Sums Qt Quick Calculator - app showcase

Sums is a QML calculator written using Qt Quick Components for Symbian v1.1 and custom components. This article explains the calculator design and implementation, and includes a brief discussion of tricks to improve application performance.

Introduction

This article explains how I implemented a calculator application using the Qt Quick Components for Symbian, 1.1. An overview of the interface design is given, showing how the QML Components are used to provide UI functions in an interface that also makes heavy use of custom-styled graphics. Some graphical tips are provided for creating custom designs that will align well with the Belle design language - particularly on how to produce "squircular" curves. A discussion of the implementation of one of the app's custom components is followed by a brief discussion of some

When the Qt Quick Components for Symbian were first released, my first approach to using these was to convert the apps I'd been working on to using the components. However, this proved to be less effective than hoped, because I was trying to solve the implementation issues of my app, *and* learn how to use a new UI component set at the same time.

So, to learn how to use these components properly, I decided to implement a simple, easily-described application - a calculator, and this is it.

The app itself is on sale on the Nokia Store.

Look And Feel

The Icon

Unless you're a practiced graphic designer, you're better off finding someone you know who's good with design, and knows how to use Inkscape or Illustrator. Give them the Symbian Iconography Guidelines, and a rough sketch of what kind of icon you want, and they should be able to run with it from there.

That said, I did draw my own icon, and here it is (drumroll please):

The icon was drawn using the Symbian Application Icon templates as a starting point. The kit containing this template also contains automated actions for Illustrator, and these were used to add the drop-shadow under the icon's central motif. This is a small detail, but like a lot of small details, it's the difference between something looking "right" and looking "not quite right".

Why I chose this icon

Briefly putting on my graphic design hat for a moment, here are some reasons why my app icon looks as it does.

Follow the colour coding: First, the background is orange, to match the usage category that this application is in. The Belle/Harmattan icons are colour coded, so as a general rule, you shouldn't use different colours without good reason. A "good reason" is that your company's branding uses that colour; a bad reason is that you happen to like orange/green/purple.

Use a simple motif:The second aim is that the image is strong enough to be visible at a small size. In this case, I had to reduce the appearance of the calculator down to a core "idea": here I chose just four buttons. A previous design with just an LCD display could have been for a digital watch just as much as for a calculator, and using more than four buttons made it hard to see what was actually on the icon. Finally, it doesn't matter very much that these buttons don't actually sit together on the calculator - the idea is to convey the *idea* of a calculator to the casual observer.

Stay in the focal zone: The icon content is within the central part of the icon surface, as per the guidelines. The eagle-eyed will notice that I did go slightly outside the lines here, but the "visual weight" of the image is contained by the focal zone.

The Application UI

The app is a calculator, and the user interface reflects this. It's a facsimile of a typical pocket calculator. At this point, you could argue that the mobile device is capable of far more than just copying an existing design for mechanical buttons, and you'd be right: Yes, there is scope to do something completely new and innovative, but in this case I kept things simple, because:

