Flicker-free drawing

Flicker free drawing can be achieved by a two step process.

1. Disable erase background event. When wxWidgets wants to update the display it emits two events: an erase background event and a paint event. You must implement an empty method for the erase background event (in other words: intercept the EVT_ERASE_BACKGROUND event and dont call event.Skip()).
2. Use a double buffer. This means drawing to a bitmap instead of to the display. When drawing is complete, you copy the bitmap to the display. Note that the bitmap must be the same size as the window.

There is the wxBufferedDC class that combines the wxDC functions with buffering to avoid flickering. Even better is wxBufferedPaintDC which is a direct replacement for wxPaintDC. (wxBufferedPaintDC does recreate the bitmap each time. If you unroll the class and only create the bitmap during wxEVT_SIZE events, you can improve performance.)

These two steps work together. If you only disabled the erase event, the display would contain leftovers from the last paint event. Since you also use a bitmap that covers the entire window you will automatically overwrite everything. If you only used a double buffer, you would still see a flash because the window was updated twice: 1st by the erase event, 2nd by the paint event.

See also: an article about flicker-free drawing on Windows: http://www.catch22.net/tuts/flicker

More Detail (no longer relevant)

I was asked in the mailing list to discuss some of my experiences with reducing flicker, so here goes.. Some of this is just the same text from the mailing list, re-edited to be a little bit more clear and accurate..

This is also STRICTLY from a MSW perspective. I am not aware if this is relevent on other platforms.

IF you have flicker, the first thing you want to do is make sure you have the following window styles set on your flickering window, in some cases this solves the problem immediately:

wxNO_FULL_REPAINT_ON_RESIZE wxCLIP_CHILDREN

The first will prevent a full repaint when you resize the screen, instead it will just paint what areas have changed. The second will "CLIP" out the windows that lie on top of the dialog (window) that your working with. For example if you have a dialog with controls on it, you don't want to paint where the controls are, because the controls will just paint over it immediately after. Many times this causes flicker, thus if you clip the controls using this setting, it should be enough.

Randall Fox June 24, 2004 fox_no_spam-usenet001@yahoo.com

Note: For a while now no repaint has been the default. wxWidgets 2.8.7 (haven't tested earlier versions) defines wxNO_FULL_REPAINT_ON_RESIZE as 0.

Controling "CONTROL" flickering

If you try the above and you still get flicker, it might be that the control itself is erasing and painting in rapid succession.

Lets look at the wxStaticText control. This is a popular source of flicker in many applications. The wxStaticText class derives itself from the wxControl class (among others). Looking at the wx source code, you will see that the wxControl class has a function that erases the background, and on many controls, like the wxStaticText, this is not needed.

How does this work? well, when the control (which is essentially a window) goes to "paint" itself, it first erases its window using an "ERASE_BACKGROUND" message and then it goes and paints the text using the DC and standard text printing methods. The thing is, the paint method used by the native control included the erasing of the background automatically when it prints the text, so there really isn't much need to have a separate erase background sequence which is found in the wxControl class. In fact this is what actually causes flicker, wxControl erases, then the native control prints (almost always) the same text over it.

To solve this problem, all you need to do is derive a class based on the wxStaticText control, and override the EraseBackground Event, and DO NOTHING in the function. This prevents calling of the background erase function in the base class, wxControl, and the native control just paints over the old

contents of the control, which is what we want. This method, in fact, works for many controls.. (but not all controls)

Here is an example:

```
class MyStaticTextCtrl : public wxStaticText {
  public:
       MyStaticTextCtrl::MyStaticTextCtrl(wxWindow* parent,
                                wxWindowID id,
                                const wxString& label,
                                const wxPoint& pos,
                                const wxSize& size = wxDefaultSize,
                                long style = 0,
                                const wxString& name= "staticText" ):
       wxStaticText(parent,id,label,pos,size,style,name){};
    void MyStaticTextCtrl::OnEraseBackGround(wxEraseEvent& event) {};
       DECLARE_EVENT_TABLE()
};

BEGIN_EVENT_TABLE(MyStaticTextCtrl,wxStaticText)
    EVT_ERASE_BACKGROUND(MyStaticTextCtrl::OnEraseBackGround)
END_EVENT_TABLE()
```

Randall Fox June 24, 2004 Updated August 16, 2004 fox_no_spam-usenet001@yahoo.com

More information about flicker and MSW

I have researched the issue some more, and found that the above methods work about 80% of the time. Some controls just don't play nice. From now on, we just have to deal with the controls on a case by case basis.

wxTreeCtrl -- Needs the Erasebackground Function in wxControl.. But, this control doesn't have any erase flicker problem, perhaps with the exception of the border of the control, and then when combined with other controls (see below)..

wxGauge -- This control also needs the Erasebackground, and when provided, it will flicker like crazy in some cases.. I have had bad luck with this control.

wxStaticBox -- A static box is a thin line box that you use to visually encase controls. It adds a nice touch to the UI. Unfortunately, I found out that much of my flicker woes were caused by this control. It seems the control is actually a window with a transparent background, and you are placing items on top of the control. The problem is that wx wants you to treat it not as a parent, but as a sibling window/control. The dialog (or parent window) gets confused, and sometimes end up erasing the background behind the control. Clipping the controls work (see above) but the wxStaticBox itself (the thin line) won't stop flickering. On top of that, placing items inside the wxStaticBox causes these controls to act differently with regard to flicker. Thus, if your having problems with flicker and some of the techniques described above don't work, try removing the item from the wxStaticBox, and much of the time this will solve the problem. I haven't found a way to use this control and eliminate flicker in the static lines.

NOTE: All the above information was from memory, I actually did the research a few weeks ago, so if I made a mistake, please correct it.. I believe the information is generally accurate, but your results may be slightly different depending on the wx version and platform you're using.

Randall Fox August 16, 2004 fox_no_spam-usenet001@yahoo.com

If you still have problems

If you're still having flicker problems and all the above techniques don't work, you can try manually clipping the controls. This is really a last ditch effort, and you should know when your going to need it. If you're not quite sure, your probably won't need to use it.

In retrospect, when I initailly used it (described below), I hadn't fully researched the flicker problem. Although it did work for me, there were actually easier ways of accomplishing the same thing.

In my case, I was putting a Panel on top of a dialog. The Dialog was erasing the background (behind the panel) and then the panel was being redrawn. Even though I had the CLIP_CHILDREN style set

for the dialog and even the child panel, I still was getting heavy flicker. In retrospect, the flicker was due to the wxStaticBox. Although this method solved the flicker for all (most?) of the controls inside the static box, the static box itself still flickered.

