Multiple Undo

by Warren Kovach

In his recent review of my book *Delphi 3 – User Interface Design* (in *Developers Review* Issue 2), Richard Stevens mentioned that one thing he would like to have seen is a section on how to implement a multiple undo system. Let it not be said that I don't give my readers what they want! Here, Richard, is how you do multiple undo...

Windows has a basic undo facility built into some controls, such as the edit control used in `TEdit` and `TMemo`. By sending an `EM_UNDO` message to an edit control you can have Windows restore the last text deleted or remove the last characters typed. However, this only works once. If you delete something then type some more text, Windows provides no way to get back the deleted text.

Most modern programs will have a multiple undo facility that lets you reverse any number of recent changes. This not only provides the security of knowing that you can reverse any accidental changes made, it also allows you to play 'what if' with a document. You can experiment with a series of changes safe in the knowledge that you can take it back to its original state if the experiment doesn't pan out.

A multiple undo facility is usually implemented as a sequential one. If you want to reverse a change you made a while ago, you must reverse all the intervening actions as well. It is done this way because the different actions may be interrelated.

Let's say you have entered some text, then later inserted some more text into the middle of the previously entered section. Now you want to perform an undo on just the first batch of text. What happens to the text you entered later? Should it be left there, perhaps out of context? Should it also be deleted? Rather than tackle these problems most programs simply undo all the intervening actions as well.

With this model the obvious method of implementing a multiple undo is as a stack. For those new to data structures, a stack is a last-in-first-out (LIFO) structure. It is similar to a stack of dinner plates. New plates are added to the top of the stack. If you then want to retrieve a plate you take one off the top, not from the bottom. If you want to retrieve the fifth plate down from the top you must also remove the four on top of it.

Command Objects

So what do we save on the stack? You could use the stack simply to store the data (perhaps wrapped up in an object). When your program draws a circle it also creates an object containing data about it (such as its position and radius) and places it on the stack. When an undo command is issued it retrieves the data and tries to remove that circle. This can get a bit messy, though, as your program code gets littered with extra lines concerned with maintaining the stack, which can reduce the readability. Also, if there are a large number of different types of actions that can be undone, the event handler for the `Edit | Undo` menu item can get convoluted.

It would make more sense (and fit in better with object oriented programming's encapsulation principle) to have the undo object take over all the work. Let's not just view the object on the stack as a repository of data. Let's look at the object as a command of some sort that knows how to reverse itself. Instead of your program drawing a circle and adding its undo data to a buffer, it should create a 'draw a circle' object. Your program simply executes the object to do the drawing, after which it is saved on the stack. If the user selects the `Undo` menu item then you just tell the circle object to remove itself from the canvas.

This approach is the classic 'command pattern' described in the book *Design Patterns – Elements of Reusable Object-Oriented Software* (by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, published by Addison-Wesley, 1995). This model, where each command handled by a program is wrapped up in an object, not only lets you undo actions, it also facilitates queuing and logging actions and directing them to appropriate subsystems or clients.

A minimum undoable command object needs two methods. One, perhaps called `DoCommand` or `Execute`, actually carries out the action, such as drawing a circle on a canvas. Another one, called `Undo`, reverses the effect of the command. If you wish to be able to redo undone commands you can also have a `Redo` method. This may simply call `DoCommand` again or it may have to take special steps to redo the action.

A nice addition, from a user interface point of view, is to have the object also return a description of the type of command it performs. The description can then be used for the text on a menu or displayed in the hint area of a status bar so that the user knows what will be undone. You can have different descriptions for different purposes, such as *Undo line drawing* or *Redo typing 'Hello world'*. Each descendant object defines its own descriptions.