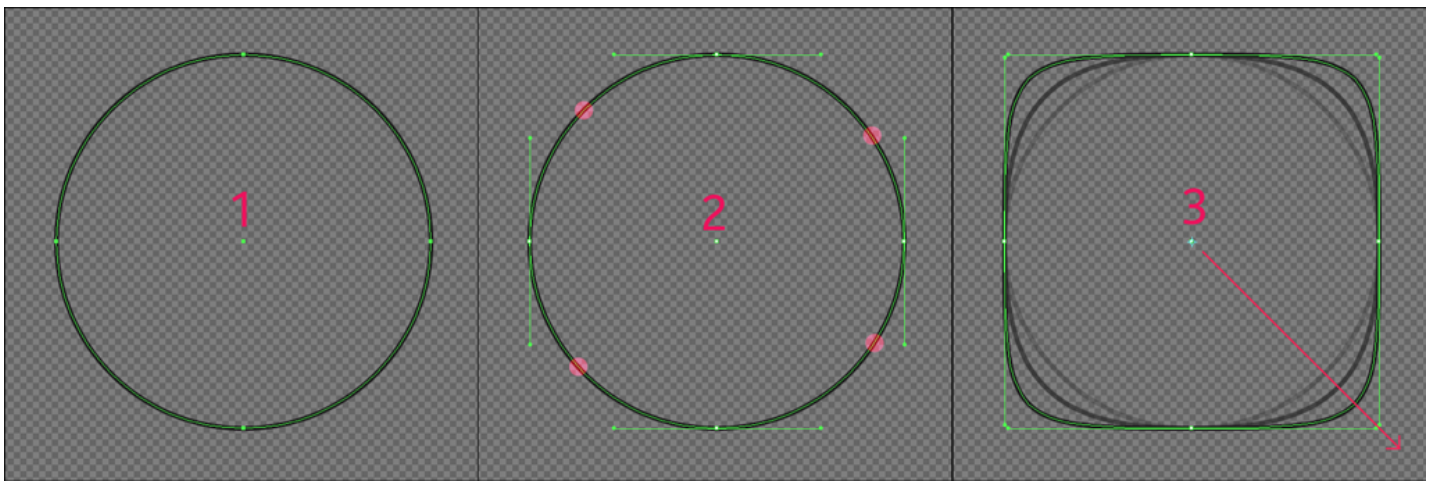
1. The user's expectation of a pocket calculator is pretty fixed.
2. Working out new and radical user interactions would get in the way of the goal of this project: to learn to use the QML Components

Très Belle: Superellipses made easy

Although most of the app's content is, in fact, composed of custom QML elements, it was important to try and have it blend in with the look and feel of the QML Components. Because of their large-radius corners, the standard Button components did not stack nicely into grids, so I had to use something a little more square.

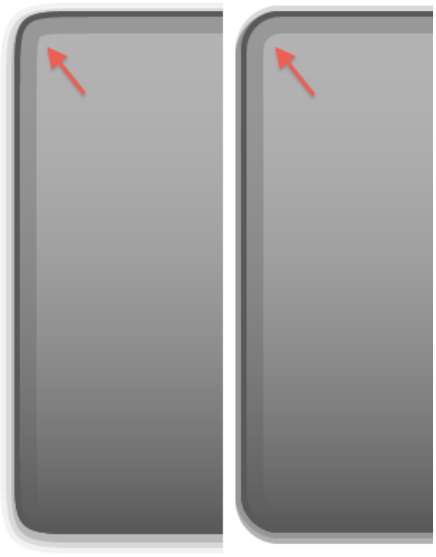
What I really wanted was the same kind of natural curves as the Belle buttons, but on a smaller radius... but how to draw these? Time for some research. The "squircle" shape used in Belle is actually just one of a larger family of shapes called "superellipses" (a name coined by the famed Danish poet and designer, Piet Hein, who used the shape extensively in his work).

Nice as these shapes are, your drawing package most likely does not have a built-in tool to create them. There is good news, however: you can copy the paths out of the icon templates if you need to make custom graphics with the same "squircle" shape that is used throughout the Belle UI. For any other kind of superellipse, however, there's a simple trick to get a very good approximation: Use your favourite vector drawing package, and follow these three steps:



1. Draw a circle of the appropriate size. The circle you get will be composed of four Bézier line segments (four anchor points and eight control ponts).

2. Using the "Direct selection tool" (Illustrator) or "Node tool" (Inkscape), click the four curved sections of the circle, but not the top, right, bottom, or left sides (click where the red dots are in the illustration). You should now have selected the eight Bézier control points, but not the four anchor points.
3. Use the Scaling tool to extend these control points outwards. The circle becomes more and more square as you drag the control points outwards. Don't overdo it, or you'll end up with a "cushion" shape.



Using a supelliptical curve to round off corners, left, produces a more organic shape than the circular corners, right. The effect is very, very subtle, but it looks less "machined" than the normal rounded-rectangle.

New and old

A calculator is a very functional application, so to add some visual interest, I decided to implement the display as a classic seven-segment LCD display. This involved drawing a set of digits, and developing a custom control (DigitDisplay) to show them. The calculator's display uses two of these DigitDisplay items, as shown below:

Because I had used a scalable graphics format for the artwork, it was possible to use the exact same image files for the large (base) and small (exponent) displays.