Anyway, what I did was to manually clip out all the windows on the dialog, to prevent the dialog from painting behind the child windows and controls which were placed on the dialog. You see, I needed to tell the paint routine which area of the window it could safely paint the background color on. I used the erase background event to essentially set the clipping region, which tells the paint routine what part of the dialog needs to be painted in the background color or what NOT to paint.

Below is some code I used to do the above. Please note that this was done on 2.4.2, and there was a work around for a bug, which has been solved in 2.5.1 (or is it 2.5.2 ?)

```
void Dlg_Preferences::OnEraseBackGround(wxEraseEvent& event) {
  wxDC * TheDC = event.GetDC();
  wxColour BGColor =  GetBackgroundColour();
  wxBrush MyBrush(BGColor ,wxSOLID);
  TheDC->SetBackground(MyBrush);

  wxCoord width,height;
  TheDC->GetSize(&width,&height);
  wxCoord x,y,w,h;

  // IIRC, there was a bug in 2.4.1 where the function
  // GetClippingBox didnt work right, so I had to call
  // SetClippingRegion BEFORE calling Get clipping box
  // this is just a work around for a bug.  the Bug should be
  // fixed on wxWidgets 2.5.1 and later, but has not been
  // tested

  TheDC->SetClippingRegion(0,0,width,height);
  TheDC->GetClippingBox(&x,&y,&w,&h);

  // Now  declare the Clipping Region which is
  // what needs to be repainted
  wxRegion MyRegion(x,y,w,h);

  //Get all the windows(controls)  rect's on the dialog
  wxRect TreeRect = ptrTree->GetRect();
  wxRect PanelRect = ptrCurrentPanel->GetRect();
  wxRect Button1Rect = ptrButton_Ok->GetRect();
  wxRect Button2Rect = ptrButton_Apply->GetRect();
  wxRect Button3Rect = ptrButton_Cancel->GetRect();
  wxRect Button4Rect = ptrButton_Help->GetRect();

  // now subtract out the controls rect from the
  //clipping region

  // Perhaps this could be automated by just
  // FINDing each child, and then subtracting
  // its rectangle...   Then you could just drop it
  // into ANY Dialog/Panel and have it work w/o
  // any modification ??

  MyRegion.Subtract(TreeRect);
  MyRegion.Subtract(PanelRect);
  MyRegion.Subtract(Button1Rect);
  MyRegion.Subtract(Button2Rect);
  MyRegion.Subtract(Button3Rect);
  MyRegion.Subtract(Button4Rect);

  // now destroy the old clipping region
  TheDC->DestroyClippingRegion();

  //and set the new one
  TheDC->SetClippingRegion(MyRegion);
```

```
  TheDC->Clear(); // this actually does the erasing
  //event.Skip(); // don't call -- not needed, we already erased the background
}
```

Randall Fox June 24, 2004 Updated August 16, 2004 fox_no_spam-usenet001@yahoo.com

No-flickering for wxListCtrl with wxLC_REPORT | wxLC_VIRTUAL style

```
void OnEraseBackground(wxEraseEvent & event) {
  // to prevent flickering, erase only content *outside* of the
  // actual list items stuff
  if(GetItemCount() > 0) {
     wxDC * dc = event.GetDC();
     assert(dc);

     // get some info
     wxCoord width = 0, height = 0;
     GetClientSize(&width, &height);

     wxCoord x, y, w, h;
     dc->SetClippingRegion(0, 0, width, height);
     dc->GetClippingBox(&x, &y, &w, &h);

     long top_item = GetTopItem();
     long bottom_item = top_item + GetCountPerPage();
     if(bottom_item >= GetItemCount()) {
        bottom_item = GetItemCount() - 1;
     }

     // trick: we want to exclude a couple pixels
     // on the left side thus use wxLIST_RECT_LABEL
     // for the top rect and wxLIST_RECT_BOUNDS for bottom
     // rect
     wxRect top_rect, bottom_rect;
     GetItemRect(top_item, top_rect, wxLIST_RECT_LABEL);
     GetItemRect(bottom_item, bottom_rect, wxLIST_RECT_BOUNDS);

     // set the new clipping region and do erasing
     wxRect items_rect(top_rect.GetLeftTop(), bottom_rect.GetBottomRight());
     wxRegion reg(wxRegion(x, y, w, h));
     reg.Subtract(items_rect);
     dc->DestroyClippingRegion();
     dc->SetClippingRegion(reg);

     // do erasing
     dc->SetBackground(wxBrush(GetBackgroundColour(), wxSOLID));
     dc->Clear();

     // restore old clipping region
     dc->DestroyClippingRegion();
     dc->SetClippingRegion(wxRegion(x, y, w, h));
  } else {
     event.Skip();
  }
}
```

Alternatively you may want to try adding the following after calls to wxListCtrl::InsertItem() and ::SortItems() for instance

```
wxIdleEvent idle;
 wxTheApp->SendIdleEvents(this, idle);
```

This could be of assistance when modifying the contents of wxListCtrl from code (i.e. not as the result of user action generated event) with many (>20) entries. -EL

I have tried this code but still i am getting flickering problem in wxListCtrl code.

Breakthrough: The Great Flicker Wars - A New Hope

I've been working on a solution to the flicker on resize problem on Windows native apps, and I believe I've made something of a breakthrough, although status bars still await a solution. Most other compilers and frameworks on Windows, such as .Net and MFC apps do not flicker so badly. It his claimed that this problem does not show up on Vista, but I have not as yet been able to confirm this. Even so, software developers will still probably be expecting their apps to look nice on XP for some time. A little detail like this could be seen as slightly less professional, even if fancy display is not important; since most modern windows apps flicker on resizing much less noticeably. Most native Win32 apps have this problem as well. The problem seems to lie in the common controls, which the .Net compilers do not usually use. (The problem is claimed to have something to do with HCI guidelines that resulted in the problem.) The good news is, it principally seems to affect static controls. If you convert them to their non-static counterparts, the flickering goes away. This is better than either double-buffering and/or repainting them all programmatically. So far, I've seen no working attempts toward OnPaint() handlers that repaint entire forms and all their children. The above code is the closest I've seen to a general case solution that you can just drop into existing applications. I tried a version of the preceding example, but received only an access violation for my efforts. On the other hand, I've come up with strategies that work for everything except the native status bar. I figure with a special Win32 version of that, 99.5% of all flickering problems will be solved. My method for dealing with flicker is as follows:

