Under Construction: Delphi Goes Dynamic

by Bob Swart

In this month's column, we'll see how we can dynamically create components, both visual and non-visual, as well as event handlers for these components. Along the way, I'll show some practical (and sometimes fun) implementations. In short: Delphi goes dynamic.

Dynamic Components

There are a number of components that I create dynamically almost every day: database access components. To create a component dynamically all we need to do is call the `Create` constructor. For a simple `TObject`, this constructor doesn't even need a parameter, so the code is quite easy:

```
DynamicObject :=
  TObject.Create;
```

Unfortunately, for components (visual or not) we need to pass an argument to `Create`, to specify the owner of the new component. If you dynamically create components on a form, then the form will typically be the owner. If you dynamically create components otherwise (like in a console application), then you will typically not use an owner. For now, let's assume the component has no owner, so we can pass `nil` as the value for the argument to the constructor:

```
DynamicTable :=
  TTable.Create(nil);
```

When creating dynamic components, you must also be sure to delete them (in order to avoid memory leakage). In order to make sure you always delete a dynamically created component you should use a `try...finally` block, which can be seen in Listing 1.

```
var DynamicTable: TTable;
begin
  DynamicTable := TTable.Create(nil);
  try
    ... { using DynamicTable }
  finally
    DynamicTable.Free;
    DynamicTable := nil
  end
end;
```

➤ *Above: Listing 1*

Although it's usually the best option, note that we don't have to declare the dynamic component as type `TTable` right from the start. Suppose we need to create a `TTable` in one situation, but a `TQuery` in another. Then declaring the component of type `TDataSet` is enough, and the code to dynamically create a `TTable` or `TQuery` instance can be seen in Listing 2.

➤ *Below: Listing 2*

```
var
  DynamicDataSet: TDataSet;
begin
  DynamicDataSet := nil;
  if CreateTable then begin
    DynamicDataSet := TTable.Create(nil);
    try
      ... { DynamicDataSet AS TTable }
    finally
      DynamicDataSet.Free;
      DynamicDataSet := nil
    end
  end else begin
    { no Table - Query }
    DynamicDataSet := TQuery.Create(nil);
    try
      ... { DynamicDataSet AS TQuery }
    finally
      DynamicDataSet.Free;
      DynamicDataSet := nil
    end
  end
end;
```

We can optimise this code further by placing the `if` statement inside the `try...finally` block (whether we're creating a `TTable` or a `TQuery` we should always call `Free`, of course). And inside the loop we can depend on the RTTI information (using `IS` and `AS`) to work with the dynamic dataset component in those cases where a `TTable` and `TQuery` differ (see Listing 3).

```
var DynamicDataSet: TDataSet;
begin
  DynamicDataSet := nil;
  try
    if CreateTable then
      DynamicDataSet := TTable.Create(nil)
    else DynamicDataSet := TQuery.Create(nil)
     ... { DynamicDataSet AS TTable/TQuery }
  finally
    DynamicDataSet.Free;
    DynamicDataSet := nil
  end
end;
```

➤ *Listing 3*

Dynamic Visual Components

Creating non-visual components is easy. They often don't require an owner, and hence the `Create` constructor has no argument. Sometimes, however, a component can be 'owned' by another component, in which case the `Create` constructor takes an argument: the owner. A benefit of having an owner is that the owner is responsible for cleaning up the component itself (ie the owner will delete all the components it owns).

In order to create a new button (in response to a button click event, for example) we need to write the code from Listing 4.

This code indeed creates a new button, but we won't be able to see it on the form. That's because any visual component that gets created must have a parent: a window to position itself upon. And while the constructor gets an argument to use as the owner, there's no argument that you can pass for the parent. So, when dynamically creating a visual component, the first thing we need to do is assign the parent, for example to the form or a panel on which the component should show itself. In our example (see Listing 4), we should perform the following action right after the button is created:

```
Parent := Self;
```

This will make sure that the button's parent is the form itself (the `Self` pointer inside the method), and hence the button will be shown as a child control of the form (and positioned on the form according to the `Top` and `Left` properties of the new button itself).

