* necessary is drawing glyphs with OpenGL.

Use segments instead of lines There are two ways to draw lines: as multi-segment lines and as separate segments. The underlying libraries and hardware are not optimized for drawing many multi-segment lines. These lines also require extra effort to cleanly join the segments. To get around this, simply break lines into

their component segments. There will be no line joining, but this is not necessary when drawing pixel-width lines. The performance benefit is at least an order of magnitude.

Draw filled rectangles and circles This might seem strange, but thanks to windowing systems and games with explosions, graphics hardware is optimized for drawing filled, not open, rectangles and circles (up to a certain size). Whenever possible, draw filled rectangles and circles with no outline.

3 Input Handling

3.1 Basics

So far we have introduced how to display data on the screen. It remains to add interactive features to our graphics. As with drawing, the logic for handling user input is implemented in callback functions. There are 18 different user events to which a layer can respond. These include mouse click and move events, drag and drop events, key presses, and even context menu requests, among others. The stacking order of layers affects event handling in a way analogous to drawing. Events propagate from the top to the bottom. Once an event is accepted by a layer, the propagation ceases. Thus, just as in drawing, a layer masks those below it, even though the processing proceeds in the opposite direction.

Dealing with many of event types, like drag and drop, requires sophisticated knowledge of Qt. Here we demonstrate only the mouse press event, which occurs when the user clicks a mouse button over the layer, without pressing any of the buttons. All event callbacks take two arguments: the layer and the event description. The callback defined below will add a point to a dataset depending on where the user has clicked.

```
> pointAdder <- function(layer, event) {
+   df <<- rbind(df, event$pos())
+   qupdate(scene)
+ }
```

The above imagines that we have a two-column *data.frame* named `df` that is being drawn as a scatterplot. We retrieve the position (already mapped from screen to data coordinates) using `event$pos()`. This syntax may seem odd if one has not been introduced to the **qtbase** syntax for directly manipulating Qt objects. After the point has been added to the data, we need to display it in the plot, which requires calling the `qupdate` function, which directs the scene to redraw its layers. It is very common to call this at the end of event handlers, so that the graphic can respond to the user action.

For the event callback to be invoked in response to an event, it must be attached to a specific event type, on a specific layer. This occurs at construction time, as shown below:

```
> pointAddingLayer <- qlayer(paintFun = scatterPainter,
+                            mouseReleaseFun = pointAdder)
```

3.2 Mapping Coordinates to Data Records

As in the previous example, it is very common for an interactive graphic to provide an interface to the underlying data. Often, we are not adding a new record to the dataset, but performing some operation on an existing record, like changing one of its visual attributes. **qtpaint** leverages an efficient BSP tree algorithm, implemented by Qt, to quickly return the indices of the records that are drawn within some region of the plot.

The query, like everything else at the layer level, is specified in layer coordinates. However, it is often necessary to define the query in device coordinates. For example, we may want to color every point within 10 pixels of the mouse pointer. The transformation to device coordinates depends on the view that sent the event. We need to obtain that transformation, invert it, map the 20x20 rectangle to layer coordinates, center it on the current pointer position, and finally query for the records that fall within it. For example:

```
> pointBrusher <- function(layer, event) {
+   rect <- qrect(0, 0, 20, 20)
+   mat <- layer$deviceTransform(event)$inverted()
+   rect <- mat$mapRect(rect) # 20x20 square now in data space
+   pos <- event$pos()
+   rect$moveCenter(pos) # centered on the pointer data pos
+   hits <- layer$locate(rect) # get indices in rectangle
+   df$color[hits] <- "blue" # color points blue
+   qupdate(scene)
+ }
```

The `locate` method returns the position in the drawing order for each graphical primitive located within the given region. In Oour case, we simply draw points in the same order as the data, so it is easy to index into our dataset. Behind the scenes, an efficient spatial data structure is constructed the first time `locate` is called. This structure is cached so that it need not be rebuilt for each invocation of `locate`. If the any of the graphical primitives change their position in layer coordinates, the cache will need to be invalidated, so that a new spatial index is created. This is the purpose of the `invalidateIndex` method.

The next step would be to pass the callback as the `hoverMoveEvent` argument to the `qlayer` constructor. We are thus well on our way towards implementing highly interactive GGobi-like graphics purely within R.