1. Turn on NO_FULL_REPAINT_ON_RESIZE and wxCLIP_CHILDREN on the main frame (or presumably on other main windows). This seems to solve the flickering problem for probably most static widgets which do not move on resizing. The flickering vanished e.g. for labels on listbook panels which were absolutely positioned, and therefore didn’t move on resizing, but not for labels that did move. This will probably improve performance slightly, but only solves some of our problems. (I’ve been told that NO_FULL_REPAINT_ON_RESIZE is already the default.)
2. Set the MinSize() of a frame to prevent scrollbars from doing some horrendous flickering as the form is resized (at least on a listbook with icons on the left in my test app). This method is inherited from the basic wxWindow. Alternately, you can set the scrollbars to always be on or off. You don't want a situation in which they can appear or disappear while not under your control as the user resizes the form.
3. The menu and titlebars will flicker on resize, but the effect is relatively unnoticeable on the menus, and even less so on the title bar. These can be disregarded for the short term.
4. Use non-static components to code around the buggy Win32 ones. Use a wxTextControl instead of wxStaticText for elements which do move on resizing. You can make a text box look like a label by setting the foreground and background colors to suitable system defaults for labels, setting it to wxTE_READONLY, using the style wxNO_BORDER, and using SetLabel() to set the text. It would require some research to figure out how to prevent a control from taking the focus by cathing the event, finding out the event, and setting the focus to another control, or alternately, figure out how to make a disabled textbox not display in gray. You can show smaller images by making them a wxBitmapButton with no borders, no events, and no label, and disabling them likewise. Anywhere you have a flickering static control, see if you can replace it with a non-static (i.e. what would normally be a user editable counterpart). As long as the options on the main form are on, these won't flicker anymore.
5. Rely on controls that already use their own methods to paint in order to avoid this problem, such as double buffering or (preferably) bit blit; or write custom paint routines – especially if you have a large amount of data to display, as performance may be an issue. Some existing controls already employ such methods, and so do not suffer from flickering. Bit blit is probably preferable to using a static bitmap anyway, since it is much faster, although some developers (such as those for MAME32) leave it as an option to turn it off, in case it causes problems on some systems. As a practical matter, it is probably fine to do this. The PNG sample program that comes with wxWidgets is a fine example of this technique.
6. Turn off the status bar, or write one that doesn't rely on the Win32 native one. You can see my best attempt to implement one with non-static controls Non-Static Status Bar Attempt.

This leaves us at the status bar. While some people conclude that Win32 will always flicker with common controls, I conclude that only the native status bar still has to, since group boxes can be bitmaps; and so far, I know only of the statusbar that cannot be fixed with these techniques (which is

not to say I've tested tree views, etc). I tried making a status bar class with a wxTextBox on top of an existing field, following the status bar sample, but no matter what I tried, the status bar still flickered. Thus, I intend to develop a new plug-in class that can draw in a panel at the bottom of a sizer to replace the status bar, with the native one turned off in Win32; to be replaced by a custom one made up of native, non-static components. I have not found intercepting EVT_ERASE_BACKGROUND events with null functions to be helpful with other widgets, as they either had no effect or left parts of the screen undrawn. They might yet be helpful with status bars.

Ultimately, this idea of replacing existing static controls with their non-static, custom-painted, blitted and/or double-buffered counterparts could be extended into the wxWidgets library itself, so that a simple recompile would eliminate virtually all noticeable flicker except the status bar – and even a replacement status bar made up of native controls could be added to replace the buggy component in Win32 with just a recompile. Also, the wxStaticBitmap would probably be a more useful standard widget if you could use an additional optional setting, e,g, ::SetDrawType() to tell it to either 1)use static images on all OS's, 2) replace it with a button bitmap on Win32 and use static bitmaps on all other OS's, 3) use bit blits (best), 4)use double buffering. This would save programmers from having to implement custom blit routines in their application code for the most common cases. This would have the effect of giving numerous wxWidgets apps a slightly more professional feel across the board with a mere recompile, and without sacrificing performance by custom drawing everything. The number of widgets that would have to use replacements or special solutions would not be great, as only static controls flicker on Win32, and most are not static; while yet others are already flicker-free due to special ::OnPaint() routines. Any performance difference would be infinitesimal, as this solution only requires using non-static native widgets for static ones. This solution would not affect the look of wxWidgets on other platforms, as it might be a setup.h directive, or might, e.g. have code in the library to use Vista's native Win32 toolbar, if it does not have flicker problems. A programmer would probably have to, at most, drop in a call to a custom status bar routine or setting; if a suitable replacement status bar can be made with wxWidgets calls, as some applications have already done. Even so, Win32 code might be required to make it a work-alike. Code already adapted to non-static widgets should still run after any such library changes, but while code changes to non-static controls in the short term will probably work on other platforms without further code changes, making it a precompile directive only for MSW in the library would help avoid unintended consequences on other platforms without the need for compiler dependent code in the developer's application source.

What follows is a table that summarizes what I've found out so far about flicker-free widgets, which can be extended by any and all interested parties. It refers to widgets prior to the 3.0 build, although I know of no pre-existing plans to make these substitutions. The table will hopefully, eventually include both static and dynamic widgets, including contributed ones, so as to be able to keep track of progress during the great flicker-free initiative. The symbols "--" in a cell refer to untested results, or no ready fix. Remember that testing must be conduced when suitable opions, as above, are enabled on the parent frame/window. If changes in wxWidgets are decided upon, an additional "Library Updated" column should be added to indicate whether "Yes" a widget requires code changes, "No" it needs changes but they have not been made, or "- -" when none are needed.

- JC

Widget	**Already Flicker Free?**	**Replacement**	**Notes**
wxBitmapButton	Yes	None Needed	- -
wxButton	Yes	None Needed	- -
wxCheckBox	Yes	None Needed	- -
wxDatePickerCtrl	Yes	None Needed	- -

exGrid	Yes	None Needed	- -
wxListbook	Yes, With Caution	None Needed	make sure scrollbars cannot pop in and out at will
wxMenuBar	Mostly	- -	effect is negligible
wxPanel	Yes	None Needed	- -
wxSpinCtrl	Yes	None Needed	- -
wxStaticBitmap	No	wxBitmapButton	set attributes to look and act like a static image
wxStaticText	No	wxTextControl	set attributes to look and act like a label
wxStatusBar	No	- -	native control insists on flickering
wxTextCtrl	Yes	None Needed	- -
wxToolbar	Yes	None Needed	- -
wxToolbar separater	Yes	None Needed	- -
wxToolbarButton	Yes	None Needed	- -