Usually, the form is the *owner* of every control placed on it, but not the *parent* of every control placed on it (you can put a panel on the form, which has the form as its parent, and then a number of controls on top of the panel, meaning their parent is now the panel, and not the form). The form as owner means that every visual control will be deleted automatically when the form itself is deleted.

Dynamic Events

Once we have created dynamic components, it's time to go a step further and see if we can assign values to event handlers. So far, we've been able to dynamically create a component and set a few property values, but how do we actually assign an event handler? Well, it turns out to be not much more complex than assigning a regular property value (after all, an `OnClick` event handler is nothing more than a property `OnClick` of type `TNotifyEvent`, which is defined as follows:

```
type
  TNotifyEvent =
    procedure(Sender: TObject)
    of object;
```

The `of object` part here means that the method must be part of an object (ie a method that gets a `Self` pointer as an invisible first argument). The `TNotifyEvent` type is the type for events that only have the sender as parameter (ie the object that caused the event to fire).

In order to create a custom `TNotifyEvent` method, we can type the following declaration in the `private` section of the form (thus making it a method `of object`):

```
procedure OnClickEventHandler(
  Sender: TObject);
```

Go to this line and hit `Ctrl+Shift+C` to generate the source code for the empty event handler. Now, assign the dynamic button's `OnClick` event handler (placeholder) to the event handler (routine) we've just written, as follows:

```
OnClick := OnClickEventHandler;
```

If we run the combined application, we start with an empty form. If we click on the form, a new dynamic button is created and placed upon the form. If we then click on the button, the `OnClick` event handler (assigned to the method `OnClickEventHandler`) is fired, resulting in a messagebox showing `Hello, world!`. We have dynamic visual components with dynamic event methods.

```
procedure TForm1.FormClick(Sender: TObject);
begin
  with TButton.Create(Self) do begin
    Caption := 'Hello, world!';
    Top := 100;
    Left := 100;
  end
end;
```

➤ *Listing 4*

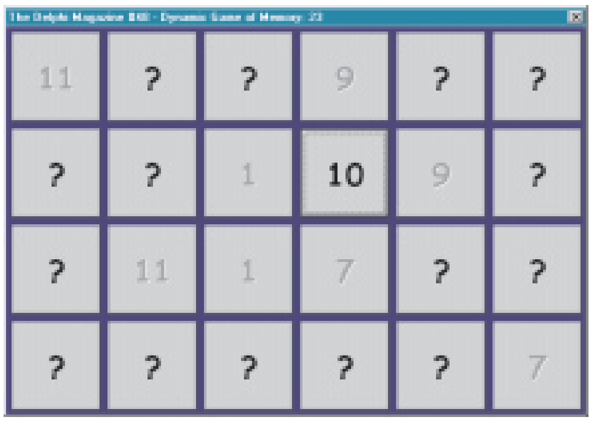
Final Example

As a final example, let's consider a game of memory. The game consists of a playing board with an even number of blank buttons. Each button is associated with a specific event (image, sound, movie, etc), and every button has a 'clone' button, that is, for every button with a specific image there's exactly one other button with the same image (the same holds for sound waves or video movies). The objective of the game is to find pairs of buttons, by clicking on two buttons, and if they are the same, then the buttons disappear and the player scores (and may continue), otherwise it's the other player's turn to try. You can even play this game by yourself, in which case the number of turns divided by the number of pairs is an indication of your skill level.