Let's look at the abstract base class for our hierarchy of command objects (Listing 1). As promised, it has three public methods: `DoCommand`, `Undo` and `Redo`. These are abstract methods, as each type of command class will have to tailor these to its own use. The class also has a series of properties and associated read methods that return appropriate descriptions of the

```
type
  TUndoItem = class(TObject)
    protected
      function GetUndoDescription : string; virtual;
      function GetShortUndoDescription : string; virtual;
      function GetRedoDescription : string; virtual;
      function GetShortRedoDescription : string; virtual;
      function GetUndoMenuText : string; virtual;
      function GetRedoMenuText : string; virtual;
    public
      procedure DoCommand; virtual; abstract;
      procedure Undo; virtual; abstract;
      procedure Redo; virtual; abstract;
      property UndoDescription : string read GetUndoDescription;
      property ShortUndoDescription : string read GetShortUndoDescription;
      property RedoDescription : string read GetRedoDescription;
      property ShortRedoDescription : string read GetShortRedoDescription;
      property UndoMenuText : string read GetUndoMenuText;
      property RedoMenuText : string read GetRedoMenuText;
  end;
```

➤ *Listing 1*

command. The MenuText ones provide the entries for the Undo and Redo menu items on the Edit menu, while the others provide either long or short descriptions of the undo and redo actions.

Stacking 'Em Up

Now that we have our basic undoable command object design we need a data structure in which to keep instances of it. We could roll our own stack structure using pointers or arrays, but a simpler method is to customize Delphi's TList class. We can use the Add method to add objects to the stack and Delete to remove them. However, we must take care to enforce the LIFO nature of a stack, so we don't call Add and Delete directly. This is particularly important for Delete, since this method can delete items anywhere, not just at the top of the stack. Instead we define our own methods for manipulating the stack. These always use Add, which adds objects to the end of the list, and Delete(pred(Count)), which removes the last item in the list.

Our stack is going to provide a multiple redo facility as well as undo. We do this by providing a pointer to the top of the undo stack. When a command is undone the pointer is moved down, to point to the next item in the stack that can be undone. Anything above this pointer is a candidate for redo. If the user chooses to redo an action, the item just above the stack pointer is redone and the pointer is moved up. Figure 1 illustrates this.

We also must dispose of all redo items each time a new command is added. This is done for the same reason that the undo stack is sequential: if we try to redo a command after we have changed the underlying document, the results could be undefined. We cannot redo the formatting of a section of text if an intervening new action has deleted that text.

Listing 2 gives the important methods of our undo stack, TUndoStack. A key field in the object is CurrentUndo. This is the stack pointer that defines the boundary between undo and redo objects. If nothing has been undone (and therefore there are no redo items) then the pointer is equal to the highest list index (Count - 1). If CurrentUndo is ever less than this then there are some undone commands that can be redone. The CanRedo boolean method is used to check this status. The accompanying CanUndo method simply checks whether there are events that can be undone. These methods and fields are used to set the properties CurrentItem and CurrentRedoItem, which return pointers to the appropriate TUndoItems.

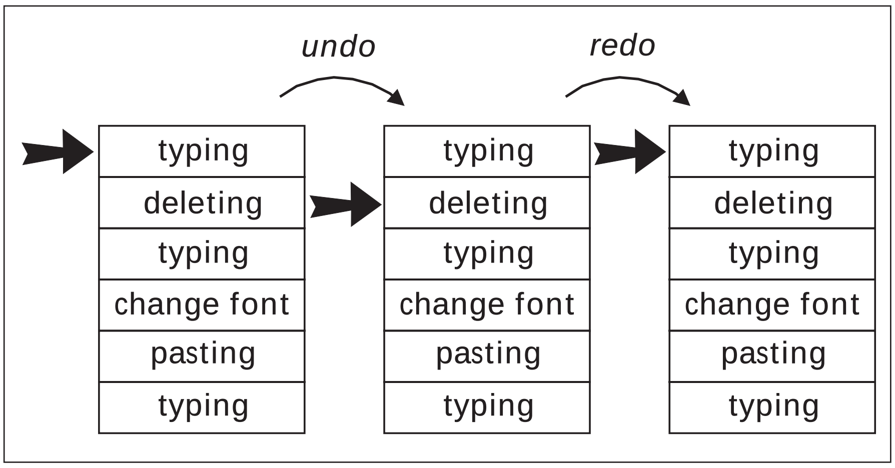
Now, on to the heart of the undo stack, the Submit method. It is through this that command objects are actually executed and added to the stack. The calling program will create an appropriate command object and pass it to the stack through this method. The first order of business is to call the command object's DoCommand method. This will perform whatever needs to be done, such as draw a circle or change the formatting of some text. Submit then sets about maintaining the stack. If any items are available for redoing these are disposed of. If the stack is full then an older undo item must be removed to make room for the new one. Finally, the inherited TList.Add is called to add the object to the stack and CurrentUndo is pointed at it. The method returns an indicator of whether the stack is full, in case the calling program needs to know if earlier undo items were removed.

