Simple Screen Savers

by Paul Warren

There is nothing like a screen saver to bring out the frustrated artist in me. Screen savers give me a chance to experiment with graphics, animations and sound without worrying about things like deadlines. They are pure fun. Unfortunately, I've been asked to write custom screen savers often enough that I'm getting tired of coding them from scratch. I had to develop an automated coding system and some animation components to restore the fun.

I still have to support more Windows 3.1 systems than Win95 so I generally develop in Delphi 1 and port to Delphi 2 but for this article I will show 32-bit code examples. If you are working in Delphi 1 don't worry, 16-bit code is included on this month's disk too.

The Screen Saver Application

The first screen saver I ever wrote used the OWL framework from Borland Pascal 7.0. It was a crude and unsuccessful effort which served only to start me on the long road to useful animations.

The next couple of screen savers used Delphi but relied on the same Windows API routines I had used in the original. One big difference Delphi made for me though, was to simplify the application framework. This allowed me to concentrate on improving the animations.

The easiest part of creating a Delphi screen saver is the application framework. All you need to do is create a new blank project, set a few properties for the main form, add a couple of lines of code and you have a nearly complete, albeit boring, screen saver.

I generally make the main form small so as not to clutter the desktop. Set the `BorderStyle` property to none, `Color` to something pleasing and `FormStyle` to `fsNormal`. In the `OnCreate` method I size and position the form to the screen. I don't use

```
WindowState := wsMaximized
```

because the form doesn't maximize properly in Win95. Next, I give the application a title, by convention this is

```
Screen Saver.SomethingDescriptive
```

Since Windows is going to activate your screen saver periodically you must check for any previous instance of the application before allowing a new instance to run, otherwise you could end up with many active copies after an extended period of inactivity. The easiest way to check for another instance of the application in Delphi 1 is to insert the line

```
if HPrevInst <> 0 then Exit;
```

in the project (.DPR) file after the first `begin`. Things are more complicated in Win95. I use code from *The Delphi 2 Developers Guide*, page 907, to determine if another instance of my app is running. Listing 1 shows the complete project source file for the screen saver application.

Note the second line of Listing 1. This is necessary to tell Windows that the application is a screen saver. Delphi's Help describes the Description Directive as "Inserts the specified text into the module description entry in the header of an EXE file or DLL."

Windows uses a command line parameter to tell your screen saver when the user has clicked the setup button. The command line parameter is `-C` or `/C` or just `C`. In the simplest implementation the Setup Form is just an About box. For 32-bit screen savers Win95 issues a second parameter `/P` or `P`, the preview parameter. The simplest way to respond to this one is to `Exit`. Many thanks to Bob Swart for the info on this parameter.

Finally you have to add event handlers for `KeyDown`, `MouseDown` and `MouseMove` events to terminate the screen saver when activity resumes. Optionally, and certainly by convention, you can turn off the cursor when your screen saver starts and turn it on when it terminates. You should also capture the mouse to your main form using `SetCaptureControl(Self)`.

Believe it or not, after compiling the program and changing the .EXE

➤ *Listing 1*

```
program SBASIC32;
{$D SCRNSAVE: Screen Saver.Boring}
uses
  Windows, Forms, SysUtils,
  Main in 'Main.pas'   {SaverForm},
  About in 'About.pas' {AboutBox};
{$R *.RES}
var
  CmdLine: String;
  MutHandle: THandle = 0;
begin
  MutHandle := OpenMutex(MUTEX_ALL_ACCESS, False, 'Screen Saver.Boring');
  if MutHandle = 0 then
    MutHandle := CreateMutex(nil,false,'Screen Saver.Boring')
  else Exit;
  CmdLine := UpperCase(ParamStr(1));
  if (CmdLine  = '/P') or (CmdLine = '-P') or (CmdLine = 'P') then Exit;
  if (CmdLine  = '/C') or (CmdLine = '-C') or (CmdLine = 'C') then begin
    AboutBox := TAboutBox.Create(Application);
    AboutBox.ShowModal;
    AboutBox.Free;
  end else begin
    Application.Title := 'Screen Saver.Boring';
    Application.CreateForm(TSaverForm, SaverForm);
    Application.Run;
  end;
end.
```

file's extension to .SCR you now have a working screen saver. You'll find complete source on the disk as SBASIC16.DPR and SBASIC32.DPR.

