WxSmith tutorial: Drawing on the Screen and Saving Drawings

From Code::Blocks

Contents

- 1 wxSmith Tutorial 8: Drawing on the Screen and Saving Drawings
  - 1.1 Drawing on the Screen
  - 1.2 Save the Picture as PNG and JPEG Files
  - 1.3 An Expandable Graph

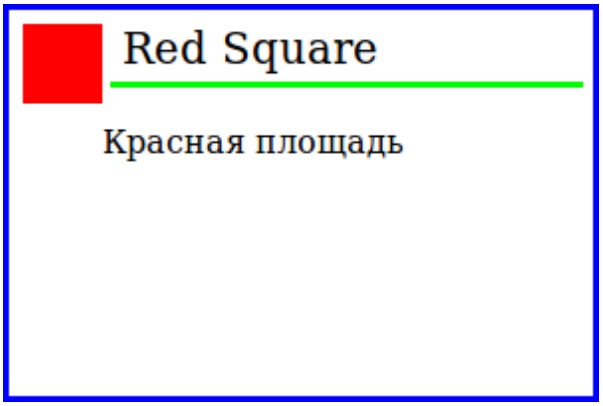
wxSmith Tutorial 8: Drawing on the Screen and Saving Drawings

The main remaining tasks for these tutorials are to show how to

- draw, label, show, and save a graph, and
- how to redirect standard output, as from printf(), to a wxTextCtrl.

We will cover the first of these in this tutorial and the second in the next. Both of these subjects are more a question of using wxWidgets than of using wxSmith itself. However, it is not especially easy to extract the necessary information from other sources, so a unified presentation here may be helpful.

In this tutorial, we will draw the figure shown below on the screen and save it as both a .jpg and a .png file, which can be inserted into word processors such as Writer or MS Word.



(The picture started out as just the red square, but that title suggested the Russian theme, so the name of the square in Cyrillic letters has been added.)

In wxWidgets, one always draws a graph on some kind of Device Context. From the point of view of a programmer using wxWidgets, a Device Context is a black box

which spares us from having to know the details of how to send a graph to a printer, or to the screen, or to a bitmap and on to a .jpg file. Exactly the same code creates the graph for use on all three output devices. Let me illustrate with schematic code for our example.

```
void Repin(wxDC dc){
    … code to draw our picture on any device context …
    }

// When Panel1 gets a “paint” message, draw our picture on it.

wxPaintDC dc( Panel1 );
Repin(dc);

/* To save our drawing to a file, we first create a bitmap, then
        a memory DC, then hand the bitmap to the memory DC to use as
        paper to draw on, then have Repin draw on it, then free the
        bitmap from the DC, then make it write itself as a .jpg file.
        Here are these steps in real code.
*/
// Create a bitmap 300 pixels wide and 200 pixels high.
// Call it "paper" because we will write on it.
   wxBitmap *paper = new wxBitmap( 300,200);

// Create a memory Device Context
  wxMemoryDC memDC;

// Tell memDC to write on “paper”.
  memDC.SelectObject( *paper );

// Call Repin to draw our picture on memDC
  Repin(memDC);

// Tell memDC to write on a fake bitmap;
// this frees up "paper" so that it can write itself to a file.
    memDC.SelectObject( wxNullBitmap );

// Put the contents of "paper” into a png and into a jpeg file.
  wxString wxGraphName = wxString::FromAscii("RedSquare.png");
  paper->SaveFile(wxGraphName, wxBITMAP_TYPE_PNG,(wxPalette*)NULL );
  wxGraphName = wxString::FromAscii("RedSquare.jpg");
  paper->SaveFile(wxGraphName, wxBITMAP_TYPE_JPEG, (wxPalette*)NULL );
// Free the memory claimed for "paper".
  delete paper;
```

One and the same Repin() draws the picture both for output on the screen and for saving in a file.

The SaveFile() function used to save the drawing on disk requires that the name of the file to which the drawing is saved should be in Unicode, not Ascii. The wxString variable wxGraphName will be in Unicode and the wxString function FromAscii() will convert an ordinary C string such as "RedSquare.png" to Unicode. It works also when the argument is a variable name, something declared perhaps as "char s[30]" into which a valid file name has been stored.