The Flicker Wars: Update

After some experimentation, I am less optimistic than I had been. The idea is to come up with work-alikes for static text, bitmaps, etc., which function identically to the originals, but do not flicker. The downside of using wxTextCtrl's for wxStaticText I found out only later, after I wrote the above. With the control set to read only, you can still select it and see a cursor, and highlight its contents. Depending on your needs this may be fine, but it still behaves differently than a label. In my first draft of the above, I thought I could set the color of a wxTextCtrl and set Enable(false). This forces the text to be gray, without special paint routines, no matter what you set the color to, so this will not do, unless there you have some brilliant OnPaint() routine, and lots of event handlers. Even doing it this way, you have the extra overhead of having to maintain an intermediate drawing surface, and thus to have to transfer the bits a second time. Another possible idea is that Win32 controls can be locked. This should prevent it from receiving the focus, but MSVC says I can't access a protected member at compile time. Same for trying to retrieve an hWnd handle. In theory, one can override the wxTextCtrl class to have their own AcceptsFocus() function, but being rather new to wxWidgets and

C++, I was not able to satisfy the compiler in attempting this. One posting I read suggested putting a transparent panel in front of the controls, but 1) wxTRANSPARENT_WINDOW does not work for panels in Win32 (as of XP), 2)nobody's either tried or else succeeded in finding a way to make this work for wxWidgets (and they really should if they can, and 3) whether I put a panel, or an image control in front of the text controls (e.g. containing a single pixel GIF which is transparent and tiled), the OS still allows you do select the underlying controls. The posting suggested the programmer had got it to work, so I may not have been doing it right.

This brings us to one approach that does work. You can set up an event handler to see when a control receives focus and set it to another control, but there are some limitations here. The greatest of them seems to be that although you can do "wxTextCtrl *m_wxWin = (wxTextCtrl *)event.GetEventObject();" in the event handler, and it does work, it seems that the method you might most want to use, to set the wxTextCtrl to wxTE_READONLY, as with wxTRANSPARENT_WINDOW, the IsEditable() property is broken (as of 2.8.3). Getting the Style or the Extra Style still doesn't give you anything you can use to detect wxTRANSPARENT_WINDOW, so these function unexpectedly, as well. Even when a text control is read only, it always returns false. The GetWindow() function promises to tell you which control lost the focus, but doing "event.GetWindow()->SetFocus();" inside the event handler only results in an access violation. There is no event veto implemented for SetFocus, which would have been a nice touch. The only workable approach for the library which we have now (2.8.3) seems to be one of the following: 1) detect the background color to see if it is the same as the parent panel (unless you want such an editable text control, which seems unlikely), or 2) use the ID numbers, such that ID's in one range might be text controls intended to be used as labels, and in another range, they would be editable text controls. You would have to push kill focus event handlers for every control on a form so you could save an object reference to give control back to, and then define OnFocus event handlers for every text control you want to use as a label, to keep it from receiving focus. In the OnFocus event handler, you could test the background color to see if it matches the form, and if so, do a SetFocus to the saved reference of the last control which had the focus, to send it back.

It would become quite tedious to add a handler for every control, to do the job GetWindow is supposed to be doing, although one might think of doing this in the wxWidgets library, so that suitable KillFocus events are sent to it before the programmer's application to save the last window. If one were to revamp the library along these lines, better yet would be to save an object reference you could use with SetFocus in the focus event structure, and then have a new class which the precompiler resolved to when you used wxStaticText, which would send all OnFocus events for that class to the wxWidgets library, so that GetEventObject() could be used in detecting the class of the object, so you could the set the focus back to the other control if needed. Thus, you would add only one handler per event, and do so transparently to the programmer. One might be tempted to use FindFocus to find the window which previously had the focus, but at least in the SetFocus handler, event.GetEventObject() returns not the object which previously had the focus, but the one which received it. Only in the KillFocus event does it represent the one that previously had the focus. It seems the focus has already been changed by the time the SetFocus event has been called, so unless I've missed a way to use GetWindow() to find the last object with the focus, it seems every widget which might have the focus would have to be tracked, if you want clicking on a text box to return the focus to whatever else had it originally. A rather less cumbersome method would be to have each "label" have its own handler, and set the focus to a text box next to it. I'm sure this would be acceptable user interface design, and would probably even be preferable, but it would not function identically with Static Text controls.

One caveat about this strategy is that it might use more GDI resources to use non-static controls than static ones. I cannot for sure confirm or deny this. Even so, by making it a precompiler directive, you could give the programmer the opportunity to choose between esthetics and a small GDI footprint. Some apps might not need many GDI resources, although esthetics might be at a premium. Since Win95, every consumer version of windows MS has put out has had more GDI resourves available, so you can have more programs open at once. Thus, most programmers and users would not find GDI resource usage a significant problem. On the other hand, revamping the library to use a single window that is drawn on and blitted to the screen might ultimately use less GDI resouces and run faster. In that case, custom controls would have to be developed for wxWidgets so as to allow it to draw them via a bit blit. wxWidgets has always been about using native controls on the assumption that they are faster, but a single window with owner drawn controls blitted to the screen could theoretically be faster and use less resources. As above, there are ways to disable controls with the current library, if one is willing to set event handlers for everything that might have the focus, or direct the focus to nearby controls, but right now, programming to disable a control with the 2.8.3 library will probably be somewhat tedious, no matter how the issue is approached.

-JC

One more problem. It seems pretty workable to use wxTextCtrl's for wxStaticText, and prevent the editing of labels by moving the focus to nearby controls, but in order to implement the little grab button in the lower left corner in your own status bar class, you would need to make the raised ridges you see in MS apps a bitmap. I had planned to make it a gif image with black and white lines on it, so that the system background color could show through, so things would display correctly as one changes themes. It seems that the very last things I checked were the very worst problems in trying to do this. My ambition was to use the appropriate sizer cursor arrows over this, and respond to the left mouse button down and up events over it, as well as movements, to make it resize the window as under Windows. The new fly in this ointment is that setting a bitmap button's cursor seems to be broken in wxWidgets. In the "controls" sample, there is a listbox over which the cursor changes to a hand, with this line: "m_listbox->SetCursor(*wxCROSS_CURSOR);". This works as expected. However, adding the line "bmpBtn->SetCursor(*wxCROSS_CURSOR);" to the bitmap button declarations fails to change the cursor over a bitmap button. The manual nowhere warns you of this limitation, so it is unclear if it is a bug, a Win32 limitation, or was designed in for unguessable reasons. Well, I've already come up with quite a number of instances where using wxWidgets leads to quite unexpected results; i.e. it does not work how the manual would lead you to expect. Could somebody else make bug reports on all of these? I've already been asked to, and said I would, but I've spent entirely too long already finding all this out - which is arguably the greater service to the wx community. I guess I'm done evaluating wxWidgets for the time being. I love the rich variety of widgets, and the cross-platform design, but so far have found getting results comparable to other development tools a purely frustrating experience. Also, I have no great love of maintaining my own event handlers for every little action. Ciao!