Don't move the furniture

Not all of the design decisions are based on visual appearance or logical grouping. Some are made just to make the application more comfortable to use, and more familiar to the user. This means adopting behaviour from other parts of the Belle system. As an example, the backspace key for the Sums app is placed at bottom right of the keypad layout. I have never seen a calculator with such a key in this position, but on the soft keyboard that the user uses every day in every other application, the backspace key is at the bottom right. Because the user will be used to going to this part of the screen to rub out input, this is where the equivalent key should go on the calculator.

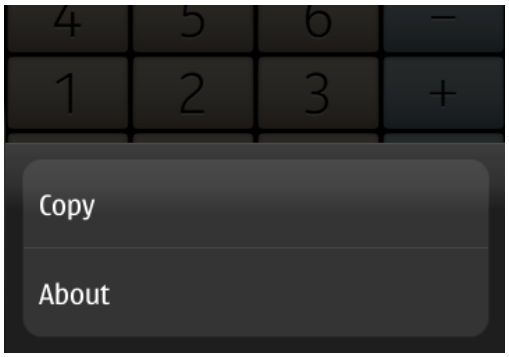
Conversely, "AC" (All Clear) is located far from the backspace key, because you don't want to accidentally press it, and most calculators place it at the top of the keypad.

Follow the pattern

This app is a normal utility, and so it should follow the general patterns of the Belle interface. To achieve this, the application is based on the Window element, with the app content contained in a Page item. The app's Window item includes both the StatusBar and ToolBar components.

Having the status bar allows the user to access Belle's notification and status screen, and there's really no reason to exclude this component on a utility app.

The toolbar is also a place to put those functions that don't have a natural place elsewhere. Currently the options menu (bottom right, where it's supposed to be!) has only two options: Copy the current value to the clipboard, and show the application details (the "About" box):



The back or exit button is at bottom left, as it is in every other part of the UI. On the main, top-level page, this button is changed to the "exit" button (a pattern that is also seen in Maps and the Store application); on other pages, it's "back".

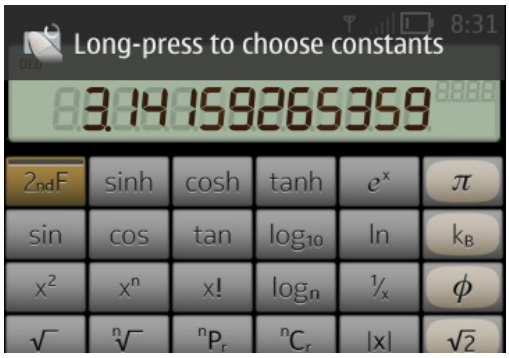
Unhiding hidden features

One of the less obvious features of the app are the four constant keys that can be used to insert mathematical constants - handy if

you can't remember the eleventh digit of pi (it's 5, by the way, and yes, I did have to check that):

Because there are more than four available constants, I made these keys user-programmable. The challenge was to indicate to the user that these keys can be changed, without interfering with their normal use of the app.

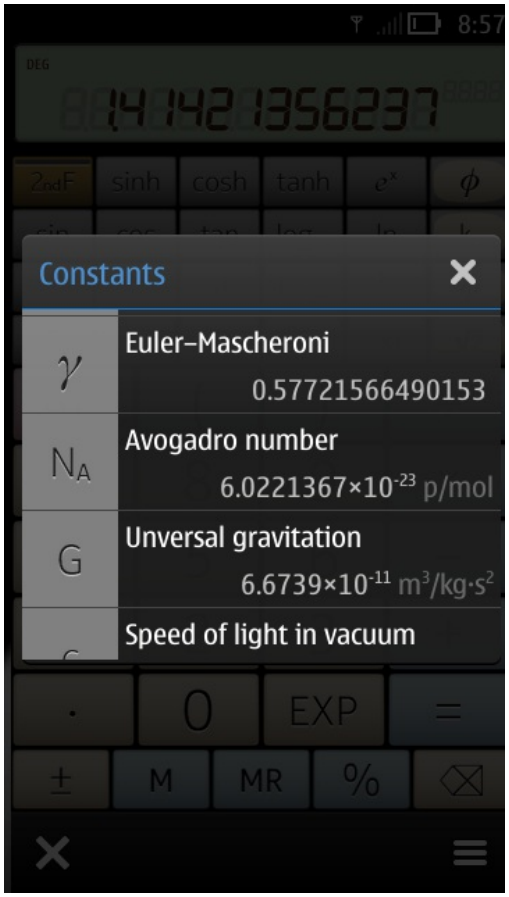
After trying a couple of different approaches, I finally settled on using an InfoBanner component. For the first two times the user presses the constant key, an InfoBanner with the prompt "Long-press to choose constants" is shown. After two uses of the constant keys, the bar is no longer shown.