Here are the details of the code for Repin(). In reading them, bear in mind that the coordinates of points begin at (0, 0) in the *upper* left corner of the drawing rectangle and then go to the right for the first (or x) coordinate and *down* for the second (or y) coordinate. The comments, plus the result (seen above) should make clear what is happening.

```
void Repin(wxDC &dc){
    // with apology to Russian painter Ilya Repin (1846-1930)
```

```
    // Clear the Device Context to all white
    dc.SetBrush(*wxWHITE_BRUSH);
    dc.Clear();

    // Draw blue border around a white rectangle
    //Create a blue pen 5 pixels wide to draw the border.
    wxColor Blue(0,0,255);
    wxPen myBluePen(Blue,5,wxSOLID);
    // Tell dc to use it
    dc.SetPen(myBluePen);
    // and fill the inside with the current brush,which is white.
    dc.DrawRectangle(0,0,300,200);

    // Set the Brush and Pen to red
    dc.SetBrush( *wxRED_BRUSH );
    dc.SetPen(*wxRED_PEN );
    // Draw rectangle 40 pixels wide and 40 high
    // with upper left corner at 10 , 10.
    dc.DrawRectangle( 10, 10, 40, 40 );

    // Create a green pen 3 pixels wide drawing a solid line
    wxPen myGreenPen(*wxGREEN,3,wxSOLID);
    // Tell dc to start using this pen to draw.
    dc.SetPen( myGreenPen );
    // Draw a horizontal line
    dc.DrawLine( 55, 40, 290, 40);

    // Set the text foreground to black
    dc.SetTextForeground( *wxBLACK);

    // Put some Cyrillic text on the drawing
    dc.DrawText(wxT("Красная площадь"), 50, 60);

    // Create a 16 point, serif font, that is not bold,
    //   not italic, and not underlined.
    wxFont BigFont(16,wxFONTFAMILY_ROMAN,wxNORMAL,wxNORMAL,false);
    // Tell dc to use this font
    dc.SetFont(BigFont);
    // Write the title of our picture.
    dc.DrawText(wxT("Red Square"), 60, 10);
    }
```

wxSmith doesn't help with this coding beyond providing an editor to do it with; it has to be done by hand in C++ using wxWidgets. I will add only that other options for the style of a wxPen, besides wxSOLID, are wxDOT, wxLONG_DASH, wxSHORT_DASH, wxDOT_DASH, and wxTRANSPARENT.

Drawing on the Screen

With Repin ready to paint, let's start a new project, call it Tutorial_8 and remember to add in the Close(). On the main form, put a box sizer; in the sizer put a panel; on the panel put another box sizer; and into it put two buttons, one labeled “Show” and the other labeled “Save”. For showing the graph on the screen, we will need a panel in a frame, so on the Code::Blocks main menu click the wxSmith item and pick “Add wxFrame”. When the window comes up asking for the Class Name and suggesting “NewFrame”, let's instead call it “PictureFrame” just for fun. Accept the other defaults suggested, and finish adding the frame. You will then be greeted by another field of dots, but they represent the new PictureFrame, not the main frame.

Before going further, we must fix up what happens to this frame when the user tries to close it. Double click on the field of dots. The C++ code associated with the frame appears. At the bottom of the file you should see these lines:

```
        void PictureFrame::OnClose(wxCloseEvent& event)
```

```
        {
        }
```

This is where control comes when the user tries to close the PictureFrame window. As you can plainly see, nothing will happen; and the window will hang around until the main window is closed. If, however, we put into the body between the braces the Close() command as before, closing this window will close the whole application. Instead, we must put Destroy(), which will wipe out the present window, but not kill the whole program. So we should have:

```
        void PictureFrame::OnClose(wxCloseEvent& event)
        {
                Destroy();
        }
```

This is a good time to copy Repin() from the above box and place it just below the OnClose block of code.

Now come back to the field of dots (click on PictureFrame.wxs in the bar above the C++ code) and put on it a boxsizer and in the sizer put a panel. This panel is where we will draw our picture for viewing on the screen. In the properties browser, check its Expand box, uncheck Default size, and fill in Width as 310 and Height as 210. (Repin draws a 300 X 200 rectangle, so this gives him a little extra space around edges.)

We now need to add a bit of code for the Paint event for this panel. So click on the panel, click on the {} above the Properties browser, find EVT_PAINT (it should be at the top of the list), click on it, then click on the down arrow at the right edge of the line, and pick Add new handler. Accept the suggested name and click OK.

You find yourself right back in PictureFrame.cpp just below where you put Repin() and presented with the following frame for writing the code to handle this event:

```
void PictureFrame::OnPanel1Paint(wxPaintEvent& event)
{
}
```

We need only add two lines in the middle of the frame, as shown here:

```
void PictureFrame::OnPanel1Paint(wxPaintEvent& event)
{
            wxPaintDC dc( Panel1 );
            Repin(dc);
}
```

The first of those two lines says that dc is going to draw on Panel1 whenever the operating system paints the panel. It will do so when its frame is first displayed, or moved, or resized, or uncovered after being covered. The second line calls Repin to paint for us.

Now we just have to make the Show button on the main window display the panel in PictureFrame. So click on Tutorial_8Frame.wks in the line above the code editor to get back to the main window; go nearly to the top and add under the first group of #include statements

```
#include "PictureFrame.h"
```

so that with the neighboring statements it looks like this:

```
. . .
#include <wx/msgdlg.h>
#include "PictureFrame.h"

//(*InternalHeaders(Tutorial_8Frame)
```

This “include” has to be added because otherwise the main program would not know about the panel it is supposed to open in PictureFrame.cpp.