Undoing and redoing commands is done through the Undo and Redo methods of TUndoStack. These call the equivalent methods of the current undo and redo items. The methods allow several items to be dealt with at once.

Using The Stack

Now, let's take a look at how to create and use an undo stack. We will create a graphics window that lets you place different shapes on the canvas by clicking with the

➤ *Figure 1*



```
constructor TUndoStack.Create(AMaxItems : integer);
begin
  inherited Create;
  FMaxItems := AMaxItems;
  if FMaxItems > MaxListSize then
    FMaxItems := MaxListSize;
  CurrentUndo := -1;
end;
destructor TUndoStack.Destroy;
begin
  Clear;
  inherited Destroy;
end;
procedure TUndoStack.Clear;
var i: Integer;
begin
  for i := pred(Count) downto 0 do
    DeleteAndFree(i);
  inherited Clear;
end;
procedure TUndoStack.DeleteAndFree(Item : integer);
begin
  TUndoItem(Items[Item]).Free;
  inherited Delete(Item);
end;
procedure TUndoStack.SetMaxItems(AMaxItems : integer);
var i : integer;
begin
  { delete oldest entries if list is shrinking }
  if AMaxItems < FMaxItems then
    for i := 0 to pred(FMaxItems - AMaxItems) do
      DeleteAndFree(0);
  FMaxItems := AMaxItems;
  CurrentUndo := pred(Count);
end;
function TUndoStack.GetCurrentItem : TUndoItem;
begin
  if CanUndo then
    Result := Items[CurrentUndo]
  else
    Result := nil;
end;
function TUndoStack.GetCurrentRedoItem : TUndoItem;
begin
  if CanRedo then
    Result := Items[succ(CurrentUndo)]
  else
    Result := nil;
end;
function TUndoStack.CanUndo : boolean;
begin
  Result := CurrentUndo >= 0;
end;
function TUndoStack.CanRedo : boolean;
begin
  Result := (Count > 0) and (CurrentUndo < pred(Count));
end;
function TUndoStack.Submit(Item:TUndoItem) : TStackStatus;
var i : integer;
begin
  Item.DoCommand;
  if CanRedo then
    for i := pred(Count) downto succ(CurrentUndo) do
      DeleteAndFree(i);
  if Count >= MaxItems then begin
    DeleteAndFree(0);
    Result := ssFull;
  end else
    Result := ssNotFull;
  inherited Add(Item);
  CurrentUndo := pred(Count);
end;
procedure TUndoStack.Undo(Num : integer);
var i : integer;
begin
  if CanUndo then
    for i := 1 to Num do begin
      CurrentItem.Undo;
      dec(CurrentUndo);
      if not CanUndo then exit;
    end;
end;
procedure TUndoStack.Redo(Num : integer);
var i : integer;
begin
  for i := 1 to Num do
    if CanRedo then begin
      CurrentRedoItem.Redo;
      inc(CurrentUndo);
    end;
end;
procedure TUndoStack.RemoveLastItem;
begin
  if Count > 0 then begin
    DeleteAndFree(pred(Count));
    dec(CurrentUndo);
  end;
end;
```

➤ *Listing 2*

mouse. The drawing (and undoing) of each shape will be handled by a command object called TShapeUndoItem, descended from TUndoItem.

TShapeUndoItem has three private fields, the settings of which are passed through the constructor Create. Location is a TPoint that gives the position at which a graphic object should be drawn, DrawingTool is an enumerated type indicating which shape should be drawn, and Canvas is the drawing surface that should be used.

