# ImageList In Depth

*by David Collie*

If you've used Delphi 3 then you're probably familiar with the `TImageList` component that ships with the VCL. This component is a wrapper for the image list common control provided with 32-bit versions of Windows. The image list is used to manage sets of images, either icons or bitmaps, in an efficient manner. The `TImageList` component is normally used in conjunction with other components, such as the listview or the treeview, to provide images that the other components can display. At its simplest level an image list just acts as a container for images, but it is much more versatile than that and in this article I'll show you what else it can do.

## Oddities...

Before delving into anything more complicated, I first want to cover an unusual aspect of the image list. The `Width` and `Height` properties, unlike any other component, do not specify the size of the component since it is a non-visual component. Instead they specify the size of the images to be stored. For example, setting `Width` and `Height` to 16 specifies that all images are 16 by 16 pixels in size. The component has a special constructor designed to define this size, `CreateSize`, that takes the image width and height as parameters. This constructor will not be called normally since the virtual `Create` constructor is called when components are created on a form. To use `CreateSize` you have to create the component by hand (see Listing 1).

This code creates a new image list that can store images 16 by 16 pixels in size. If the component is created automatically (by being placed on a form in the form designer) then set the `Width` and `Height` properties in the property inspector to define the image size. This should be done before adding any images to the list as changing these properties results in the list being cleared of all images it contains.

```
Procedure CreateImageList;
Begin
  SmallImageList := TImageList.CreateSize (16, 16);
End;
```

➤ *Listing 1*

```
Procedure AddIcon (AnIcon: TIcon);
Begin
  ImageList.AddIcon (AnIcon);
End;
Procedure AddBitmap (ABitmap: TBitmap);
Begin
  ImageList.Add (ABitmap, nil);
End;
Procedure AddImages (AnImageList: TImageList);
Begin
  ImageList.AddImages (AnImageList);
End;
Procedure FileLoad (AFileName: String);
Begin
  // Requires the mask colour to be defined (red).
 ImageList.FileLoad (rtBitmap, AFileName, clRed);
End;
Procedure ResourceLoad (AName: String);
Begin
  ImageList.ResourceLoad (rtBitmap, AName, clRed);
End;
Procedure ResourceInstanceLoad (AName: String);
Begin
  // HInstance is the global instance handle for the app.
  ImageList.ResInstLoad (HInstance, rtBitmap, AName, clRed);
End;
Procedure GetResourceInstance (AName: String);
Begin
ImageList.GetInstRes (HInstance, rtBitmap, AName, 0, [lrDefaultSize],
  clRed);
End;
```

➤ *Listing 2*

## Image Masks

Masks are used when the image list contains bitmaps and are used to make the images transparent, ie the image's background colour is transparent when drawn. It is not necessary to worry about masking if the image list contains icons, since an icon already contains information about its transparent colour. However, bitmaps do not contain this information and the image list needs to know how to mask them. There are two options: use a mask image that defines the transparent portion of the bitmap or define the mask colour for the images. Using a mask image is more involved and normally unnecessary and won't be covered here. It is far simpler to use a mask colour. This is done by setting the `BkColor` property to the colour that will act as a mask. All images will have any region of this colour drawn transparently. For example, if the `BkColor` is `clRed` then any red pixels in the image will be drawn transparently.

## Adding Images

There are several different ways of adding images to an image list. The simplest way is to double click the image list component in the form designer and use its property editor to add the desired images from disk files. This is the way that most people probably use it. However, there are a few twists when adding images this way. If a bitmap is added that is the same height as the image list but its width is a multiple of the image list's, then the property editor will offer to split it into multiple images. For example, if the image list's width is 16 and the bitmap's width is 80 then the property editor will split the

```
unit DragDropForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ComCtrls, ExtCtrls;
type
  TMainForm = class(TForm)
    DragImages: TImageList;
    DragSource: TStaticText;
    Target: TImage;
    procedure DragSourceMouseDown(Sender: TObject; Button:
      TMouseButton; Shift: TShiftState; X, Y: Integer);
    procedure DragSourceMouseUp(Sender: TObject; Button:
      TMouseButton; Shift: TShiftState; X, Y: Integer);
    procedure DragSourceMouseMove(Sender: TObject;
      Shift: TShiftState; X, Y: Integer);
  private
    ImageIndex: Integer;
  end;
var
  MainForm: TMainForm;
implementation
{$R *.DFM}
procedure TMainForm.DragSourceMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  Randomize;
  ImageIndex := Random (DragImages.Count);
  DragImages.SetDragImage (ImageIndex, 0, 0);
  DragImages.DragCursor := crDefault;
  DragImages.BeginDrag (Self.Handle, DragSource.Left + X,
    DragSource.Top + Y);
end;
procedure TMainForm.DragSourceMouseUp(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  MousePos: TPoint;
  TargetControl: TControl;
begin
  DragImages.EndDrag;
  // Find out what's under the mouse.
  MousePos.X := DragSource.Left + X;
  MousePos.Y := DragSource.Top + Y;
  TargetControl := Self.ControlAtPos (MousePos, True);
  if TargetControl = Target then begin
    DragImages.GetIcon (ImageIndex, Target.Picture.Icon);
  end;
end;
procedure TMainForm.DragSourceMouseMove(Sender:
  TObject; Shift: TShiftState; X, Y: Integer);
begin
  if DragImages.Dragging then begin
    DragImages.DragMove (DragSource.Left + X,
      DragSource.Top + Y);
  end;
end;
end.
```