We already have the “Show” button; we just have to add a handler for its OnClick event. Double click on the button. The frame for adding the event handler for the button opens up and we fill it in as follows:

```
void Tutorial_8Frame::OnButton1Click(wxCommandEvent& event)
{
    PictureFrame* frm = new PictureFrame(this);
    frm->Show();
}
```

At last, we have a program we can build and run. Click the Code::Blocks build-and-run icon, and you should soon see the Repin's work on the screen.

Save the Picture as PNG and JPEG Files

We will save the drawing to a PNG and to a JPEG file in response to a click on the “Save” button. Double click on the button and the frame opens up for the code to respond to the click. Add into the frame the code shown above. The end result is as follows:

```
void Tutorial_8Frame::OnButton2Click(wxCommandEvent& event)
{
/* To save our drawing to a file, we first create a bitmap, then
        a memory DC, then hand the bitmap to the memory DC to use as
        paper to draw on, then have Repin draw on it, then free the
        bitmap from the DC and make it write itself as a .jpg file.
*/
// Create a bitmap 300 pixels wide and 200 pixels high.
// Call it "paper" because we will write on it.
   wxBitmap *paper = new wxBitmap( 300,200);

// Create a memory Device Context
  wxMemoryDC memDC;

// Tell memDC to write on “paper”.
  memDC.SelectObject( *paper );

// Call Repin to draw our picture on memDC
  Repin(memDC);

// Tell memDC to write on a fake bitmap;
// this frees up "paper" so that it can write itself to a file.
    memDC.SelectObject( wxNullBitmap );

// Put the contents of "paper” into a png and into a jpeg file.
  paper->SaveFile( _T("RedSquare.png"), wxBITMAP_TYPE_PNG,
  (wxPalette*)NULL );
  paper->SaveFile( _T("RedSquare.jpg"), wxBITMAP_TYPE_JPEG,
     (wxPalette*)NULL );
```

```
   delete paper;
   }
```

Click the build-and-run icon, click the Save button, start your word processor, and insert either the JPEG or the PNG file.

(for the above to compile successfully, you need to add a function prototype so that Repin() is visible to Tutorial_8Main.cpp. add the line void Repin(wxDC &dc); to the end of PictureFrame.h, before the end header guard. Ensnarer 03.08.2015)

An Expandable Graph

When we clicked the Show button, we could drag the size of the window, but the size of the drawing was not affected. Let's add another button, call it Stretch, and draw a figure which does change size and proportions as the user adjusts the size of the box in which it is displayed. The figure will be a red rectangle with a green border of fixed width.

As before, on the Code::Blocks main menu, pick wxSmith | Add wxFrame; accept the suggested name for the frame, namely NewFrame. Fill in the OnClose code frame with Destroy(), as before. On the frame, put a box sizer; in the sizer put a panel and in the properties browser make the panel 200 wide and 200 high. Be sure to click the Expand property. Change the property browser to the event browser by clicking on the {} icon, find the Paint event, click on the drop-down arrow on the right of the line, and choose “Add new handler”. In the code frame which appears add code to get the following:

```
void NewFrame::OnPanel1Paint(wxPaintEvent& event)
{
    wxPaintDC dc( Panel1 );

    dc.SetPen( wxPen( *wxGREEN, 5 ) ); // Greem pen 5 pixels wide
    dc.SetBrush(*wxRED_BRUSH);

    // Get window dimensions
    wxSize sz = GetClientSize();

    // Our rectangle dimensions
    wxCoord w = sz.x/2 , h = sz.y/2;

    // Center the rectangle on the window, but never
    // draw at a negative position.
    int x = wxMax(0, (sz.x - w)/2);
    int y = wxMax(0, (sz.y - h)/2);

    wxRect rectToDraw(x, y, w, h);
    dc.DrawRectangle(rectToDraw);
}
```

This code has introduced several new wxWidgets constructs: wxSize, wxCoord, wxRect, and the wxMax function. What they mean is clear enough from the name and the way they are used here. The C++ function GetClientSize() is, of course, the key to making the size of the drawing depend on the size of the window it is drawing into.

Now back in the code for the main window, add in the #include "NewFrame.h" line at the the top. Then make wxSmith give you the code frame for the new button labeled “Stretch”. In that code frame add lines to make the whole look like this:

```
void Tutorial_8Frame::OnButton3Click(wxCommandEvent& event)
{
    NewFrame* ffrm = new NewFrame(this);
    ffrm->Show();
 }
```

Click the Code::Blocks build-and-run icon. Click the “Stretch” button. You should get the picture of the red rectangle with a green border. Now try changing the the size of the window. H'mmm – the rectangle does not change. How disappointing! What have we forgotten?

Look back at the properties of the panel on which we drawing. We have Expand checked, so that is not the trouble. Well down the list, there is a cryptic item called Style with a + sign in a box to the left of it. Click on that + sign. A long list of properties appears. The last one is wxFULL_REPAINT_ON_RESIZE. That sounds promising. Check its box, rebuild and run. Click the Stretch button. You should see the image as before. Use the mouse to change the size of the window, and, lo, the image adjusts to the size of the window.

Previous | Index | Next