A Screen Saver Expert

Even the small amount of coding needed to create a screen saver application framework can be avoided by writing an expert to do the job. Normally this would be time consuming but if you read my article in Issue 13 you know I have an expert-creating expert that's just right for the job. Naturally I used this to create my screen saver expert.

16-bit and 32-bit experts are included on the disk. You could also install the basic screen saver application in the Delphi 1 Gallery or Delphi 2 Object Repository.

Extending The Screen Saver

Obviously you would quickly get tired of a plain colored screen appearing after an idle period. There are a number of simple things you can do to improve the situation. You could put a `TImage` on the main form and set the `Picture` property to your favorite bitmap, or you could paint the form with a patterned brush in the `OnShow` method. But let's face it, without animations a screen saver is boring.

In Issue 1 Xavier Pacheco demonstrated some useful sprite animation techniques. I decided to use the same general methods, but I wanted more re-useable code; in other words my animation system would be written as components.

The TSprite

My `TSprite` component, descended from `TGraphicControl` via `TCustomSprite`, has `ANDImage` and `ORImage` properties to hold an image and mask. To draw the sprite you have to copy the `ANDImage` to a canvas using the bitwise `AND` operator. You then copy the `ORImage` using the `OR` operator. This procedure was explained very well by Xavier Pacheco so I won't go into detail *[You can download an electronic version of Issue 1 from our Web site. Editor]*. Suffice it to say that copying the two images this way allows the background to show through wherever the `ANDImage` mask is drawn.

I wanted to do the drawing to `TSprite`'s `Parent.Canvas` but this won't work. In the Delphi VCL, `Canvas` properties are declared `protected` and are read only. `TSprite` can't access the `Canvas` property of its parent. You can create a new component, say a `TXPanel`, and re-declare its `Canvas` as `public` (read/write), but then you can only drop a `TSprite` on a `TXPanel`. Any component that causes an error when you drop it on the wrong component wouldn't be well received.

One possible solution is to `AND` and `OR` the images to the `TSprite`'s own canvas and actually move the component to animate it. This would be an elegant solution since the sprite could be animated at design time. I actually got a sprite working this way but here Delphi itself let me down (for the first time). The problem was that I was drawing each image directly to the canvas and with each drawing Delphi makes two `Invalidate` calls. This makes four drawings for each move of the sprite. As you can imagine, for all but the smallest sprites the flicker was something to behold.

I finally settled on a sort of hybrid component for the working system. I wrote a `Paint` method (see Listing 2) that would copy the image to the component's `Canvas` at design time. At run time though, `Paint` would do nothing. I would control the drawing from another component, a sort of drawing surface. This way I felt I could achieve smooth animations and still have a component which would behave properly when designing.

Since the `Left` and `Top` properties hold the design time location of `TSprite` I added `SLeft` and `STop` properties to hold the location of the moving sprite at run time. This stopped some strange behavior: the sprite would be at a different location after running the application from the IDE.

There is also a `MoveSprite` `public` method which, when called, changes the sprite's `SLeft` and `STop` properties in the direction of the `VX` and `VY` accelerator properties. `MoveSprite` is the animation engine.

The TSpriteBox Drawing Surface

`TSpriteBox` is a drawing surface for `TSprite`. I tried descending from a `TPaintBox` initially, hence the name. I gave the component `BackGnd1` and `BackGnd2` properties of type `TBitMap` to handle the off-screen drawing and a field `FSprite` of type `TSprite`. In the `Create` constructor there are calls to create the `BackGnd` bitmaps. The destructor looks after freeing the `BackGnd` bitmaps. To handle the animation I added a `DrawSprite` method using Windows API routines to draw the sprite off-screen. Finally, the `Paint` method needed an override to copy `BackGnd1` to the `Canvas` whenever a `WM_PAINT` message gets processed (see Listing 3).

The next step is to size the bitmaps when the component is loaded. When bitmaps are created the `Width` and `Height` properties are both `0` so there is nothing to draw on. I set the bitmap size to that of the component in the `Loaded` method.

