Splat: Sound And Graphics With Delphi

by Ray Lischner

I recently searched the web for computer games and toys for my infant son and was disappointed by the tiny selection for children that young. As a software developer, my natural response was to write my own game, which I call Splat (the reason for the name will become clear next month). It turns out that Splat touches on many aspects of Windows and Delphi programming, from multimedia to internationalization. It is an example of Delphi put to practical use. This is the first of two articles to explore Splat and its design.

Splat is a simple toy. It takes over the screen, showing a black background. When you press any key or mouse button, it displays a colored shape and plays a sound. That's all. Repeat until bored. Figure 1 shows a screenshot.

To keep things interesting, once a shape appears it slowly grows and moves across the screen. New shapes are drawn on top of old ones, and old shapes fade to black. It's so simple that even a child can play it. At four months, my son has a lot of fun pounding on the keyboard, watching the shapes and hearing the sounds. Older children can learn about cause and effect: pressing the same key always produces the same sound. Children can 'play' the keyboard as a musical instrument to produce their favorite sounds.

To appeal to the parents, Splat has an educational aspect. When you press a letter or digit, the sound that it makes is the name of that letter or digit. Children can practise their alphabet and numbers. Using Delphi's multiple-language support, you can easily add other languages, so the child or adult can practise French, Spanish, German, Greek or even Russian (provided you have the right keyboard).

This article presents the architecture of the Splat program: how it plays sounds and draw shapes. It also presents a simple sound recorder to help record the sound files for letters and digits. The next article will examine international issues, add compression for the sound files, and more refinements.

The Splat Framework

The basic framework for Splat is simple. The main form takes over the screen and defines handlers for keyboard and mouse events. When the user clicks a mouse button, Splat plays a random sound and adds a shape at the mouse position. When the user presses a key, Splat plays a sound that corresponds to the key and adds a shape at a random position.

A master list keeps track of all shapes. To draw the screen, the shape list draws each of the shapes in the list. Newer shapes are drawn on top of older shapes. A double buffer prevents screen flicker: the master list draws to an off-screen bitmap and, when the bitmap is complete, it's drawn on the screen.

A timer redraws the screen every 0.25 second. After drawing all the shapes in the list, every shape produces its next generation. Each shape class dictates what a generation change really means, but in most cases it means the shape's position, size, and color change slightly. After each generation, any shape that becomes invisible, by moving off the screen or by fading entirely to black, is removed from the list.

The form has a `TImage` whose `Align` property is set to `alClient` so that it takes over the form. Thus, all mouse events go to the image, and it handles the `OnMouseDown` events. Splat draws a shape at the mouse position, and plays a sound chosen at random.

When the user presses a key, Splat draws a shape at a random position on the screen and plays a sound that corresponds to that key. Rather than using a simple `OnKeyDown` handler, Splat takes special steps to handle keystroke events. With a child pounding on the keyboard, you never know what keys will be pressed. `Alt+Space` by default pulls down the system menu, for example. To avoid surprises, Splat intercepts all keystroke events before they reach any control. Thus, `Alt+F4` does not close the main form and the application. To intercept all keystroke messages, Splat uses a `TApplicationEvents` component and sets an `OnMessage` event handler. For all keystroke events (`Wm_KeyDown`, `Wm_SysKeyDown`, etc), Splat sets `Handled` to `True` and returns immediately. To exit the program, the user must press `Escape`. Listing 1 shows the mouse and keyboard event handlers.

Playing Sound Resources

All the sounds are originally stored in .WAV files, which are linked as `WAVE` resources. Not many Delphi users know that you can include a resource script (.RC file) as part of a project, and Delphi 5 will automatically compile the script into a .RES file when you build an application. To take advantage of this feature, just add the .RC file to the project. View the project source to see the new `$R` compiler directive.

The format of an .RC file is flexible. Splat uses the simplest form: each line describes a resource as a resource identifier, a resource type (`WAVE`) and a file name that contains the resource data. A