➤ *Listing 3*

bitmap into 5 images each 16 pixels in height. This is useful when, for example, you've stored all of the images for a toolbar's buttons in one bitmap for convenience. Note that this will not work if the image list and bitmap share the same width but have a different height.

Images can be added at runtime as well as design-time. Here the options are to add an image from an existing `TImage` or `TBitmap` using the following methods.

`AddIcon` adds an icon to the image list. `Add` adds a bitmap and optionally a mask to the image list. As discussed previously, using an image mask is too involved so the `Mask` parameter can be set to `nil`.

`AddImages` takes all of the images defined in another image list and adds them to the image list.

The following only work when loading bitmaps. `FileLoad` loads an image from a file. This will only load bitmaps. `ResourceLoad`, `ResInstLoad` and `GetInstRes` all load an image from the resource section of the program or a DLL. Examples of all these methods are in Listing 2.

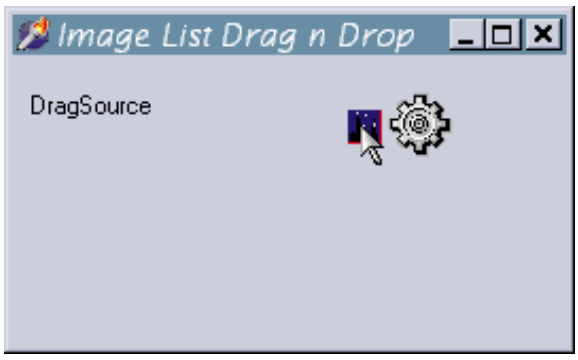
The method `GetInstRes` does the real work of loading an image from a resource. `FileLoad`, `ResourceLoad` and `ResInstLoad` are all 'helper' methods that pass the appropriate parameters to `GetInstRes`.

## Drag And Drop

The image list control also allows us to use an image it contains as a drag cursor. This can look much better than the standard drag cursor. The image can be of any type: it doesn't have to be a cursor, it can be either an icon or a bitmap. Unfortunately to do this the standard drag and drop mechanism cannot be used, ie you cannot set a control's `DragMode` property to `dmAutomatic` or call the `BeginDrag` method. If the standard drag and drop mechanism is invoked whilst using the image list's drag and drop methods, then the two will conflict and you'll get undesirable results. This means that there is more work involved using an image list for drag and drop than there is doing it the normal way.

To enable drag and drop the `SetDragImage` method must be called first. This defines the image index to display as the drag cursor and the image's hot spot. The image list's `DragCursor` can then be set which will result in the drag cursor being a combination of the defined cursor and the selected image. To display the new drag image the `BeginDrag` method is called. This takes as parameters the handle of the control which defines the boundaries of the drag operation, normally the parent form, and the initial position of the image. Because we're not using any of the standard drag and drop mechanisms this code must be called in the `OnMouseDown` event of the control to be dragged (the source). To turn off the drag image the `EndDrag` method of the image list is called in the `OnMouseUp` event of the source control. When the source is dragged the drag cursor must be moved manually by calling the image list's `DragMove` method in the source control's `OnMouseMove` event, specifying the X and Y coordinates of the cursor's new position. Listing 3 shows an example of how to string this all together. As you can see, this is quite a lot of work but may be worth it if you need cursor handling above and beyond that provided by standard drag and drop.