The DoCommand method, which the TUndoStack calls when the command object is submitted, simply calls the DoDrawing method (Listing 3). The first order of business in that method is to set the Pen.Mode property of the Canvas to pmNotXOR. This will draw a shape, allowing us to erase it by simply calling DoDrawing again: this is all that the Undo and Redo methods do. After this is a case statement that draws the appropriate shape, depending on what type of drawing tool was passed to the command object on creation.

Submitting the command object to the undo stack is a simple procedure. We just create an OnMouseDown event handler and add the code in Listing 4 to it.

```
procedure TDrawShapeUndoItem.DoDrawing;
begin
  with Canvas,Location do begin
    Pen.Mode := pmNotXOR;
    case DrawingTool of
      dtLine :
        begin
          MoveTo(Right,Top);
          LineTo(Left,Bottom);
        end;
      dtRectangle : Rectangle(Left,Top,Right,Bottom);
      dtEllipse   : Ellipse(Left,Top,Right,Bottom);
      dtRoundRect : RoundRect(Left,Top,Right,Bottom,
                      (Left-Right) div 2,(Top-Bottom) div 2);
    end;
  end;
end;
```

➤ *Listing 3*

```
var Item : TUndoItem;
{ ... }
if Button = mbLeft then begin
  Item := TDrawShapeUndoItem.Create(Image1.Canvas,
    Rect(x,y,x+50,y+50),DrawingTool);
  if UndoStack.Submit(Item) = ssFull then
    ShowMessage(Format(sStackFull,[UndoStack.MaxItems]));
end;
```

➤ *Listing 4*

Extended Actions

The above simple example works fine when you are dealing with discrete actions and events. But what about commands that, to the user, seem like one action, but are programmatically a series of connected events? When a user of a graphics program presses and holds the mouse button, moves the mouse to draw a free-form line, and releases the button, he or she views that as one action. However, at the programmer's level this is a series of events: the mouse button down, a series of mouse move events, and the mouse button up. Each segment of the line is drawn with a separate call to the LineTo method each time a mouse move event is trapped. When the user selects Edit | Undo after drawing a free-form line the whole line should disappear. But, if we put each LineTo command onto the undo stack then the user may have to select Undo dozens of times to undo the single free-form line. To rectify this, we must trap the series of events and turn them into one single undo object.

Listing 5 shows the main methods of a TDrawLineUndoItem class, descended from TDrawShapeUndoItem. This extends the previous class by allowing the program to add new data to that already stored in the object. When the user starts to draw a line an instance of the object is created (in the OnMouseDown event handler), storing the initial mouse position. Then, each time the mouse is moved a method of the undo object called AddSegment is called. This receives the new position of the mouse, draws a line from the previous position to the new one, then adds the new position to a list of previous mouse coordinates. When Undo or Redo are called the object steps through the list, redrawing the lines to each position.

The simplest way to maintain a list is with a TList. Normally a TList is used to hold objects, or at least pointers to something. However, if you want to store some type of data that fits into the same space as a pointer (four bytes) you can store these directly in a TList. Then, by typecasting the value returned by TList.Items, we can retrieve the data. This is done in the AddPointToList and Undo methods in Listing 4, where TPoints are stored. Under Delphi 1 a TPoint is 4 bytes, so it can be stored as a single item in a TList. Under 32-bit Delphi it is 8 bytes, so the x and y coordinates must be stored separately.

AddPointToList is first called in the DoDrawing method, where the initial mouse coordinates are stored and a MoveTo is issued to set the starting point for drawing. Then, each time the mouse is moved the method AddSegment is called. This takes the new mouse coordinates, draws a line to the new position and stores the position in the list. AddSegment is called one final time when the mouse button is released. To undo (or redo) the line we simply walk through the list of positions, retrieving them by typecasting the value returned by the TList.Items array property, and draw the line segment.