The icon is there to provide an extra visual cue to users whose first language is not English, or for those who don't read the text in time.

Constant Selection

A long-press on any constant-key shows the following SelectionDialog, containing all the available constants:

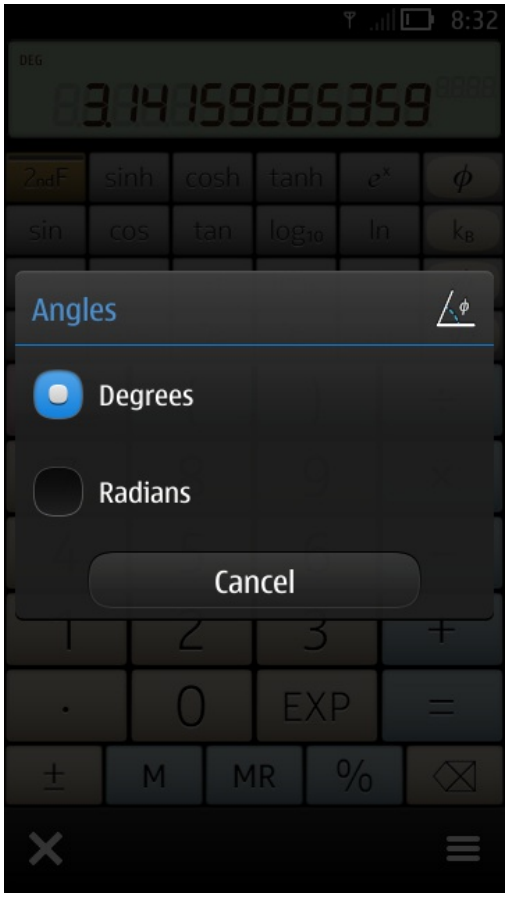


The user can tap any of these to assign it to a key. Tapping outside the dialog, or on the close-box, will dismiss the dialog with no change, as the user would expect. The list is flickable, as you'd expect from using the rest of the Belle-powered device.

This dialog also illustrates another small, and useful visual trick: the use of text colour to distinguish between the value and its units. The "units" string is drawn in a slightly darker-coloured type, which makes it appear "lighter" (thinner) than the value text. Because the Belle devices do not have a wide choice of font weights, this technique is a good way of emphasising or de-emphasising text without sacrificing readability (The different *role:* properties of the ListItemText element use this technique). Also note again how the use of scalable source graphics has allowed the exact same image files to be used for this list as for the calculator button captions.

Direct Manipulation: Angle mode

The other piece of "hidden" functionality in the UI is the Angles mode. The calculator can operate in either degree or radian mode for its trigonometric functions. The display shows the current more at top left. If the user taps on this, a CommonDialog dialog box is displayed with the options. Directly choosing one option causes the box to close, or the user can dismiss it and keep the current setting using the "Cancel" button.



Implementation

Buttons

The Layout of the calculator buttons made heavy use of the Grid layout item. There are three Grids: the function keys, the main keypad, and the smaller row of buttons. Each Grid chooses either the "main" model, or the "secondary" model, depending on whether the calculator's second function is engaged:

```
    Grid {
        ...
        model:  (calc.is2ndF)?  mainbuttons : mainbuttons_2ndf;
        ...
    }
```

Each button was represented by a custom type, CalculatorButton, which combined a button surface image and a caption image to produce the button. The list model contained fields which specified the function of the button (from which the caption image URL was generated), and the style, using which the button surface image was chosen.