-JC

Putting it in Perspective

The flicker issue is probably an important one for professional results, but flickering on Windows is not uncommon. This can be seen by observing text on status bars of various MS apps. Under XP, Explorer windows flash like wxWidgets apps. Text on the IE7 status bar does not flash so significantly, although there is a little, related no doubt to video card scanning. This level of flicker is still less than that of static, moveable, common controls as discussed here. It seems that static text on common controls which move on resizing must flash with every little mouse movement. MS Excel in MS Office 2003 flashes significantly less than IE7. MFC applications in general use controls that do not have this liability. wxWidgets uses no commercial libraries, and in general tries to maintain a philosophy of using native controls. In practice, this means the Win32 common controls, which are no longer enhanced with new features (e.g. Visual Studio presents the user with no grid or splitter controls for native mode dialogs on its control palette). Thus, wxWidgets programmers are forced to implement its own versions of such controls. They are also stuck with the liabilities of these lowest common denominator of Windows controls, since wxWidgets drawn controls are (unbelievably) discouraged.

The flicker issue probably should not be considered a stopper, unless the look of an application is at a premium. wxWidgets is fast, as it is in Win32 mode, and not interpreted, as with the new, interpreted code of other MS languages. Other choices are also far from perfect. Borland has their own controls, but bcc is consistantly slower than the MS, MinGW, and the Intel compiler. (I would include the legendary CodeWarrior compiler here too, but the pin-headed bean counters at freescale sold the Windows version to Nokia, who killed it.) The performance computing community, it seems, still awaits a flicker-free toolset on Windows (unless the Vista Common Controls don't do this) that is usable from native mode. - JC

I now think that the flicker problem relates to the native win32 widget sets. I have very little experience in C++, and have just been playing around with MFC. (It seems to take way too much code to get anything accomplished, and seems much more difficult and convenient than wxWidgets.) In MFC, some apps do flicker, and others do not. See: [Flicker-Free Resizing|http://www.codeguru.com/Cpp/W-D/dislog/resizabledialogs/article.php/c1925] for an example. Solving the flicker problem may merely be a matter of using the right widget set, and the right window options in the underlying wxWidget set. With the flicker problem fixed, and a few bug fixes, this could be a very nice development framework for even applications that are relatively visually demanding. -JC

A Separate, "Double-Buffering" Version of wxWidgets?

I had a chat with a fellow on #chandler on the Freenode IRC network who had been recommended to me by somebody else in the channel as someone to talk to regarding flicker and wxWidgets. I had been intending to post a page here regarding another page devoted to one programmer's struggles with this very issue, available here[1]. As it turns out, you can download and install it from here[[2]], and see an example of it yourself. A word of warning here. The program crawls as it starts up, but

this may be related to its use of C++ with Python, or its database structure, and is even a little slow on resizing, but in theory would not be any slower than any other XP program that uses these techniques.

The fellow on #chandler explained that Chandler runs with a version of wxWidgets that has already been modified not to flicker. He says it is a composite of wxWidgets 2.8.2 & 2.8.3 code. XP introduces what is called a Composited Window, and the WS_EX_BUFFERED_DRAW attribute. He says it is not truly double buffered. I think it means each of the controls draws itself, so no need to keep redrawing everything and its children. In theory, it should be faster than the main wxWidgets now, but reality, I think it redraws slower. I say this because I've played around with a number of other frameworks, and noticed some of them had their own solutions to the flicker problem. U++ has an optional flag you can set to "double buffer" (although I'm not sure if it is real double-buffering, or not), although he tries to make it so you don't need it. VCF says it double-buffers the controls by default. DWinLib goes so far as to make all widgets dynamic. MFC SDI wizard projects don't flicker, but I don't know how they accomplish that. In playing around with the VCF framework's examples, I got a chance to play around with cross-platform, native apps running a double-buffered user interface. (Unfortunately, there's no IDE for it yet.) I found that they were acceptably fast, as long as one was not using unusually large numbers of controls in your application. Thus, I'm going to build their version of wxWidgets, and link it to a test application to see if the user interface is indeed as fast as with the VCF. He says he intends to update the wxWidgets main code to offer it. I suggest that it be either a precompiler option, or some other method be used via which it can be disabled, such that people who want the best performance from wxWidgets can leave it off with no real performance penalty, and for those to whom flicker is an issue, they can enable it.

The URL of the modified version of wxWidgets used in Chandler is here[3], and the original version of wxWidgets it is taken from here[4]. The license is a BSD license: free for commercial use. The programmer on #chandler explained that he intended to add some of the basic changes, but he thought some of the changes to the wxGrid might be controversial. It seems to me, as it does not significantly complicate the codebase, that objections to providing this choice should be minimal if it can be disabled, or is off by default. If wxWidgets is to be a world-class platform, it ought not to BLOCK programmers from underlying OS-dependent graphical icing, as GUI these days is becoming increasingly more graphical. If a feature is not supported on a platform, it could simply be ignored. It seems a valid design decision to decide that the look of an application can be slower for the sake of an attractive display, so long as what's under the hood, i.e. running natively in Win32 mode, is not any slower. There have already been several attempts in the SourceForge codebase to fix this very flicker problem, but it seems they are thus far unsuccessful. If the fellow on #chandler doesn't get around to making the updates, I might have time to, (after satisfying myself that wxWidgets then is not slower than the VCF examples on resizing, which is a fair measure of their overall speed), but I'm still nearly a novice in C++, and thus it might take quite a while.

- JC

Freeze/Thaw

Calling Freeze() before updating the sizes or adding controls and then - Thaw() after the update in many cases completely removes the flicker. With some controls this has the opposite effect - immediately after Freeze() - the control's background is drawn on top.

- GeLeTo

Less frequent resizes

When the sizes have to be constantly changed (for instance as a result of dragging the mouse) and the Windows GUI can't cope with the speed of the updates - graphical glitches and an ugly flickering ensues. On MouseDown/Up I start/stop a timer with a 100ms period. Then I do the actual resizing only from the timer's Notify() method. In many cases this completely removes the flicker. Another approach would be not to change the sizes before a new repaint is finished, but I haven't tried this.

- GeLeTo

Actually, I strongly recommend using WS_EX_BUFFERED_DRAW, as above. It is the normal behavior for XP, but it is disabled and completely unsupported in wxWidgets 2.8.3. This seems to solve the problem to the extent it is normally solved on XP. -JC

What Finally Worked

Here is the skinny from the Chandler Project on a special version of wxWidgets they have, that has been modified to remove the flickering. In the file tbar95.cpp, essentially, they have used an #if 0

directive to remove a refresh in the toolbar that they believe was causing flickering - one of the last statements in the function "wxToolBar::HandlePaint":

```
#ifdef 0
control->Refresh(false);
#endif
```

In the file window.cpp, they have added the statement

```
if ( (GetExtraStyle() & wxWS_EX_BUFFERED_DRAW) && (wxApp::GetComCtl32Version() >= 582) )
   *exstyle |= WS_EX_COMPOSITED;
```

to the function wxWindowMSW::MSWGetStyle, just before the statement that says:

```
switch ( border )
```

It is not necessary to use wxNO_FULL_REPAINT_ON_RESIZE and wxCLIP_CHILDREN when this is done.