➤ *Figure 1*

```
// Create new shape at (X,Y) or generate random position
procedure TMainForm.CreateShape(X, Y: Integer);
begin
  if X < 0 then
    X := Random(Width);
  if Y < 0 then
    Y := Random(Height);
  ShapeList.AddShape(X, Y);
  RedrawShapes;
end;
// Intercept all keystroke events and play a WAVE file for
// each key press without interpreting the key event.
procedure TMainForm.AppEventsMessage(var Msg: tagMSG; var Handled: Boolean);
begin
  case Msg.Message of
  Wm_KeyDown, Wm_SysKeyDown:
    begin
      // Handle key down events by playing a sound and drawing a shape.
      HandleKeyDown(Msg.wParam);
      Handled := True;
    end;
  Wm_DeadChar, Wm_Char, Wm_KeyUp, Wm_SysKeyUp:
    Handled := True;     // Ignore up and other key events
  else
    {Skip};
  end;
end;
// Pick a WAVE file to play based on the key that the user pressed.
procedure TMainForm.HandleKeyDown(KeyCode: Word);
begin
  if KeyCode = Vk_Escape then
    Close
  else begin
    PlayWave(KeyCodeToText(KeyCode));
    CreateShape;
  end;
end;
// When the TImage gets a mouse down event, generate a new
// shape at the mouse position, and play a random sound.
procedure TMainForm.ImageMouseDown(Sender: TObject; Button:
  TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  CreateShape(X, Y);
  PlayRandomWave;
end;
```

➤ *Listing 1*

quick search on the web turns up a number of sources of free .WAV files. Most of them seem to be illegal copies of copyrighted material, but two websites offer free sample downloads of commercial sound effect products. You can download the files from:

```
www.novadevcorp.com/products/
  kaboomwin/index.html
www.softseek.com/Home_Family_
  and_Leisure/Music/
  Sound_File_Collections/
  Review_6120_index.html
```

The other sound files in Splat were made with the Recorder utility, which I will describe later in this article. Listing 2 shows excerpts from the SoundRes.rc script.

Windows resource names and types can be strings or numbers. To distinguish between string names and numeric identifiers, Windows plays a dirty trick. It always uses a `PChar` for a resource identifier and, if the high word is zero, the low word is the numeric identifier. Otherwise, the entire pointer is a plain zero-terminated string. In a Delphi program, it is much easier to work with strings exclusively, but there is no reason to restrict the programmer from using numeric resource identifiers. Two functions hide the details by mapping strings to numbers as needed. The string form of a numeric resource identifier is a hexadecimal number, eg `$001A`, making it easy to convert the string back to a number. Listing 3 shows the functions to convert between strings and resource identifiers.

In order to find the sound that corresponds to the key that the user presses, Splat looks up a sound resource whose name is the same as the key's name. For example, the letter A has the textual representation of `A`. Digits have the name `DIGIT0` or `NUMPAD0` depending on whether the key pressed is one of the the digits above the letters or on the numeric keypad. Special keys have representative names, such as `F1` or `PgUp`. The `KeyCodeToText` function takes a virtual key code and returns a string representation of the key. As you will learn later in this article, you don't usually need to know the actual names that Splat uses because a companion program creates the .WAV files for you, using file names that match the corresponding resource names. Listing 4 shows the `KeyCodeToText` function.

The `PlayWave` function plays the named `WAVE` resource by calling `PlaySound` (in the `MMSystem` unit). Windows offers many different ways to play a sound resource, but `PlaySound` is the simplest: it can play a .WAV file, a `WAVE` resource, any of the system sound effects, or a waveform in memory. You can choose to wait until the sound finishes playing before continuing, or let the program continue running while the sound plays. Almost

➤ *Listing 2*

```
A WAVE "Sounds\A.WAV"
...
Z WAVE "Sounds\Z.WAV"
...
DIGIT0 WAVE "Sounds\0.WAV"
...
DIGIT9 WAVE "Sounds\9.WAV"
```

➤ *Listing 3*

```
// Return a string for a resource name or identifier. A resource name can be a
// string or a numeric identifier. Convert a numeric identifier to a string as a
// hexadecimal constant (e.g., $12A). The dollar sign makes it easy to convert
// back to a number and to distinguish a numeric ID from a string name.
function ResIDToString(ResName: PChar): string;
begin
  if LongRec(ResName).Hi = 0 then
    Result := Format('$%x', [Integer(ResName)])
  else
    Result := ResName;
end;
// Convert a string back to a resource identifier.
function StringToResID(const ResText: string): PChar;
var ID: Word;
begin
  if (ResText = '') or (ResText[1] <> '$') then
    Result := PChar(ResText)
  else begin
    ID := StrToInt(ResText);  // Make sure the ID is within the proper bounds.
    Result := PChar(ID);
  end;
end;
```

```
// Return a text representation for a virtual key code.
function KeyCodeToText(KeyCode: Word): string;
begin
  case KeyCode of
  Ord('0')..Ord('9'):
    Result := 'Digit' + Chr(KeyCode);
  Ord('A')..Ord('Z'):
    Result := Chr(KeyCode);
  Vk_NumPad0..Vk_NumPad9:
    Result := 'NumPad' + Chr(Ord('0') + KeyCode - Vk_NumPad0);
  else
    Result := ShortCutToText(KeyCode);
    // If name not a valid resource name, use plain hexadecimal representation
    if (Result = '') or (Pos(' ', Result) > 0) or not (Result[1] in
      ['a'..'z','A'..'Z']) then
      Result := Format('Char%.2X', [KeyCode]);
  end;
end;
```