➤ *Figure 1: Drag n drop in operation.*

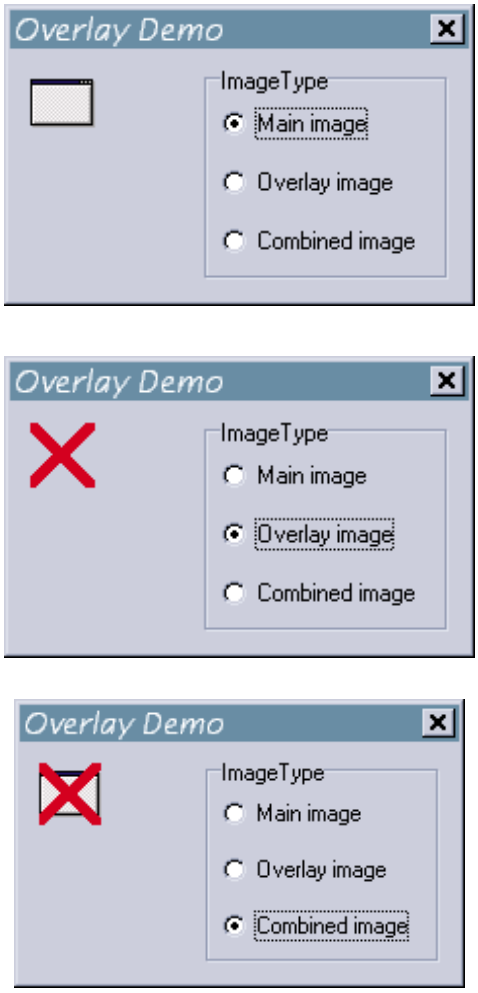



## Image Overlays

The last feature of the image list that I would like to cover is the ability to draw one image overlaid transparently onto another. Using

this feature you can, for example, overlay the 'shortcut' image onto another image as Windows does. There are two methods that are used when drawing overlaid images. The first, `Overlay`, registers an image in the list as an overlay image. Only images registered in this way can be drawn transparently over another. The method takes as parameters the index of the image to be used as an overlay and an overlay number between 0 and 3, which is used later when drawing the overlay to specify which overlay image is to be used. Once an image is registered in this way it is then possible to draw the overlaid image using the `DrawOverlay` method. This method draws onto a canvas the combination of an image in the list transparently overlaid with another image, previously registered using the `Overlay` method. Listing 4 has an example of this which allows you to display either the main image, the overlay image or the combination of the two. The code to draw the image is in the form's `OnPaint` event because any draw operation is not persistent. You can see the program running in Figures 2 to 4.

So what use can we put this overlay ability to? I used it to create a component, `TOverlaidImageList`, that holds a number of overlay images followed by a number of main images (to be overlaid). At runtime the image list creates an overlaid image for all the combinations of overlays and main images. The image list can then be used by a listview or treeview as normal but with the added capability of being able to display overlaid images for each list/treeview item. I've used this in my applications to convey the state of a list item by using a different overlay for each state. The component itself is quite simple. At design-time it behaves exactly the same as a normal image list. It does, however, have one extra property, `OverlayCount`, that specifies the number of images that are overlays. The first `0..OverlayCount -1` images in the list are then the overlay images, the rest being the main images that are going to be overlaid later. At runtime the component iterates through the combination of overlay and main images, adding a new image to the list for each combination. For example, if there are 2 overlay images and 5 main images an additional 10 images will be added to the list. These new images can then be referenced by other components such as the listview. The heart of this component is the `Loaded` method. This is called after a component has been created and its property values loaded from the form's resources. It is here that the additional images are created. The overlaid images are actually added to a temporary image list to make the routine easier to write. Once all of the overlaid images are created they are added from the temporary image list to the real image list and the temporary image list is destroyed. To create an overlaid image we use the above technique of calling `DrawOverlay`. However, we need a canvas to draw the overlaid image onto. The component creates a temporary bitmap for just this




➤ *From the top:*
*Figure 2: Main image displayed.*
*Figure 3: Overlay image displayed.*
*Figure 4: Combined image.*