Now, I think the name they use in their code, wxWS_EX_BUFFERED_DRAW, is inaccurate. This is not a buffered draw, and I believe that calling it that will cause some wxWidgets programmers to avoid it. Double-buffering can slow down GUI, but instead, this XP window attribute tells the OS that some of the widgets redraw themselves. As such, I'd suggest it be called wxWS_EX_COMPOSITED_WINDOW. Alternately, the attribute wxNO_FULL_REPAINT_ON_RESIZE could be made to behave as it appears it would, and go ahead and define this attribute too, since it does turn off full repainting on resize. Apparently, it is drawing by the system, followed by an additional round of drawing by the widgets themselves that cause the flickering. In theory, it should be faster with fewer redraws. One wxWidgets programmer I chatted with said that it would be an excellent idea to make this the default for all wxWidgets. I am inclined to agree with him, but add the caution that since wxWidgets is designed to be a performance-oriented GUI, it might also theoretically slow things down. For those who are so inclined, they might test it by timing multiple iterations of window refreshes both with and without, to be sure, and if it is indeed faster, then include it as standard for MSW. wxWidgets applications would begin to flicker less everywhere, next time they are recompiled.

Here are the complete diffs for the Chandler Project files. Please note that I downloaded and tried out the Chandler application, and it is SLOOOOOWWWWW, even just to resize a window. I don't know if that has to do with some database they are using, or other changes to the graphics primitives. Be that as it may, when making these changes to my local version of wxWidgets, and recompiling, my resulting application was as fast, e.g. during a window resize, as any wxWidgets sample application. Even so, it eradicated nearly all traces of flickering on resizing, even though some of my non-static components that changed position during resizing (due to the sizers) had flickered badly beforehand, as well as the status bar. There is a small amount of residual flicker during resizing in e.g. the title bar, which other XP apps share, but is noticeable, and barely visible when it is. It can be safely disregarded, IMHO. While wxWidgets has a number of innovative GUI features, it seems it will only gain this most basic of XP attributes only after Vista has already been released. I encourage someone to add this to SVC or CVS or whatever you guys use. I'm still learning C++, being pretty new to it, and haven't gotten as far as selecting a revision control system for myself. (I used the Polytron PCVS back in the day when I programmed in good ole "C".) Now come on guys, and add some nice, up-to-date visual icing like gradients, everywhere an application like Visual Studio might use them, and the nice little grey gradient line down the left of their menu's, so I don't have to do everything to make wxWidgets applications look modern. (wxAUI was certainly a milestone in that department, however.) After all guys, I'm still nearly a newby at C++!

For us programmers who use wxWidgets in win32, it may be a matter of chasing the look they are getting with their managed code projects, since they are not updating their native win32 code similarly. Thus, the philosophy of using native controls for wxWidgets may leave us using primitive-looking win32 controls, unless we tweak them. Still, wxWidgets is a good thing to use, IMHO. True, GUI will be slower with a system independent wrapper like wxWidgets, but then, for that matter, MFC is a wrapper for win32 also. Nobody ever complains that MFC applications are too slow. For windows programmers, it is not a matter of adding a wxWidgets wrapper, but *replacing* MFC with wxWidgets, and gaining platform independence (unless it is managed wxWidgets, of course).