We'll now create a graphics drawing window that uses both the shape and line drawing command objects (Listing 6). To keep track of line drawing actions two boolean fields must be used, LineDrawing and MouseMoved. LineDrawing is set to True when the mouse button is pressed and back to False when it is released. Then, when mouse movements are trapped in Image1MouseMove, AddSegment is only called if line drawing is occurring. MouseMoved is also set to True here. It is a flag that lets us know if the mouse was moved while the button was down. We don't want to store anything on the undo stack if a user simply clicks the mouse without moving it to draw a line, so we remove the command object with the RemoveLastItem method of TUndoStack.

Text Editing

As I mentioned at the start of this article, TMemo has its own single undo built in. We can extend this to multiple undo using another descendant of TUndoItem. However,

➤ *Listing 5*

```
procedure TDrawLineUndoItem.AddPointToList(P : TPoint);
begin
 {$IFDEF Win32}
  PointList.Add(pointer(p.x));
  PointList.Add(pointer(p.y));
 {$ELSE}
  PointList.Add(pointer(p));
 {$ENDIF}
end;
procedure TDrawLineUndoItem.DoDrawing;
begin
  with Canvas,Location do begin
    Pen.Mode := pmNotXOR;
    MoveTo(Left,Top);
    AddPointToList(TopLeft);
  end;
end;
procedure TDrawLineUndoItem.AddSegment(NextPoint : TPoint);
begin
  with Canvas do begin
    Pen.Mode := pmNotXOR;
    LineTo(NextPoint.x,NextPoint.y);
  end;
  AddPointToList(NextPoint);
end;
procedure TDrawLineUndoItem.Undo;
var
  i : integer;
  P : pointer;
 {$IFDEF Win32}
  Point : TPoint;
 {$ENDIF}
begin
  with Canvas do begin
    Pen.Mode := pmNotXOR;
   {$IFDEF Win32}
    P := PointList[0];
    Point.X := Longint(P);
    P := PointList[1];
    Point.y := Longint(P);
    with Point do Moveto(x,y);
    for i := 2 to pred(PointList.Count) do begin
      P := PointList[i];
      if Odd(i) then begin
        Point.y := Longint(P);
        with Point do LineTo(x,y);
      end else
        Point.X := Longint(P);
    end;
   {$ELSE}
    P := PointList[0];
    with TPoint(P) do Moveto(x,y);
    for i := 1 to pred(PointList.Count) do begin
      P := PointList[i];
      with TPoint(P) do LineTo(x,y);
    end;
   {$ENDIF}
  end;
end;
procedure TDrawLineUndoItem.Redo;
begin
  Undo;
end;
```

```
procedure TGraphicWindow.Image1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var Item : TUndoItem;
begin
  MouseMoved := false;
  Case Button of
    mbLeft :
      begin
        if DrawingTool = dtLine then begin
          LineDrawing := true;
          Item := TDrawLineUndoItem.Create(Image1.Canvas, Rect(x,y,x,y),
            DrawingTool);
        end else
          Item := TDrawShapeUndoItem.Create(Image1.Canvas, Rect(x,y,x+50,y+50),
            DrawingTool);
        if UndoStack.Submit(Item) = ssFull then
          ShowMessage(Format(sStackFull,[UndoStack.MaxItems]));
      end;
  end;
end;

procedure TGraphicWindow.Image1MouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin
  if LineDrawing then begin
    MouseMoved := true;
    with UndoStack.CurrentItem as TDrawLineUndoItem do
      AddSegment(Point(x,y));
  end;
end;

procedure TGraphicWindow.Image1MouseUp(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if LineDrawing and (Button = mbLeft) then begin
    if MouseMoved = false then
      UndoStack.RemoveLastItem
    else with UndoStack.CurrentItem as TDrawLineUndoItem do
      AddSegment(Point(x,y));
    LineDrawing := false;
    MouseMoved := false;
  end;
end;
```

➤ *Listing 6*

the nature of the Windows edit control introduces more problems. In the graphics drawing window our program had direct control of doing the drawing. But, in a TMemo, Windows takes care of entering the characters. All we can do is watch what is happening then try to take some action after the event. Any command object we wrap around this will not actually enter any text itself, it will just store the changes to the text, then reverse them later if Undo is called.

As with the line drawing object, our text command object must store a whole series of characters that are typed. In this case it will look for keys being pressed then, if text was entered, that text is added to the current command object with an AddText method.