➤ *Listing 4*

```
unit OverlayForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls;
type
  TMyForm = class(TForm)
    ImageList: TImageList;
    ImageType: TRadioGroup;
    procedure FormPaint(Sender: TObject);
    procedure ImageTypeClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
  public
  end;
var
  MyForm: TMyForm;
implementation
{$R *.DFM}
const
  OverlayNumber = 0;
  MainImage = 0;
  OverlayImage = 1;
procedure TMyForm.FormPaint(Sender: TObject);
const
  itMain = 0;
  itOverlay = 1;
  itCombined = 2;
begin
  case ImageType.ItemIndex of
    itMain    : ImageList.Draw (Self.Canvas, 10, 10,
                MainImage);
    itOverlay : ImageList.Draw (Self.Canvas, 10, 10,
                OverlayImage);
    itCombined: ImageList.DrawOverlay (Self.Canvas, 10, 10,
                MainImage, OverlayNumber);
  end;
end;
procedure TMyForm.ImageTypeClick(Sender: TObject);
begin
  // Draw the requested image (actually done in OnPaint).
  Invalidate;
end;
procedure TMyForm.FormCreate(Sender: TObject);
begin
  // Register the overlay.
  ImageList.Overlay (OverlayImage, OverlayNumber);
end;
end.
```

```
unit OverlaidImageList;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs;
type
  { This image list allows you to define overlays for the
    main images. The first OverlayCount images are overlaid
    onto the other images. At runtime each set of images is
    overlaid with the overlays. }
  TOverlaidImageList = class(TImageList)
  private
    FOverlayCount: TOverlay;
    NonOverlayCount: Integer;
  protected
    procedure Loaded; override;
    function GetOverlayedImageIndex(ThisOverlayIndex:
      Integer; ThisImageIndex: Integer): Integer;
  public
    property OverlayedImageIndex[OverlayIndex: Integer;
      ImageIndex: Integer]: Integer
      read GetOverlayedImageIndex;
  published
    property OverlayCount: TOverlay
      read FOverlayCount write FOverlayCount;
  end;
const
  NoOverlay = -1;
implementation
function TOverlaidImageList.GetOverlayedImageIndex(
  ThisOverlayIndex: Integer; ThisImageIndex: Integer):
  Integer;
// Returns index of the image overlaid with given overlay.
begin
  Assert((ThisOverlayIndex = NoOverlay) or
    (ThisOverlayIndex in [0..3]),
    'Overlay must be between 0 and 3.');
  { first set of images are the overlays, 2nd set not
    overlaid, 3rd set with overlay[0] and so on }
  Result := (NonOverlayCount * (ThisOverlayIndex + 1)) +
    ThisImageIndex;
end;
procedure TOverlaidImageList.Loaded;
var
  WorkList: TImageList;
  ThisImage: Integer;
  WorkBitmap: TBitmap;
  ThisOverlay: Integer;
begin
  inherited;
  NonOverlayCount := Count - OverlayCount;
  if not (csDesigning in ComponentState) and
    (OverlayCount > 0) then begin
    if OverlayCount > Count then begin
      raise EListError.Create(
        'Overlay count exceeds image count');
    end;
    WorkList :=
      TImageList.CreateSize (Self.Width, Self.Height);
    WorkBitmap := TBitmap.Create;
    try
      // Copy all non-overlaid images to a temp image list.
      WorkList.Assign (Self);
      // overlay each image in turn and add to working list.
      for ThisOverlay := 0 to OverlayCount - 1 do begin
        // Register the overlay.
        Overlay (ThisOverlay, ThisOverlay);
        for ThisImage := 0 to NonOverlayCount - 1 do begin
          // Clear out the bitmap.
          WorkBitmap.Create;
          WorkBitmap.Height := Self.Height;
          WorkBitmap.Width := Self.Width;
          // Overlay main image with its overlay onto bitmap
          DrawOverlay(WorkBitmap.Canvas, 0, 0,
            OverlayCount + ThisImage, ThisOverlay);
          // Add this to the working list.
          WorkList.Add (WorkBitmap, nil);
        end;
      end;
      // Now copy the overwrite me with the working images.
      Assign (WorkList);
    finally
      WorkList.Free;
      WorkBitmap.Free;
    end;
  end;
end;
end.
```

➤ *Listing 5*

purpose, ensuring that the bitmap is of the same size as the component. If you attempt to add an image to the image list that is not the same size as the image list then you'll get an exception (except in the circumstances I mentioned earlier).

The temporary bitmap is a `TBitmap` object that is easy to use and manipulate. A `TIcon` object cannot be used since it does not provide a `Canvas` property. Before each call to `DrawOverlay` the bitmap's constructor, `Create`, is called. This has the effect of reinitialising the bitmap. Calling `Create` on an already constructed object is different to calling the constructor as a class method, eg `TBitmap.Create`, in that it doesn't allocate memory for the object since this has already been done. To allow access to overlaid images at runtime one last property is added, `OverlaidImageIndex`. This is not published since it is only useful at runtime and not design-time, so runtime type information is not required for the IDE's object inspector. As it is an array property the RTTI for it is inaccessible anyway. This property takes two parameters, the index of an overlay image and the index of a main image. It then returns the index of the image that is a combination of the two. Passing the constant `NoOverlay` will return the main image by itself, not overlaid with any other image. Listing 5 shows the component's source code. To use the component at runtime, assign it to another component, for example as the `SmallImages` property of a listview. Then, in code, set the image index of the listviews using the `OverlaidImageIndex` property. An example of this is given in Listing 6.

➤ *Listing 6*

```
Procedure SetItemImage (Item: TListItem);
Const
  Shortcut = 0;
  WordDocument = 1;
Begin
  Item.ImageIndex := OverlaidImageIndex[Shortcut, WordDocument];
End;
```

## Conclusion

The image list can do far more than immediately meets the eye. However, most of its methods are not well documented and can be tricky to use. In this article I've shown you how to use some of its 'hidden' abilities.

Dave Collie is a senior Delphi and OO design consultant with Informatica Consultancy & Development, specialising in the design and implementation of large applications in an object oriented environment. He can be reached by email at dave@informatica.uk.com.