Here is a complete listing of the diffs (from 'fc') between the wxwidgets 2.8.3 files (with the .original extension) and the Chandler ones. Most of the differences are due to other changes the Chandler team made to the source, and can be disregarded. The fellow who had made the changes said the flicker-free changes were in these two files, so I've included the diffs, in case I've missed anything:

```
Comparing files tbar95.cpp and TBAR95.CPP.ORIGINAL
```

```
***** tbar95.cpp
 541:
 542: //
 543: // OSAF: recommendations
 544: // - for efficiency, determine if MapBitmap needs to be called for disabled bitmaps
 545: // - for efficiency, hoist and retain return value for wxApp::GetComCtl32Version()
 546: // - for visual fidelity, determine if SetBackground(192 x 3) is proper
 547: // - for code clarity, retain value for wxSystemOptions::HasOption(wxT("msw.remap"))
 548: // - for code safety, calls to bmp.GetWidth, etc. should happen after bmp.OK()
 549: // - check MSDN for best practices
 550: //
 551: bool wxToolBar::Realize()
***** TBAR95.CPP.ORIGINAL
 541:
 542: bool wxToolBar::Realize()
*****

***** tbar95.cpp
 557:
 558:     bool doRemap, doRemapBg, doTransparent;
 559:     doRemapBg = doRemap = doTransparent = false;
 560:
 561: #ifndef __WXWINCE__
 562:     int remapValue = (-1);
 563:     const wxChar *remapOptionStr = wxT("msw.remap");
 564:     if (wxSystemOptions::HasOption( remapOptionStr ))
 565:         remapValue = wxSystemOptions::GetOptionInt( remapOptionStr );
 566:
 567:     doTransparent = (remapValue == 2);
 568:     if (!doTransparent)
 569:     {
 570:         doRemap = (remapValue != 0);
 571:         doRemapBg = !doRemap;
 572:     }
 573: #endif
 574:
***** TBAR95.CPP.ORIGINAL
 548:
 549: #ifdef wxREMAP_BUTTON_COLOURS
 550:     // don't change the values of these constants, they can be set from the
 551:     // user code via wxSystemOptions
 552:     enum
 553:     {
 554:         Remap_None = -1,
 555:         Remap_Bg,
 556:         Remap_Buttons,
 557:         Remap_TransparentBg
 558:     };
 559:
 560:     // the user-specified option overrides anything, but if it wasn't set, only
 561:     // remap the buttons on 8bpp displays as otherwise the bitmaps usually look
 562:     // much worse after remapping
 563:     static const wxChar *remapOption = wxT("msw.remap");
 564:     const int remapValue = wxSystemOptions::HasOption(remapOption)
 565:                            ? wxSystemOptions::GetOptionInt(remapOption)
 566:                            : wxDisplayDepth() <= 8 ? Remap_Buttons
 567:                                                    : Remap_None;
 568:
 569: #endif // wxREMAP_BUTTON_COLOURS
 570:
*****

***** tbar95.cpp
 614:
 615: #ifndef __WXWINCE__
```

```
 616:         if (doTransparent)
 617:             dcAllButtons.SetBackground(*wxTRANSPARENT_BRUSH);
 618:         else
 619:             dcAllButtons.SetBackground(wxBrush(GetBackgroundColour()));
 620: #else
 621:         dcAllButtons.SetBackground(wxBrush(wxColour(192,192,192)));
 622: #endif
 623:         dcAllButtons.Clear();
 624:
***** TBAR95.CPP.ORIGINAL
 610:
 611: #ifdef wxREMAP_BUTTON_COLOURS
 612:         if ( remapValue != Remap_TransparentBg )
 613: #endif // wxREMAP_BUTTON_COLOURS
 614:         {
 615:             // VZ: why do we hardcode grey colour for CE?
 616:             dcAllButtons.SetBackground(wxBrush(
 617: #ifdef __WXWINCE__
 618:                                        wxColour(0xc0, 0xc0, 0xc0)
 619: #else // !__WXWINCE__
 620:                                        GetBackgroundColour()
 621: #endif // __WXWINCE__/!__WXWINCE__
 622:                                       ));
 623:             dcAllButtons.Clear();
 624:         }
 625:
*****

***** tbar95.cpp
 627:
 628: #ifndef __WXWINCE__
 629:         if (doRemapBg)
 630:         {
***** TBAR95.CPP.ORIGINAL
 628:
 629: #ifdef wxREMAP_BUTTON_COLOURS
 630:         if ( remapValue == Remap_Bg )
 631:         {
*****

***** tbar95.cpp
 639:         }
 640: #endif // !__WXWINCE__
 641:
***** TBAR95.CPP.ORIGINAL
 640:         }
 641: #endif // wxREMAP_BUTTON_COLOURS
 642:
*****

***** tbar95.cpp
 684:
 685:                     if (doRemap)
 686:                     {
***** TBAR95.CPP.ORIGINAL
 685:
 686: #ifdef wxREMAP_BUTTON_COLOURS
 687:                     if ( remapValue == Remap_Buttons )
 688:                     {
*****

***** tbar95.cpp
 700:                     }
 701:
***** TBAR95.CPP.ORIGINAL
```

```
 702:                 }
 703: #endif // wxREMAP_BUTTON_COLOURS
 704:
*****

***** tbar95.cpp
 705:
 706:                 if (doRemap)
 707:                     MapBitmap(bmpDisabled.GetHBITMAP(), w, h);
 708:
***** TBAR95.CPP.ORIGINAL
 708:
 709: #ifdef wxREMAP_BUTTON_COLOURS
 710:                 if ( remapValue == Remap_Buttons )
 711:                     MapBitmap(bmpDisabled.GetHBITMAP(), w, h);
 712: #endif // wxREMAP_BUTTON_COLOURS
 713:
*****

***** tbar95.cpp
 724:
 725:         if (doRemap)
 726:         {
***** TBAR95.CPP.ORIGINAL
 729:
 730: #ifdef wxREMAP_BUTTON_COLOURS
 731:         if ( remapValue == Remap_Buttons )
 732:         {
*****

***** tbar95.cpp
 728:             hBitmap = (HBITMAP)MapBitmap((WXHBITMAP) hBitmap,
 729:                 totalBitmapWidth, totalBitmapHeight);
 730:         }
 731:
***** TBAR95.CPP.ORIGINAL
 734:             hBitmap = (HBITMAP)MapBitmap((WXHBITMAP) hBitmap,
 735:                                          totalBitmapWidth, totalBitmapHeight);
 736:         }
 737: #endif // wxREMAP_BUTTON_COLOURS
 738:
*****

***** tbar95.cpp
1350:
1351: //void wxToolBar::SetToolLabel(int id, const wxString& label)
1352: //{
1353: //    wxToolBarTool* tool = wx_static_cast(wxToolBarTool*, FindById(id));
1354: //    if ( tool )
1355: //    {
1356: //        wxCHECK_RET( tool->IsButton(), wxT("Can only set label on button tools."));
1357: //
1358: //        tool->SetLabel(label);
1359: //        Realize();
1360: //    }
1361: //}
1362:
1363: // ----------------------------------------------------------------------------
***** TBAR95.CPP.ORIGINAL
1357:
1358: // ----------------------------------------------------------------------------
*****

***** tbar95.cpp
1588:
```

```
 1589:                 // As it turns out, the following code causes the toolbar to need another
 1590:                 // paint, which causes a Refresh, which causes another paint .... whenever
 1591:                 // we're drawing in a window with WS_EX_COMPOSITED style.
 1592:                 //
 1593:                 // I suspect that the style refered to below is the one that causes only
 1594:                 // the newly exposed space from sizing a window to be drawn. If so I don't
 1595:                 // see why refreshing the control is necessary in that case, since the
 1596:                 // control's location doesn't change when the window is resized. Also,
 1597:                 // when testing on Windows XP, if I don't call Refresh on the control,
 1598:                 // I don't see any display problems (either with or without
WS_EX_COMPOSITED).
 1599:                 // However erasing the rectangle above is necessary.
 1600:                 //
 1601:                 // So for now, I'm disabling this code for the Chandler Alpha 0.3 release
 1602:                 // since this is the last blocking bug. I will follow up with Julien
 1603:                 // before finishing this task and declaring the bug fixed -- John Anderson
 1604:  #if 0
 1605:                 // Necessary in case we use a no-paint-on-size
***** TBAR95.CPP.ORIGINAL
 1583:
 1584:                 // Necessary in case we use a no-paint-on-size
*****

***** tbar95.cpp
 1607:                 control->Refresh(false);
 1608:  #endif
 1609:             }
***** TBAR95.CPP.ORIGINAL
 1586:                 control->Refresh(false);
 1587:             }
*****

***** tbar95.cpp
 1669:
 1670:  WXHBITMAP wxToolBar::MapBitmap(WXHBITMAP bitmap, int width, int height)
***** TBAR95.CPP.ORIGINAL
 1647:
 1648:  #ifdef wxREMAP_BUTTON_COLOURS
 1649:
 1650:  WXHBITMAP wxToolBar::MapBitmap(WXHBITMAP bitmap, int width, int height)
*****

***** tbar95.cpp
 1689:
 1690:   #if 0
 1691:       // OSAF (06-Nov-05):
 1692:       // 1) this block ruins the appearance of icons with gradients,
 1693:       // so disable it until we get an explanation from the wx group.
 1694:       // 2) the triple-nested loops are computationally expensive,
 1695:       // the effect of which is noticeable in Chandler. Note also that
 1696:       // this block gets called for disabled bitmaps regardless of the
 1697:       // value returned by
 1698:       // wxSystemOptions::GetOptionInt(wxT("msw.remap"))
 1699:     wxCOLORMAP *cmap = wxGetStdColourMap();
***** TBAR95.CPP.ORIGINAL
 1669:
 1670:     wxCOLORMAP *cmap = wxGetStdColourMap();
*****

***** tbar95.cpp
 1713:             {
 1714:                 ::SetPixel(hdcMem, i, j, cmap[k].to);
 1715:                 break;
***** TBAR95.CPP.ORIGINAL
 1684:             {
```

```
                if ( cmap[k].to != pixel )
                    ::SetPixel(hdcMem, i, j, cmap[k].to);
                break;
*****

***** tbar95.cpp
    }
#endif

***** TBAR95.CPP.ORIGINAL
    }

*****

***** tbar95.cpp
    return bitmap;

    // VZ: I leave here my attempts to map the bitmap to the system colours
    //     faster by using BitBlt() even though it's broken currently - but
    //     maybe someone else can finish it? It should be faster than iterating
    //     over all pixels...
#if 0
    MemoryHDC hdcMask, hdcDst;
    if ( !hdcMask || !hdcDst )
    {
        wxLogLastError(_T("CreateCompatibleDC"));

        return bitmap;
    }

    // create the target bitmap
    HBITMAP hbmpDst = ::CreateCompatibleBitmap(hdcDst, width, height);
    if ( !hbmpDst )
    {
        wxLogLastError(_T("CreateCompatibleBitmap"));

        return bitmap;
    }

    // create the monochrome mask bitmap
    HBITMAP hbmpMask = ::CreateBitmap(width, height, 1, 1, 0);
    if ( !hbmpMask )
    {
        wxLogLastError(_T("CreateBitmap(mono)"));

        ::DeleteObject(hbmpDst);

        return bitmap;
    }

    SelectInHDC bmpInDst(hdcDst, hbmpDst),
                bmpInMask(hdcMask, hbmpMask);

    // for each colour:
    for ( n = 0; n < NUM_OF_MAPPED_COLOURS; n++ )
    {
        // create the mask for this colour
        ::SetBkColor(hdcMem, ColorMap[n].from);
        ::BitBlt(hdcMask, 0, 0, width, height, hdcMem, 0, 0, SRCCOPY);

        // replace this colour with the target one in the dst bitmap
        HBRUSH hbr = ::CreateSolidBrush(ColorMap[n].to);
        HGDIOBJ hbrOld = ::SelectObject(hdcDst, hbr);

        ::MaskBlt(hdcDst, 0, 0, width, height,
```

```
 1772:                 hdcMem, 0, 0,
 1773:                 hbmpMask, 0, 0,
 1774:                 MAKEROP4(PATCOPY, SRCCOPY));
 1775:
 1776:         (void)::SelectObject(hdcDst, hbrOld);
 1777:         ::DeleteObject(hbr);
 1778:     }
 1779:
 1780:     ::DeleteObject((HBITMAP)bitmap);
 1781:
 1782:     return (WXHBITMAP)hbmpDst;
 1783:  #endif // 0
 1784:  }
***** TBAR95.CPP.ORIGINAL
 1693:     return bitmap;
 1694:  }
*****

***** tbar95.cpp
 1785:
 1786:  #endif // wxUSE_TOOLBAR
***** TBAR95.CPP.ORIGINAL
 1695:
 1696:  #endif // wxREMAP_BUTTON_COLOURS
 1697:
 1698:  #endif // wxUSE_TOOLBAR
*****

Comparing files window.cpp and WINDOW.CPP.ORIGINAL
***** window.cpp
 1377:
 1378:         if ( (GetExtraStyle() & wxWS_EX_BUFFERED_DRAW) &&
(wxApp::GetComCtl32Version() >= 582) )
 1379:             *exstyle |= WS_EX_COMPOSITED;
 1380:
 1381:         switch ( border )
***** WINDOW.CPP.ORIGINAL
 1377:
 1378:         switch ( border )
*****
```