➤ *Listing 2*

```
{ Paint the component as a dashed clear box the size of any loaded image,
  with the image rendered, at design time. Do nothing at run time. }
procedure TSprite.Paint;
begin
  if csDesigning in ComponentState then
    with Canvas do begin
      Pen.Style := psDash;
      Brush.Style := bsClear;
      Rectangle(0, 0, Width, Height);
      CopyMode := cmSrcAnd;
      CopyRect(ClientRect,FANDImage.Canvas,ClientRect);
      CopyMode := cmSrcPaint;
      CopyRect(ClientRect,FORImage.Canvas,ClientRect);
    end;
end;
```

After compiling TSpriteBox and TSprite into my component palette I tried them out. I placed a TSpriteBox on a new form and added a TSprite and a TTimer. Then I set the ANDImage and ORImage properties of TSprite. The sprite image displayed properly at design time so I gave myself a pat on the back.

In the TTimer's OnTimer event I made a call to SpriteBox1.DrawSprite then compiled and ran the project. Unfortunately the sprite started to move but quickly disappeared. I puzzled over this for a while until I realized the drawing bitmaps were the size of the component when it was loaded and not the size after I had aligned my component to the form. The solution was to descend from TCustomControl.

Descending From TCustomControl

Delphi provides a number of Custom... base classes in the VCL. One of them is called TCustomControl, which descends from TWinControl and as such has the methods and properties to process messages and to own other components.

By descending TSpriteBox from TCustomControl I could make my drawing surface respond to the Windows WM_SIZE message and re-size the drawing bitmaps any time the component changed size. At the same time I added a method called DrawBMP to isolate the sizing from the drawing of the background. After these changes my TSprite worked properly.

Extending TSpriteBox Capabilities

Once I had TSpriteBox working my imagination started to run riot. There seemed to be any number of enhancements that would meet all my requirements for a truly universal screen saver, not to mention other uses. I immediately thought of different colors, a background gradient and background images.

Adding color is quite simple. A property Color of type TColor serves to hold the selected color and the SetColor method writes the chosen color to the FColor field as well as calling DrawBMP, which sets the current brush color and then calls FillRect, which fills the BackGnd1 bitmap with the current brush. A call to Invalidate forces the BackGnd1 bitmap to be painted to the Canvas.

A gradient is not much harder. The method I called GradientFill. I chose FillRect again but this time I repeatedly used a Rect parameter which is a fraction of the screen Height and a progressively darker color. The Gradient property is type boolean and the SetGradient method sets the property and again calls DrawBMP.

Adding an image proved a little harder. I didn't want to just stretch the image to the background because of aspect ratio distortions, although I did want this as an option. If an image is not stretched I wanted the choice of centering it or leaving it at the top left corner.

Apart from having a colored background and possibly a gradient the image behavior would have to be the same as TImage. Here, the VCL source was indispensible. The paint method of TImage calculates the TRect based on the Stretch and Centered properties. This was exactly what I needed. In the DrawBMP method I set the color first, then add the gradient if requested and finally calculate the TRect for an image if required. As with the Color and Gradient properties I used an Image property of type TBitMap and in the SetImage method I call DrawBMP. The Stretch and Center properties hold the user's selections and SetStretch and SetCenter methods also call DrawBMP. See Listing 4.

About this time I realized TSpriteBox was a useful component for displaying images even without animating sprites. Animation was the reason for creating the component though and the final enhancement would be the ability to animate any number of sprites.

Iterating The Control List

All TControl descendants can own other components. They have a property called Controls of type

➤ *Listing 3*

```
procedure TSpriteBox.Paint;
begin
  Canvas.StretchDraw(ClientRect, FBackGnd1);
end;
```

