Streaming Delphi Components

Using Delphi's component streaming to create screensavers

by Jason Southwell

As you all know, Delphi is a rich and powerful development system. However, sometimes its RAD features can allow us to overlook some of its most powerful fundamental classes and functions. Classes like `TStreams` have always been around, but enhancements throughout each Delphi version have made these old dependable objects quite easy to use. The methods I will investigate in this article allow us to take advantage of Delphi component streaming.

Ever since I started to understand the benefits of component streaming, I find `TStreams` popping up more and more in my applications. In this article, we will be discussing a few methods, used internally by Delphi, that are also extremely useful for a variety of applications you may be developing.

It is true that streams are useful for data manipulation at the tiniest detail, but they also provide some higher-level functions to make data access and manipulation simple. Some of these higher-level functions allow us to do component streaming. For a basic introduction, component streaming is the process of copying a `TComponent` or descendant into a stream. The data in the stream is in a format very similar to a Delphi .DFM file. In fact, if you wrote a component stream to a file and assigned it a .DFM extension, you could open that file in Delphi and view it as though it were a form.

Once a component has been copied to a stream, you have many options for what to do with it. As we've already mentioned you can write the stream to a file; however, that is only the beginning. You could also send the component to another application via a memory-mapped file or TCP/IP socket. You could even embed the component into an EXE file for distribution. For the purposes of this article, we will embed our component stream into another EXE file in order to create distributable and customizable screensavers.

TStream: The Basics

While `TStreams` can be very useful tools, implementing them can be a daunting task for the beginning developer. I will assume that you know some stream basics, such as creation, destruction and simple navigation. If you need a good tutorial on the basics of `TStreams`, check out this article at the Project Jedi site: ftp://delphi-jedi.org/voyager/Strmhlp.zip.

We will make use of two methods introduced by `TStream` for the purposes of component streaming:

```
function ReadComponent(
  Instance: TComponent):
  TComponent
procedure WriteComponent(
  Instance: TComponent)
```

There are also two sister methods for writing in a format compatible with Windows resource files:

```
function ReadComponentRes(
  Instance: TComponent):
  TComponent
procedure WriteComponentRes(
  const ResName: string;
  Instance: TComponent)
```

Either will work for component streaming in general, but using the *write* method from one set means that you will have to use the *read* method from the same set. It's usually a good idea to use `ReadComponent` and `WriteComponent` instead of the resource versions, unless you need to store the streams for resource file compatibility.

Regardless, when well thought out and with an object oriented design, these streaming methods can be a 'silver bullet' for many programming challenges. Let's begin using these functions to create an application that generates windows screensavers.

The Task At Hand

To begin, I'll explain the basic premise of the code to follow. First, we will create a screensaver application that, when launched, will simply display a slideshow of pictures.

This seems simple enough, especially if you know the basics of how to create a windows screensaver. However, we want to do a little more. What if you wanted a way to create multiple screensavers, each with a different set of images in the slideshow. These screensavers could be sold, or given away as software packages. It would make sense, then, to create a screensaver shell on which you can superimpose a list of images every time you want to create a screensaver. Ideally, you would like to distribute these screensavers without having to recompile any code or manually rewrite the screensaver for the new images.

To accomplish this we need to come up with a clever way to retrieve the graphics to display. There are actually many ways you could do this, but only a few that would allow you to generate a new screensaver without a recompile. Well, since the topic of this article is practical uses for component streaming, let's accomplish our task via component streams.

The solution to our situation involves creating two projects. The first will be the actual screensaver application. This is the program that will eventually be placed in the Windows directory and selected via the Control Panel Display options. The second application will be a program that embeds the images into that screensaver. The idea is that when Windows launches the screensaver, it will load the images from within itself and display them in a slideshow.

So Let's Do It

In writing this article I am faced with a chicken and egg dilemma. You see, I can't really show you the screensaver application without showing you how those images are embedded into the screensaver, because the screensaver *is* the images embedded into the screensaver. I also can't show you the embedding application without having the compiled screensaver in which to embed them. Let's just pick a spot to start and dig right into it.