P.S. to Above

I forgot to mention that the above code also needs a change in include/wx/defs.h. They also add the lines

```
/*  An extended style that causes the window to be draw depth-first */
/* and buffered. Currently implemented only on Windows XP */
#define wxWS_EX_BUFFERED_DRAW          0x00000040
```

just before the line

```
#define wxWS_EX_PROCESS_UI_UPDATES      0x00000020
```

I noticed this when I tried to start with a fresh wxWidgets and recompiled. By the way, upon reviewing the MS description of their attributes, I believe wxWS_EX_BUFFERED_DRAW is an excellent name for a new wxWidgets window attribute. As it double-buffers, it could be slower. It might be faster at times, as it does not have to redraw everything so much - i.e. only the exposed widgets. As such, it might worthwhile to get other opinions, but if wxWidgets is to be a performance oriented toolkit, it might be better to leave it off by default. I plan to use it on virtually all my stuff.

New setting coming: SetDoubleBuffered()

My original request, here has given rise to changesets here and[5] that have patches which will presumably make it into the next release, describing a new wxWindow setting you can make for MSW, called SetDoubleBuffered(), which sets the WS_EX_COMPOSITED attribute, so that Windows XP "Paints all descendants of a window in bottom-to-top painting order using double-buffering."[6] and adjusts the toolbars for flickering. This rolls up all the important techniques from this webpage, and will make them obsolete when it is available and working. In order to make a slick looking, non-flickering app, you should always call it when it becomes available. I did not write the patch, but it presumably depends heavily on my posts above and the issue it submitted. The wxChandler solution of using

```
#define wxWS_EX_BUFFERED_DRAW        0x00000040
```

is not as good, since it must share this attribute bit with the Mac metallic attribute. While in practice this would cause few problems, it might cause some ported code to look metallic unexpectedly. Oddly enough, it is in the 2.8.9 release of wxWidgets, and I updated my window.cpp with the wxChandler changes involved from above, not realizing this. My old source, which used the wxChandler technique, worked fine, and had eliminated the flicker. I tested the SetDoubleBuffered(true) function, to replace my SetExtraStyle(wxWS_EX_BUFFERED_DRAW); statement, but although the code looked like it should work just fine, IT DID NOT! Things still flicked while resizimg my app, signifying that the attribute had not been successfully set. Commenting it out and restoring my original statement: "SetExtraStyle(wxWS_EX_BUFFERED_DRAW);" returned my app to a noticeably improved, basically flicker-free status. Either SetDoubleBuffered() has been disabled in some way for this release (although it compiled and ran fine), or the feature that was added is buggy. Stay tuned, all ye loyalists of the great flicker rebellion! Once this new feature is working right, maybe we can get gradient backgrounds on menus, toolbars and tabs, regardless of themes (in spite of the no owner-drawn controls policy); the scrollbar overlap problem fixed; and good charting. With these done, wxWidgets may well be on its way to being a framework second to none, or perhaps solid competition for the VCF with better design tools.