➤ *Listing 4*

```
procedure TSpriteBox.DrawBMP;
var Dest: TRect;
begin
  { set size of BackGnd1 }
  FBackGnd1.Width := Width;
  FBackGnd1.Height := Height;
  { set brush color }
  FBackGnd1.Canvas.Brush.Color := FColor;
  { fill BackGnd1.Canvas }
  if FGradient then GradientFill(FColor,clBlack)
    else FBackGnd1.Canvas.FillRect(ClientRect);
  { if Image set then... }
  if (FImage.Width <> 0) and (FImage.Height <> 0) then begin
    { ...set Dest values... }
    if Stretch then
      Dest := ClientRect
    else if Center then
      Dest := Bounds((Width - FImage.Width) div 2,
                (Height - FImage.Height) div 2,
                FImage.Width, FImage.Height)
    else
      Dest := Rect(0, 0, FImage.Width, FImage.Height);
  end;
  { ...StretchDraw to BackGnd1.Canvas }
  FBackGnd1.Canvas.StretchDraw(Dest, FImage);
  { copy backgnd1 to backgnd2 }
  FBackGnd2.Assign(FBackGnd1);
  Invalidate;
end;
```

TList and a property called ControlCount which is the number of controls owned by the component. Using these properties and RTTI on the owned components it is possible to iterate through the controls list and, if they are of type TSprite, call TSprite.MoveSprite.

To implement this system I removed the property Sprite from TSpriteBox since it was no longer needed. Then I changed DrawSprite to the code in Listing 5.

Events Versus Methods

The only thing I didn't like about the sprite system so far was the hardcoded TSprite.MoveSprite method. Sure I could create a descendant and override the MoveSprite engine to achieve different sprite movement, but this would be messy. There would be TXSprites cluttering up my palette and every time I wanted a new animation I would probably end up creating another TXSprite. There had to be a better solution.

An event handler solved the problem. I created a type TSpriteMoveEvent with a variable Bounds to pass the sprite's present location and to return the desired new location (Listing 6). Within the old MoveSprite method I make a call to any assigned OnMove event and change the SLeft and STop properties accordingly (see Listing 7). Now TSprite has a plug-in engine. Figure 1 shows a demo application in simulated action. Don't worry if this image doesn't give you the true sense of what can be done, the demo is included on the disk along with all the code.

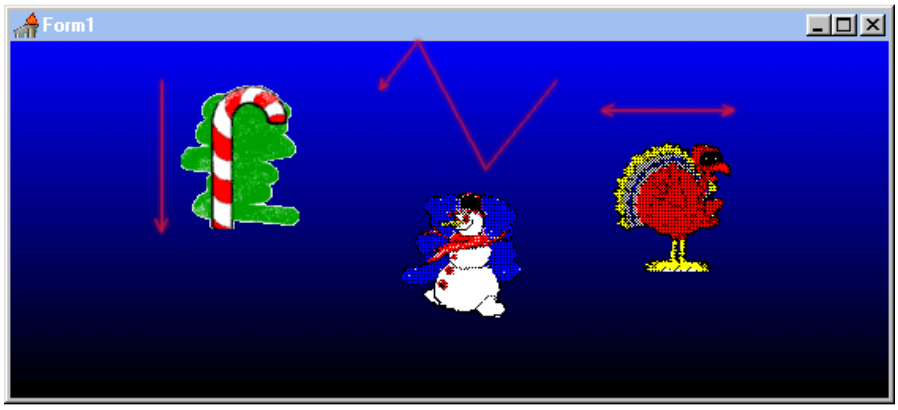
Back To The Screen Saver

Now we can finally finish the screen saver. Using the screen saver expert create an application framework. Name it ScreenSaver.Sprite, choose an About Box and click OK. Add a TSpriteBox to the form and set TSpriteBox.Align to alClient. Drop a TSprite on the form and insert an image and mask of your choice (there are some examples on the disk). Your form should look like Figure 2.

Finally, insert the code in Listing 8 into the Sprite1.OnMove event and put SpriteBox1.DrawSprite in the Application.OnIdle event defined by the expert.