How would this game be an example of creating dynamic components and event handlers? Well, I'm glad you asked. The thing is: the layout of the board is not fixed. Erik, my 5-year-old son, usually starts with a 6 x 4 board (24 buttons), but Tasha, my little daughter, prefers the 4 x 3 board (only 12 buttons). This means that we don't know the number of buttons beforehand. In fact, we'd need a two-dimensional dynamic array of buttons to store them:

```
var
  Buttons:
    Array of Array of TButton;
```

Once the player has specified the dimensions, we can allocate memory for the button pointers (using the `SetLength` method), and then dynamically create each individual button. Each set of two buttons get the same `Tag` value (so these buttons 'belong to' each other), and a question mark as the caption. All this is implemented in method `CreateBoard` (see Listing 5), where a default `OnClick` event handler is also assigned. Once the



➤ *Figure 1*

two dimensional set of buttons is created, we need to shuffle them, of course (otherwise they're nicely paired next to each other, an easy game, even for Tasha). This is implemented in the `Shuffle` routine.

After all these events (if you will pardon the pun), the game is ready to start. Each button has a method called `FirstButtonClick` which is assigned to its `OnClick` event handler. This method makes sure the caption of the button is shown, and the game gets ready to respond to the second button click. In order to do so, the event handler of each button must be replaced by the method `SecondButtonClick`. We can do this by iterating through the `Components` array of the Form itself. If a component is of type `TButton` and still `Enabled`, then we should assign `SecondButtonClick` to the `OnClick` event handler property. All this is implemented in the method `FirstButtonClick`, of course.

Finally, once the `SecondButtonClick` method has been installed as a new dynamic event handler, the game just waits for the user to click on another button. If this button has the same `Tag` value (ie the two buttons belong together), then both are disabled (so you see their caption with a grey/disabled font). Otherwise, the game waits about a second using the `Sleep` API, before clearing the captions of the two buttons, replacing them with the original question marks again. Finally, the event handlers of the remaining enabled buttons must be set to the `FirstButtonClick` event handler, of course (as it's now the turn for the first button to be clicked again). All this is done in the `SecondButtonClick` method.

The complete source code of the memory game can be seen in Listing 5.

Note that, currently, the board cannot be resized. In order to implement this, you'd need to override the `SetBounds` method,