To create the basic screensaver, I borrowed code from the document TI4534D at community.borland.com. This describes how to write a basic 32-bit screensaver. Although the screensaver it walks you though is a bit more complex than what we are doing, it provides a fine baseline for our project. Since I started with the TI screensaver, the code you see in my screensaver will look very similar to it, however there are major differences in the actual display of the screensaver and preview windows.

Compiling this project will create a screensaver.dat file. We compile to a .dat file because this program should never be directly executed. Normally a screensaver is simply an EXE file renamed with the .scr extension. In our case, we will rename the .dat extension to .scr only after it has been embedded with graphics to display.

Since we are using component streaming to embed our images we are afforded the luxury of bowing to our 'object prowess' and can encapsulate most of our screensaver functionality into the components themselves. If you look at the code in Listings 1 and 2 you will see the basic components we will use to perform that encapsulation. `TSSImage` is the base class for our components. The intention is to have a base container class for the image that provides basic functionality for copying the image to the screen. `TSSTextImage` is a class inherited from `TSSImage` which builds upon the functionally of `TSSImage` to display the given text in the center of the screen on top of the image. Additional components could be created to do more advanced operations such as transitions, 3D effects, or anything else you could think of. For the purposes of this example however, we will only implement these two classes.

In addition to these two components, we will also create a helper component called `TSSFileImageLocations`. When we get to the point of embedding our images into our screensaver, we will need to know where each of these image components begin. This `TSSFileImageLocations` component will store a list of file locations pointing to each of our images. In Listing 3 you will see the code for this component. If we decided not to implement this component, we could still get the images by knowing the starting location and cycling through until the end of the file, but I decided not to do this as it can load down the memory with unused data and cause a speed problem when progressing from slide to slide. In this case, it is simply better to keep the image location list at hand.

We had to implement the `TSSFileImageLocations` component rather than using a `TList` or `TStringList` as those classes are not inherited from `TComponent` and therefore would not stream.

Embedding The Data

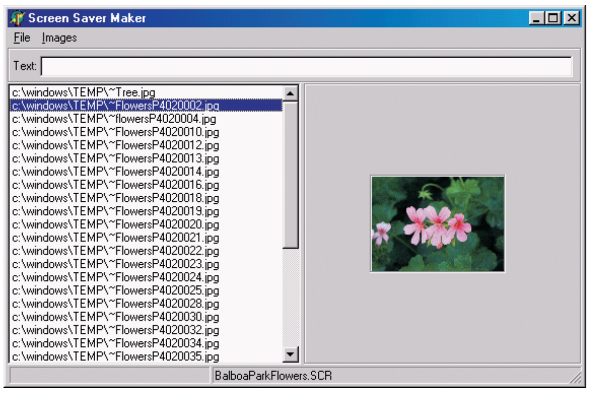
Now that we know the basic structure of how the screensaver will function, let's get into the embedding process. The second application, Maker, will create a .scr file by combining our screensaver.dat file with a series of embedded `TSSImage` components.

➤ *Listing 1*

```
TSSImage = class(TComponent)
private
  FFilename: string;
  FPicture: TPicture;
  procedure SetPicture(const Value: TPicture);
public
  constructor Create(AOwner : TComponent); override;
  destructor Destroy; override;
  procedure Execute(ToPicture : TPicture); virtual;
published
  property Picture : TPicture read FPicture write SetPicture;
  property Filename : string read FFilename write FFilename;
end;
TSSTextImage = class(TSSImage)
private
  FText: String;
public
  procedure Execute(ToPicture : TPicture); override;
published
  property Text : String read FText write FText;
end;
```