I should point out here that you could just as easily use a TTimer to animate the sprite but somehow the Application.OnIdle event seems neater. I find you must call the API routine

```
PostMessage(
  Handle, WM_USER + 1,0,0)
```



➤ *Figure 1*

```
procedure TSpriteBox.DrawSprite;
var i, OldLeft, OldTop: integer;
begin
  for i := 0 to ControlCount-1 do begin
    if (Controls[i] is TSprite) then begin
      with (Controls[i] as TSprite) do begin
        OldLeft := SLeft;
        OldTop := STop;
        MoveSprite;
        { Erase the old sprite in BackGnd2 }
        BitBlt(BackGnd2.Canvas.Handle, OldLeft, OldTop, Width, Height,
          BackGnd1.Canvas.Handle, OldLeft, OldTop, SrcCopy);
        { Draw the sprite at the new location in BackGnd2 }
        BitBlt(BackGnd2.Canvas.Handle, SLeft, STop, Width, Height,
           ANDImage.Canvas.Handle, 0, 0, SRCAND);
        BitBlt(BackGnd2.Canvas.Handle, SLeft, STop, Width, Height,
           ORImage.Canvas.Handle, 0, 0, SRCPAINT);
        { Copy a rectangle from BackGnd2 to erase and reposition
          the sprite to the form's canvas }
        BitBlt(Canvas.Handle, OldLeft - 2, OldTop - 2, Width + 2,
               Height + 2, BackGnd2.Canvas.Handle, OldLeft - 2,
               OldTop - 2, SrcCopy);
      end;
    end;
  end;
end;
```

➤ *Listing 5*

```
type
  TSpriteMoveEvent =
    procedure(Sender: TObject; var Bounds: TRect) of object;
```

➤ *Listing 6*

```
{ MoveSprite: set bounds to size and LOCATION of image. Trigger an OnMove
  event. Set the location of sprite to any changed value of bounds. }
procedure TSprite.MoveSprite;
var Bounds: TRect;
begin
  Bounds := Rect(SLeft,STop,SLeft+Width,STop+Height);
  if Assigned(FOnMove) then OnMove(Self, Bounds);
  SLeft := Bounds.Left;
  STop := Bounds.Top;
end;
```

➤ *Listing 7*

➤ *Figure 2*

```
procedure TSaverForm.Sprite1Move(Sender: TObject; var Bounds: TRect);
var i: integer;
begin
  if ((Bounds.Left <= 0) or (Bounds.Right >= SpriteBox1.Width)) then
    (Sender as TSprite).VX := -(Sender as TSprite).VX;
  if ((Bounds.Top <= 0) or (Bounds.Bottom >= SpriteBox1.Height)) then
    (Sender as TSprite).VY := -(Sender as TSprite).VY;
  Bounds.Left := Bounds.Left+(Sender as TSprite).VX;
  Bounds.Top := Bounds.Top+(Sender as TSprite).VY;
end;
```

➤ *Listing 8*

or the `DrawSprite` method only executes once, though I'm not really sure why.

When you run the application you should see a bouncing sprite. You can play with the color, image and gradient properties until you have the screen saver you want, then just change the executable's extension to .SCR and move it to your Windows directory. Hey presto! A nice new screen saver.

The Setup Form

In a more sophisticated screen saver you may want users to select from various options. To do this you must use a setup form which is created when your application receives a `C` command line parameter from Windows. You pass the user's choices back to the application through the WIN.INI file. If you want to see how a setup form works in more detail look at the demo project FESTIVE.DPR on the disk.

A Word About Passwords

Many screen savers have password support added so that you can leave your desk and have your work secured. I haven't implemented passwords here for two reasons. First, passwords are implemented differently in Windows 3.1 and Win95 and to be frank I don't know how they work in Win95. Second, I don't know anyone who uses passwords for their screen savers. I guess if your work is that important you wouldn't rely on a screen saver for security. For anyone who wants to add password support, though, I don't think it would be too hard.

Conclusion

With the coding basically automated we can now concentrate on the fun part of creating screen savers. Since `TSpriteBox` handles any number of sprites and they can be driven with one or more algorithms there is little to limit your creativity. Try it, create some screen savers and have fun.

This being the holiday season we couldn't resist creating a special festive screen saver (see the screen shot on the cover), which you'll find on this month's disk. If you have a sound card be sure to set the `Sound` option to `True`. A ready-compiled 16-bit version, plus the background bitmap (which was too big to fit on the disk) and some extra example sprites can be downloaded from our Web site at

http://members.aol.com/delphimag

Paul Warren runs HomeGrown Software Development in Langley, British Columbia, Canada and can be reached by email at hg_soft@uniserve.com or visit http://haven.uniserve.com/~hg_soft