The use of assignable constants makes the situation more difficult. The raw list model just refers to 'const_1','const_2','const_3' and 'const_4', but to find out what constants these are, the QML uses a small Javascript function that makes an enquiry to the application engine (in C++) and gets the appropriate constant for each button.

Custom Digit Display

The underlying calculator engine does some "cooking" of the number to be displayed, and it exposes three properties to the QML layer:

- the raw figures of the base (1.23446 becomes **123456**)
- the raw figures of the exponent (1.66e**-39** -->**-39**)
- the position of the decimal point within the base (13327.034 --> the point is 3 places from right)

These properties are text strings, and they are also pre-padded to the left or right (as appropriate). Doing this makes it easier to construct the digits using a simple http://doc.qt.nokia.com/4.7/qml-repeater.html Repeater] element. The code that does this is as follows:

File: DigitDisplay.qml

```
import QtQuick 1.1
import "DigitDisplayFunctions.js" as Logic;

Item {
 id: control
 property string text: ''
 property int places
 property int decimalPoint
 property int digitWidth: Math.ceil(height*0.65)
 clip: true
 width: places*digitWidth;
 Row
 {
   anchors.right: parent.right
   Repeater
   {
     id: digitRep
     model: control.places
     Image
     {
       asynchronous: false
       id: digit
       smooth: true
       source: 'images/digits_'+Logic.digitForPosition(control,index)+'.svg'
       width: control.digitWidth
       height: control.height
       sourceSize.width: width
       sourceSize.height: height
     }
   }
 }

 Image {
   asynchronous: true
   visible: (control.decimalPoint>=0)
   id: point
   source: 'images/digits_point.svg'
   width: control.digitWidth
   height: control.height
   sourceSize.width: width
   sourceSize.height: height
   anchors.bottom: parent.bottom
   anchors.right: parent.right
   anchors.rightMargin: control.decimalPoint*digitWidth
 }
}
```

The small piece of Javascript, Logic.digitForPosition() is separated out into its own file (DigitDisplayFunctions.js) to make the code more readable. Here is the included .js file:

```
.pragma library

function digitForPosition( control, pos )
{
 var stringIndex = pos-(control.places-control.text.length);
 // display string is padded out as [_________012345]
 if ( stringIndex<0 || control.text[stringIndex]=='+')
 {
   return 'digit_'; // default is a blank space
 }
 else
```

```
  {
    return 'digit'+control.text[stringIndex]; // "digit0"..."digit1"
  }
 }
```

The ".pragma library" directive is handy for performance. It tells the QML parser that this file contains functions with no state (class variables). Without this directive, the QML parser would have created one instance of this function for every DigitDisplay in my interface. By using the pragma, all of the DigitDisplay items will share this one function. (This is a more useful optimisation for the javascripts attached to the calculator buttons, as there are sixty of those buttons).

If you fancy using these in your own apps, here are the individual .svg files for the digits, and the display background, as optimised SVG Tiny files: File:Sums-app--displaydigits.zip Have fun!

Performance

Most development was done using the Qt Simulator, because this allowed for a much faster develop/test/debug cycle on my MacOS X development system (direct development on Symbian devices is only provided by the Windows versions of Qt Creator). However, at least once in each day's development, I'd fire off a remote build and install that to the test device to play with. Sometimes what seemed to work well on the desktop turned out to not work well on the phone. One of the most striking differences is in application performance - often, a powerful desktop computer can hide the costs of poor design decisions, but on the limited resources of the phone, there's no escape.

Tip 1. SVG is flexible, but slower than PNG

Producing your graphics in a scalable, vector format like SVG is something that seems like a lot of effort, but if you ever have to re-work an existing UI for a higher DPI display, or change the size of certain elements to make things fit, you'll be glad you did it. Also, vector-based graphics are far more amenable to editing and re-use than images produced with a bitmap tool such as Photoshop or Gimp.