➤ *Listing 5*

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;
const
  _Caption =
   'The Delphi Magazine #48 - Dynamic Game of Memory: %d';
type
  TArrayArrayButton = Array of Array of TButton;
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  private
    Turns: Integer;
    procedure Shuffle(Button: TArrayArrayButton);
    procedure FirstButtonClick(Sender: TObject);
    procedure SecondButtonClick(Sender: TObject);
  public
    procedure CreateBoard(X,Y: Integer);
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.FormCreate(Sender: TObject);
begin
  Randomize;
  Turns := 0;
  CreateBoard(6,4)
end;
procedure TForm1.CreateBoard(X, Y: Integer);
var
  i,j: Integer;
  Button: TArrayArrayButton;
begin
  SetLength(Button, X);
  for i:=0 to Pred(X) do begin
    SetLength(Button[i], Y);
    for j:=0 to Pred(Y) do begin
      Button[i,j] := TButton.Create(Self);
      Button[i,j].Parent := Self;
      Button[i,j].Left := 6 + (600 div X) * i;
      Button[i,j].Width := (600 div X) - 8;
      Button[i,j].Top := 6 + (400 div Y) * j;
      Button[i,j].Height := (400 div Y) - 8;
      Button[i,j].Tag := 1 + (j * X + i) div 2;
      Button[i,j].Caption := '?';
      Button[i,j].OnClick := FirstButtonClick
    end
  end;
  Shuffle(Button)
end;
procedure TForm1.Shuffle(Button: TArrayArrayButton);
var
  i: Integer;
  X,Y: Integer;
  X1,X2,Y1,Y2: Integer;
begin
  X := Length(Button);
  Y := Length(Button[0]);
  for i:=1 to 1001 do begin
    X1 := Random(X);
    X2 := Random(X);
    Y1 := Random(Y);
    Y2 := Random(Y);
    Tag := Button[X1,Y1].Tag;
    Button[X1,Y1].Tag := Button[X2,Y2].Tag;
    Button[X2,Y2].Tag := Tag
  end
end;
procedure TForm1.FirstButtonClick(Sender: TObject);
var i: Integer;
begin
  Tag := (Sender AS TButton).Tag;
  (Sender AS TButton).Caption := IntToStr(Tag);
  for i:=0 to Pred(ComponentCount) do
    if Components[i] IS TButton then
      if (Components[i] AS TButton).Enabled then
        (Components[i] AS TButton).OnClick :=
          SecondButtonClick  { assign new event handler }
end;
procedure TForm1.SecondButtonClick(Sender: TObject);
var i: Integer;
begin
  Inc(Turns);
  Caption := Format(_Caption,[Turns]);
  (Sender AS TButton).Caption :=
    IntToStr((Sender AS TButton).Tag);
  if (Sender AS TButton).Tag = Tag then begin
    { the same }
    (Sender AS TButton).Enabled := False;
    for i:=0 to Pred(ComponentCount) do
      if Components[i] IS TButton then
        if (Components[i] AS TButton).Tag = Tag then
          (Components[i] AS TButton).Enabled := False
  end else begin
    { not the same; hide again }
    Sleep(1000);
    for i:=0 to Pred(ComponentCount) do
      if Components[i] IS TButton then
        if (Components[i] AS TButton).Enabled and
          ((Components[i] AS TButton).Caption <> '?') then
          (Components[i] AS TButton).Caption := '?'
  end;
  for i:=0 to Pred(ComponentCount) do
    if Components[i] IS TButton then
      if (Components[i] AS TButton).Enabled then
        (Components[i] AS TButton).OnClick :=
          FirstButtonClick  { assign new event handler }
end;
end.
```

and make sure the buttons resize according to the same logic that's used in the `CreateBoard` method. Also, once the game is over, there's no way to restart (not even with the same board layout). One way to implement this feature is to dynamically delete all the button components and then call `CreateBoard` again (maybe asking for the new dimensions first). I leave it as an exercise for the reader to implement these two additional features.

Finally, the example I've presented here uses a simple number, and not an image or sound waves (let alone a video movie). It's not hard to implement other behaviour inside the events, of course, but the images I downloaded from the Disney and Warner Brothers websites would probably violate their copyrights if I were to distribute them with the source code on this month's disk...).

Next Time

We've seen how to dynamically create and destroy components, and even how to assign dynamic event handlers to them (and how to change event handlers 'on the fly'). These techniques can prove quite handy at times, especially when certain boundaries are unknown beforehand. With the introduction of dynamic arrays in Delphi 4, we can now even have dynamic arrays of dynamic buttons: a big step forward!

By the way, there's still one way to cheat this game (as Tasha quickly demonstrated): just click on the same button twice, which satisfies the check that the `Tag` property of the second button is equal to the one of the first button, and subsequently disables not only this button, but the associated button as well. Only a minor glitch, but Tasha was able to find it in a few seconds (by double-clicking on the same button right from the start). Of course, the simple fix is to include a test in the `SecondButtonClick` event handler to see if the caption of the second button is still a question-mark, and not a number already...

Next time, I'll focus on Delphi efficiency again, specifically the network. After CPU, memory and graphic operation optimisations on standalone machines, we must face the truth: we're no longer alone. Distributed application efficiency depends on more than a standalone application. And that's where the network comes in. And bandwidth. And the amount of data being transported along the wire, of course. Until that time, *stay tuned...*

Bob Swart (aka Dr.Bob, visit www.drbob42.com, email him at drbob@chello.nl) is a technical consultant and webmaster for TAS Advanced Technologies (www.tas-at.com) using Delphi, JBuilder and C++Builder, and freelance technical author. In his spare time, Bob likes to watch video tapes of Star Trek Voyager and Deep Space Nine with his 5-year-old son Erik Mark Pascal and his 2.5-year-old daughter Natasha Louise Delphine.