If you select some text and start typing, the new text replaces the old. Therefore, our typing command object must not just store text that is typed, it should also be prepared to store any text that has been replaced.

We will create two text undo objects, as shown in Listing 7. The first is TClearingUndoItem, a command object for deleting text. It has a field called DeletedText where the text is stored, as well as a StartPos field holding the cursor position of the start of the text, and an Editor field that points to the appropriate TMemo component. A descendant of this called TTypingUndoItem is then declared. This adds an InsertedText field and a CurPos field that stores the cursor position after each character is added to InsertedText.

The DoCommand method of these classes is simply an empty procedure, since Windows does the entering of the text. Instead, any text deleted (or the initial character of text being typed) is passed to the object through the constructor.

With TClearingUndoItem that is all there is, since clearing text is a discrete event, not an extended one like typing or line drawing. Reinserting deleted text can most easily be done under 32-bit Delphi using the Text property of the TMemo (in Delphi 1 you can avoid the 255 character limit by using GetTextBuf and SetTextBuf). The text is simply put back at the appropriate place of the memo text using the Insert procedure. The cursor position is also reset through the SelStart property and the text is scrolled to bring the cursor into view, if necessary, using the EM_SCROLLCARET message. Redoing is similar but text is instead removed with the Delete procedure.

With a TTypingUndoItem object the initial inserted text (usually the first character typed) and any replaced text is passed to the constructor. This then calls AddText, which is also invoked by the keytrapping routine each time a subsequent character is typed. In most cases this adds the text to the end of the existing InsertedText string. If a carriage return is detected it is added along with a linefeed character. This is necessary for the lines to be separated properly when the text is put back with a Redo.

If the character is a backspace then the last character in InsertedText is deleted. In some cases InsertedText may be empty because we are backspacing over text typed in a previous action. In this case the text is added to the DeletedText field. Undo and Redo are similar to those in TClearingUndoItem, except that the use of Insert and Delete are reversed. When redoing we must check if StartPos is greater than CurPos (which happens when text is being deleted with the backspace). We must ensure that the lower of the two is passed to Delete.

Code showing how to use these two command objects is in Listing 8. As with the line drawing command object, we must have a boolean variable, Typing, to keep track of whether we are currently typing some text. We watch what the user is doing in the TMemo control by trapping key down events in the Memo1KeyDown method. First we look to see if the key is the delete key. If so we create a TClearingUndoItem, passing it either the selected text or the character to the right of the cursor. Otherwise we use ToASCII to check if the key pressed was a typed character of some sort.

```
constructor TClearingUndoItem.Create(AEditor : TMemo;
  ADeletedText : string; APosition : integer);
begin
  inherited Create;
  StartPos := APosition;
  DeletedText := ADeletedText;
  Editor := AEditor;
end;
procedure TClearingUndoItem.DoCommand;
begin
  ;
end;
procedure TClearingUndoItem.Undo;
var
  TempText : string;
begin
  TempText := Editor.Text;
  Insert(DeletedText,TempText,succ(StartPos));
  Editor.Text := TempText;
  Editor.SelStart := StartPos;
  Editor.Perform(EM_SCROLLCARET,0,0);
end;
procedure TClearingUndoItem.Redo;
var
  TempText : string;
begin
  TempText := Editor.Text;
  Delete(TempText,succ(StartPos),length(DeletedText));
  Editor.Text := TempText;
  Editor.SelStart := StartPos;
  Editor.Perform(EM_SCROLLCARET,0,0);
end;
constructor TTypingUndoItem.Create(AEditor : TMemo;
  AInsertedText,ADeletedText : string; APosition : integer);
begin
  inherited Create(AEditor,ADeletedText,APosition);
  AddText(AInsertedText,APosition);
end;
procedure TTypingUndoItem.AddText(AText : string2;
  APos : integer);
const
  BackSpace = #08;
  CR        = #13;
  LF        = #10;
var
  Temp : integer;
begin
  if AText = CR then begin
    AText := AText + LF;
    CurPos := APos + 2;
  end else if AText[1] = BackSpace then begin
    if APos > 0 then begin
      if InsertedText = '' then
        Insert(Editor.Text[(APos)],DeletedText,1)
      else
        Delete(InsertedText,length(InsertedText),1);
      CurPos := pred(APos);
    end;
  end else begin
    InsertedText := InsertedText + AText;
    CurPos := succ(APos);
  end;
end;
procedure TTypingUndoItem.Undo;
var TempText : string;
begin
  TempText := Editor.Text;
  Delete(TempText,succ(StartPos),length(InsertedText));
  if DeletedText <> '' then
    Insert(DeletedText,TempText,succ(StartPos));
  Editor.Text := TempText;
  Editor.SelStart := StartPos;
  Editor.Perform(EM_SCROLLCARET,0,0);
end;
procedure TTypingUndoItem.Redo;
var TempText : string;
begin
  TempText := Editor.Text;
  if DeletedText <> '' then
    if StartPos > CurPos then
      Delete(TempText,succ(CurPos),length(DeletedText))
    else
      Delete(TempText,succ(StartPos),length(DeletedText));
  Insert(InsertedText,TempText,succ(StartPos));
  Editor.Text := TempText;
  Editor.SelStart := CurPos;
  Editor.Perform(EM_SCROLLCARET,0,0);
end;
```