➤ *Listing 2*

```
procedure TSSImage.Execute(ToPicture : TPicture);
begin
  ToPicture.Assign(FPicture);
end;
procedure TSSTextImage.Execute(ToPicture: TPicture);
var
  tmpBitmap : TBitmap;
  x, y : integer;
  s : TSize;
begin
  inherited Execute(ToPicture);
  tmpBitmap := TBitmap.Create;
  try
    tmpBitmap.Width := ToPicture.Width;
    tmpBitmap.Height := ToPicture.Height;
    tmpBitmap.Canvas.Draw(0,0,ToPicture.Graphic);
    tmpBitmap.Canvas.Brush.Style := bsClear;
    tmpBitmap.Canvas.Font.Size := 32;
    tmpBitmap.Canvas.Font.Color := clWHite;
    tmpBitmap.Canvas.Font.Name := 'Arial';
    s := tmpBitmap.Canvas.TextExtent(FText);
    x := ( ToPicture.Width div 2) - (s.cx div 2);
    y := ( ToPicture.Height div 2) - (s.cy div 2);
    tmpBitmap.Canvas.TextOut(x,y,FText);
    ToPicture.Bitmap.Assign(tmpBitmap);
  finally
     tmpBitmap.Free;
  end;
end;
```



```
c:\windows\TEMP\~Tree.jpg
c:\windows\TEMP\~FlowersP4020002.jpg
c:\windows\TEMP\~flowersP4020004.jpg
c:\windows\TEMP\~FlowersP4020010.jpg
c:\windows\TEMP\~FlowersP4020012.jpg
c:\windows\TEMP\~FlowersP4020013.jpg
c:\windows\TEMP\~FlowersP4020014.jpg
c:\windows\TEMP\~FlowersP4020016.jpg
c:\windows\TEMP\~FlowersP4020018.jpg
c:\windows\TEMP\~FlowersP4020019.jpg
c:\windows\TEMP\~FlowersP4020020.jpg
c:\windows\TEMP\~FlowersP4020021.jpg
c:\windows\TEMP\~FlowersP4020022.jpg
c:\windows\TEMP\~FlowersP4020023.jpg
c:\windows\TEMP\~FlowersP4020024.jpg
c:\windows\TEMP\~FlowersP4020025.jpg
c:\windows\TEMP\~FlowersP4020028.jpg
c:\windows\TEMP\~FlowersP4020030.jpg
c:\windows\TEMP\~FlowersP4020032.jpg
c:\windows\TEMP\~FlowersP4020034.jpg
c:\windows\TEMP\~FlowersP4020035.jpg
```

BalboaParkFlowers.SCR

➤ *Figure 1*

The program simply allows the user to build a list of images and move them into the appropriate order (see Figure 1). When the user clicks on `Build Screensaver`, it opens up a file stream on a new screensaver file. First we copy the contents of the screensaver.dat file created by our other project. Executable files (or .scr files in this case) run from beginning to end. Isn't that logical? Also, they have no idea (nor do they care) how long they are. This makes it convenient for embedding data. All we have to do is ensure the executable code is placed first in our new file.

Next we cycle through the images that the user selected for the screensaver. For each image that we write, we add the given file position into our `TSSFileImageLocations` component. Also, notice that in this maker program every other image in the screensaver will display the text entered in the `Text` edit box. To accomplish this, we alter the creation code to create a `TSSImage` or `TSSTextImage` as appropriate for alternating images. This whole text process as implemented isn't very useful other than to show how each descendent component automatically uses its own `Execute` method when run. In a more advanced Maker program, you could store text values for each individual picture: feel free to make those changes yourself! To accent the write process, you can see the main create code pulled out into Listing 4.

After writing all of the images, we need to finish the file. First we write out the `TSSFileImageLocations` component. This of course is so that we can get the images when it comes time to read them. Then we write two integer values at a defined data length. In our case each integer can take up to 20 characters. The first integer value will be the location in the file that the image components start. The second integer value is the location in the file that the `TSSFileImageLocations` component starts. This gives us all the information necessary to get at the data we just embedded.

Getting At The Data

The reading process is very similar. First you read the `TSSFileImageLocations` component to determine the count and location of the images. Then you start your timer. In each `OnTimer` call, simply find the next image location from the file and call `ReadComponent`. When the component is loaded, execute the component.

