Add animation effects to your forms

The Windows GUI is full of graphical effects: roll, fade and popups make the interface attractive.

If you want apply the same effects to your forms you need only one API function: *AnimateWindow*!

This API is simple to use and you only supply the handle of your form, how long the animation should take in milliseconds and a bunch of flags to specify effects and directions.

Try placing this code in the *OnShow* handler of your form:

```
...
AnimateWindow(Self.Handle, 250, AW_BLEND or AW_ACTIVATE);
...
```

Run the project and see the result. Cool, it isn't?

By default the function uses a roll effect, but you can change this using *AW_SLIDE* to obtain a slide effect, *AW_CENTER* to make the window collapse or expand or *AW_BLEND* to obtain a smooth fade effect.

In addition you can add the *AW_ACTIVATE* flag to specify that the form is appearing or *AW_HIDE* if you're hiding your form; this flag reverses the direction of the animation.

When you use the roll or slide you can specify the direction of the effects along the horizontal and verticale axis adding the *AW_HOR_POSITIVE* or *AW_HOR_NEGATIVE* for x-axis and *AW_VER_POSITIVE* or *AW_VER_NEGATIVE* for y-axis.

All these flags are ignored when you use the *AW_CENTER* flag.

Experiment various combinations of flags and see the results!

After you've played for a while you will notice a quirk: some controls appears correctly, others are drawn erroneously and some other are not drawn at all!!

My first thought was: "Damn! It doesn't work". My second thought was: "Why?"

Well the reason is quite simple but the solution is boring.

The MSDN documentation states that the windows procedures for the controls on a form may need to handle the *WM_PRINT* or *WM_PRINTCLIENT* messages to be used in conjuction with the *AnimateWindow* API. Those messages are used when Windows needs to draw the contents of a control on a display context different from the screen (e.g. the printer or, as in our case, an offscreen bitmap). The documentation remarks that the window procedures for controls, common controls and dialog boxes already handle the messages.

This explain why some controls are drawn correctly: *TButton*, *TCheckBox* and *TRadioButton*, for example, own and manage an underlying button control, therefore they handle messages correctly.

The garbled controls are a middle species since they own a Windows control but some parts are drawn using the Delphi *TCanvas* and not the Windows GDI: *TEdit*, *TRichEdit* and others behave that way.

The invisible controls are the ones who does not own a window ar all and are all the *TGraphic* descendants like *TShape* and *TBevel*: these controls will never be drawn because they will never get the necessary messages.

Anyway, for every Delphi component descending from *TWinControl* you can intercept and handle the *WM_PRINTCLIENT* message in this way:

```
unit TestGroupBox;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TTestGroupBox = class(TGroupBox)
  protected
    procedure WMPrintClient(var Msg: TMessage); message WM_PRINTCLIENT;
```

```
 end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Test', [TTestGroupBox]);
end;

{ TTestGroupBox }

procedure TTestGroupBox.WMPrintClient(var Msg: TMessage);
begin
  PaintTo(HDC(Msg.WParam), 0, 0);
end;

end.
```

This is a sample component derived from *TGroupBox*. Although *TGroupBox* owns a window, it is not a common control window, therefore the *WM_PRINTCLIENT* message is not handled. Adding the message handler and using the *PaintTo* method to draw the control on a different display context is straightforward.

And now the boring part: this must be done for every control you want to use on you forms; moreover you must change all your controls to the new ones before you can safely use the *AnimateWindow* API.

Enjoy!

Author:	Unknown
Contributor:	Riccardo Faiella (Topellina)
Added:	2012-07-06
Last updated:	2012-07-06