➤ *Listing 7*

ToASCII is a Windows API function that takes a virtual key code and the keyboard state and tries to convert it to the ASCII value of a character. Many keys, such as function or cursor keys, do not correspond to ASCII characters, so the result from ToASCII will be zero. This function is discussed in more detail in my article in Issue 26 of *The Delphi Magazine*.

If the key pressed is a regular character key then we check if we are in typing mode. If not, we create a TTypingUndoItem, passing it this first character typed. It is added to the stack and Typing is set to True. If we are already typing then we add the new character to the undo item with AddText. If any other type of key was pressed (such as a cursor key) then we call EndTyping, which simply sets Typing to False. EndTyping should also be called when any other action that should punctuate a typing event occurs; these could include clicking the mouse button or changing the font.

Descriptions

Our command objects have various properties (for example UndoDescription and RedoMenuText) that return descriptive text about their function (see Listing 1). We can access short or long descriptions of the undo and redo actions as well as text designed to be used as the Edit menu entries. In some cases, such as in the abstract base class and the line drawing class, we simply return a static string. TShapeDrawingUndoItem returns descriptions including the shape that has been drawn; these are produced using the Delphi Format procedure and appropriate constants or resourcestrings. The text command classes return strings incorporating the first few characters of the text that was typed or deleted. Now we must do something useful with these strings.

In the example program on the disk TUndoStack has a method called SetUndoMenuItems. This takes two TMenuItems as its parameters then makes the appropriate assignments to the Caption and Hint properties, retrieving descriptions from the current undo and redo objects. It also enables the menu items and sets their OnClick event handlers to call the stack's Undo and Redo methods. Also in the same unit as the undo stack is a global procedure called DisableUndoMenus. This disables the menu items and OnClick function, sets the menu text to generic Undo and Redo entries, and sets the hints to a string that says Command not available; nothing to undo.

When a new MDI application is created from a Delphi template it includes a method called UpdateMenuItems. This enables or disables various menu items depending on whether any child windows are open. It is in this method that our example program calls SetUndoMenuItems and DisableUndoMenus.