That said, they are not without their price: the biggest performance penalty I paid at startup was waiting for the sixty different SVG files that make up the interface to load. On my test device, an N8, it was over ten seconds from tapping the icon, to seeing the main page. Two seconds of that was the app loading, and then there was eight seconds of black screen while my QML file was parsed and converted into a working screen layout. Maybe eight seconds doesn't sound long to you, but if you're looking at a blank screen, it feels like an eternity.

By trial and error, I discovered that replacing the .svg images I was using for my captions with .png versions shaved about three or four seconds off the load time. But, that's still four seconds of looking at a blank screen. Something was needed to fill the time.

Tip 2. Slower always looks faster if you distract the user

Splashscreens are a great help for hiding a delay. Here was my first attempt. The initial page for the UI page stack contains nothing but an Image element (the splashscreen) and a Loader.

```
 // This is how NOT to do it :)    (some property details have been omitted for clarity)
 Page {
  id: mainPage
  Item
  {
      id: splashscreen;

      Image {
          source: 'images/Sums-icon256.png'
          opacity: 0.3;
      }
  }
  Loader
  {
      id: layoutLoader
      anchors.fill: parent
  }
  onStatusChanged: if (status===PageStatus.Active) { layoutLoader.source =
"MainPageContent.qml"; splashscreen.destroy(); }
  ...
 }
```

The thinking behind this is straightforward. Show the splashscreen image, then load the more complex QML file. But, if you've tried this approach, you might notice something is wrong... you still see nothing - no splash-screen, just a blank page until the big QML file loads. What is happening?

It turns out that I'd made a rookie mistake. I started loading the big QML file as soon as I'd declared my splashscreen image, but loading that file was also blocking my UI updates, so the image never showed. I'm sure this is an issue that will be addressed with a later QML version, but right now, when you load a large file, screen updates can be delayed considerably.

The root cause is that I didn't give QML enough time to update the screen before I started the very intensive job of loading my big QML file, and once the loading had started, the device got too busy to update the screen. Knowing this, the solution is simple: Create a (very) short delay between showing the splashscreen image and starting the loading. The easiest way to achieve this is to place an animation on the splashscreen image, and kick off the loading operation as soon as that animation finishes. So, I fade in the image slightly over a period of 25 ms (this is long enough for QML to update, but not long enough to add noticable time to the loading process).

If I had looked a little more closely at the Flickr demo application, I'd have noticed that it does the same thing!

Here's the revised code:

```
 Item
  {
      id: splashscreen;
      anchors.centerIn: parent
      width: parent.width -10
      height: width*1.2;

      Image {
          anchors.horizontalCenter: parent.horizontalCenter;
          anchors.top: parent.top;
          id: splashicon
          width: 256
          height: 256
          sourceSize.width: 256
          sourceSize.height: 256
          source: 'images/Sums-icon256.png'
          opacity: 0.1
          NumberAnimation on opacity { to: 0.3; duration: 20;
                          onCompleted: { mainPage.tools.tools=pagetools;
layoutLoader.source = "MainPageContent.qml"; splashscreen.destroy(); } }
      }
  }

  Loader
  {
      id: layoutLoader
      anchors.fill: parent
  }
```

The spashscreen appars early, and then the interface loads. Perfect.

By the way, don't be tempted to use a BusyIndicator as part of this splashscreen -- the same lack of screen updates that caused my "no splashscreen" issue will prevent the BusyIndicator spinning and will make it look like your app is hanging.

Summary

The Qt Quick Components for Symbian provide a good tool set for an application. You don't have to worry about hand-rolling basic controls like dialogs and buttons, and you can instead concentrate your creative energy on the bits of UI that differentiate your application from others.

I used quite a lot of custom components in my user interface, but by careful attention to the Belle look and feel, I was able to produce something that didn't look out of place when placed with the existing interface.

However, like any kind of programming toolkit - there's more than one way to do something, and some ways are better than others. When it comes to real-world performance, sometimes what is theoretically the "best" way isn't really the best way to do things.

Note: This is an entry in the Symbian Qt Quick Components Competition 2012Q1