By comparing the calls to `ReadComponent` in our application, you can see that there are two different ways to read a component from a stream:

```
sil := TSSFileImageLocations(
  fs.ReadComponent(sil));
ssi := TSSImage(
  fs.ReadComponent(nil));
```

In one case, as we do when we load the `TSSFileImageLocations`, we pass a variable to the `ReadComponent` method. The other call, like we use when we load each `TSSImage`, we pass a nil. When you pass a variable to the method, `ReadComponent` will simply make that variable have the same properties as the one being read. No new instance of the component is created, but rather the instance of the component passed is altered. On the other hand, by passing nil, you are telling `ReadComponent` to create the component for you. This is especially useful in the case of reading the `TSSImage` because we do not know if the image will in fact be a `TSSImage` or rather one of it's descendents (such as `TSSTextImage`). By allowing Delphi to create the component as the correct class, we can call the `Execute` method and allow Delphi to execute the correct code for us, as shown in Listing 5. This greatly simplifies quite a bit by encompassing all code for executing specified image types in their own component.

```
TSSFileImageLocations = class(TComponent)
private
  ListItems : TStringList;
  function GetCommaList: string;
  function GetItems(index: integer): Integer;
  procedure SetCommaList(const Value: string);
  procedure SetItems(index: integer; const Value: Integer);
  function GetCount: integer;
public
  constructor Create(AOwner : TComponent); override;
  destructor Destroy; override;
  property Items[index : integer] : Integer read GetItems write SetItems;
  property Count : integer read GetCount;
  function Add(Loc : integer) : integer;
published
  property CommaList : string read GetCommaList write SetCommaList;
end;
```

➤ *Above: Listing 3*

➤ *Below: Listing 4*

```
for i := 0 to ListFiles.Items.Count -1 do begin
  bShowText := not bShowText;
  if bShowText and (eText.Text <> '') then begin
    ssi := TSSTextImage.Create(nil);
    TSSTextImage(ssi).Text := eText.Text;
  end else begin
    ssi := TSSImage.Create(nil);
  end;
  try
    ssi.Picture.LoadFromFile(ListFiles.Items[i]);
    ssi.Filename := ExtractFileName(ListFiles.Items[i]);
    sil.Add(fsOut.Position);
    fsOut.WriteComponent(ssi);
  finally
    ssi.Free;
  end;
end;
```

```
fs := TFileStream.Create( Application.ExeName, fmOpenRead or fmShareDenyWrite );
try
  fs.Position := sil.Items[ImageIndex];
  ssi := TSSImage(fs.ReadComponent(nil));
  try
    TSSImage(ssi).Execute(Image1.Picture);
  finally
    ssi.Free;
  end;
finally
  fs.free;
end;
```

➤ *Above: Listing 5*

➤ *Below: Listing 6*

```
initialization
  classes.RegisterClass(TSSFileImageLocations);
  classes.RegisterClass(TSSImage);
  classes.RegisterClass(TSSTextImage);
```

Futureimage types can be added or current ones can be altered very easily. You must remember that when allowing Delphi to create the instance of the component, you must remember to free it when you are finished. This is an easy bug to miss and if you are not careful could cause major problems. It is not a good idea to have your system crash due to lack of available memory every time your screensaver starts.

When you pass a `nil` to `ReadComponent`, Delphi must know how to create the class for you. To inform Delphi of the existence of the class, you must call the `RegisterClass` procedure in the implementation section of your unit. This procedure registers a class of a persistent object so that its class type can be retrieved from streaming utilities such as `ReadComponent`. Unfortunately, the `RegisterClass` procedure is declared in two places. It is both a WinAPI call declared in windows.pas and a streaming utility declared in classes.pas. This can sometimes cause conflict and confusion, both for you and the compiler. Usually this is caused when your windows.pas file is placed after your classes.pas file in your `uses` clause. To avoid this potential error, it's a good idea to preface `RegisterClass` with its unit as shown in Listing 6.

Final Thoughts

You could use these techniques anywhere a `TStream` object or descendent is used. This provides you an object oriented way to read from files and execute functions via streams. Once you get familiar with these techniques, you will inevitably see other areas in which to use the benefits of component streaming.

Jason Southwell is Director of Internet Technologies for ComponentControl.com and has been developing in Delphi for 5 years, on both internet and databse projects. Contact him at jason@southwell.net