One problem with the default UpdateMenuItems method is that it is only called when the active window is changed. However, we need to have it updated whenever the undo stack changes. To do this we add a couple of new methods to the main form. First we create a message handler method to trap the Windows WM_INITMENUPOPUP message. This indicates that a menu is about to popup, so we can

```
procedure TTextWindow.Memo1KeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
var
  KeyState : TKeyboardState;
  Buffer : array[0..2] of char;
  ToASCIIResult : integer;
  Item : TUndoItem;
begin
  Item := nil;
  with Memo1 do
    if (SelLength > 0) and (key in [VK_Delete,VK_Back]) then begin
      EndTyping;
      Item := TClearingUndoItem.Create(Memo1,SelText,SelStart);
    end else if key = VK_Delete then begin
      Item := TClearingUndoItem.Create(Memo1,Text[succ(SelStart)],SelStart);
    end else begin
      FillChar(Buffer,sizeof(Buffer),0);
      GetKeyboardState(KeyState);
      ToASCIIResult := ToASCII(Key,MapVirtualKey(Key,0),KeyState,Buffer,0);
      if ToASCIIResult > 0 then begin
        if not Typing then begin
          Item := TTypingUndoItem.Create(Memo1,StrPas(Buffer),SelText,SelStart);
          Typing := true;
        end else
          with UndoStack.CurrentItem as TTypingUndoItem do
            AddText(StrPas(Buffer),(SelStart));
      end else begin
        EndTyping;
      end;
    end;
  if Item <> nil then
    if UndoStack.Submit(Item) = ssFull then
      ShowMessage(Format(sStackFull,[UndoStack.MaxItems]));
end;

procedure TTextWindow.EndTyping;
begin
  if Typing then
    Typing := false;
end;
```

➤ *Listing 8*

take any action we need to make sure the items on that menu are accurate. The method simply calls `UpdateMenuItems` followed by a call to the inherited method.

This works fine when using just the menus, but you may also want to have keyboard shortcuts for `Undo` and `Redo`. When the shortcuts are pressed the menu isn't shown, so `WM_INITMENUPOPUP` doesn't get sent. This may leave the menu items disabled and the shortcuts unusable. To ensure the menu items are always up to date we can use the `Application.OnIdle` event. This is called whenever the program finishes processing all its waiting messages and goes into an idle state. We point this event to our `IdleHandler` method in `FormCreate`. Then within the method we call `UpdateMenuItems`.

New Model Army?

The above classes implement a multiple undo facility that follows the model used by most programs. A single stack of undo objects is used for each child window and the stack is sequential. If you want to undo the changes you made a while ago you must undo all the intervening changes too. It sometimes is debatable whether it is more work to retype the text you accidentally deleted a while ago or to reimplement all the other changes you've made since. This can be particularly annoying if the intervening changes would have had no effect on the retyped text; they may have been complicated formatting changes to other areas of the text.

One solution to this is what Alan Cooper, in his book *About Face – The Essentials of User Interface Design* (IDG Books, 1995) calls category-specific undo. Instead of having all undo actions on a single stack, several stacks are created, one for each class of operation. A text editor could have one stack that saves all the typing and deleting commands while another saves the formatting commands. Then you can undo changes to each independently. This may seem ideal and in some situations would work well. However, the changes in the two stacks might not necessarily be independent.

Let's say you've typed some text, then changed its formatting to italic. You then choose to undo typing of the text, which is stored on the typing command stack, then choose to undo the change to italic from the formatting stack. The undo object will try to set a block of text back to roman letters, but that block is no longer there. What will probably happen is that the text that is now at the position where the text was undone will be converted to roman, which may not be desirable. Alternatively, some mechanism must be in place so that the formatting undo object knows that the text is no longer there; in this case it does nothing. However, this can quickly become a very complicated and error prone scenario because the text that was formatted may overlap two or more separately typed sections of text.

Another problem is one of user interface. You may feel that your program would be best served with five separate stacks of command objects for five different types of functions. How do you present this to the user? You could have five undo and five redo items on the `Edit` menu but that is very unwieldy. You could have a submenu of five entries off the `Edit | Undo` menu item and likewise for the `Redo` menu. But, some users have problems manoeuvring through nested submenus. It could be done somehow through a dialog box. At any rate, it makes the undo function much more difficult to access; a simple `Ctrl-Z` is so much easier.

I personally don't know of any programs that use category-specific undo (please let me know if you know of any!). It could be an attractive way to avoid the problems with sequential multiple undo but has its own problems. It is something that needs more development and experimentation.

Warren Kovach doesn't believe in the phrase 'What's done is done.' He also writes and sells statistical software and is the author of *Delphi 3 User Interface Design*, published by Prentice Hall. Email Warren at wlk@kovcomp.co.uk or visit www.kovcomp.co.uk