➤ *Above: Listing 4*

➤ *Below: Listing 5*

```
// Play the named WAVE resource. The resource might be located in
// the locale-specific DLL or in the main application. Try the DLL
// first, then the application. If all else fails, use a default beep.
procedure TMainForm.PlayWave(const Name: string);
var ResName: PChar;
begin
  ResName := StringToResID(Name);
  if not PlaySound(ResName, hInstance, Snd_Resource or Snd_ASync) then
    Beep;
end;
```

every sound card can easily play a sound while a program runs, so Splat opts to continue. Use the `Snd_Async` flag to play the sound asynchronously, and the `Snd_Resource` flag to load the `WAVE` resource.

If the user presses a key before the sound finishes playing, the next call to `PlaySound` stops the old sound and starts the new one. The details of calling `PlaySound` are encapsulated in the `PlayWave` function so future enhancements can be localized to the `PlayWave` procedure without affecting the rest of the application. If Splat cannot find a sound for the key, `PlaySound` plays a default sound, and if `PlaySound` returns an error, Splat tries one last time by calling Delphi's `Beep` procedure. Listing 5 shows the `PlayWave` function.

When the user clicks a mouse button, Splat chooses a sound at random. To do that, it must have a list of all the `WAVE` resources in the file. The form's `OnCreate` event handler fills a `TStringList` with the names of all the `WAVE` resources. (Here is an example of the utility of having all the resource identifiers as strings. They can all be added to the string list without further ado, which would be much harder to manage if Splat used the Windows trick of mixing numbers and `PChar` pointers.) As with any Windows callback function, `EnumWaves` must use the `stdcall` calling convention. Windows passes an arbitrary pointer parameter, so `EnumWaves` uses a `TStrings` parameter, the list of wave resources. To play a random sound, `PlayRandomWave` picks a resource name at random.

Drawing Shapes

When the user presses a key or mouse button, Splat displays a random shape. Splat can display several different kinds of shapes. Each shape is an instance of a shape class that derives from `TShape` (not the `TShape` in the VCL, but a new `TShape` that is part of the Splat program, there are only so many names to go around). The main application never deals with the individual shape classes, though. It creates a `TShapeList` object and the shape list takes care of the details of creating and managing shapes. This insulates the application from changes to the implementation of the shapes. The framework lets you add new shapes easily and alter the behavior of the shapes, without changing any code in the main application. A registration procedure lets you add new shapes without modifying the main Shapes.pas source file. Just add a new unit to the project and call `RegisterShapes`.

Listing 6 shows the `TShapeList` class declaration. The shape list needs to know the screen size so it can tell when shapes leave the screen and are no longer visible. Rather than hardcoding a direct reference to the `Screen` object, `TShapeList` gets its bounds from the main application. Flexibility is always a handy asset, as you'll see later. The `AddHelp` method adds a special shape, `THelp`, which displays the message *'Press ESC to exit the program'*. The application can call `AddHelp` anytime to help users who may not know how to quit.

Each shape class inherits from `TShape`. Many of the methods of `TShape` are protected and virtual, so derived classes can customize the behavior. The default behavior is that a shape starts small and grows with each generation. When the shape is created, it starts with a small, random, velocity in the form of a `TPoint`. The point is added to the shape's position for each generation.

The initial color for most shapes is also random. To keep the color bright and interesting, a random

TShapeList

`TShapeList` inherits from `TObjectList`, which seems to be poor programming style. Good style dictates that the object list should be a private field of `TShapeList`, so the shape list can delegate work to the object list without exposing the list contents to the outside world. This exposure is potentially unsafe, because the application could add non-shape objects to the list, which would be very, very bad. In a simple application such as Splat, the lack of safety is outweighed by the simplicity of inheritance. As the application grows more complex, you should consider changing the implementation of `TShapeList`. The new class would hide the `TObjectList` as a private field and implement simple methods, such as `GetCount`, by getting the object list's `Count` property. The shape list's `Add` method would take a `TShape` object as a parameter, to help ensure the validity of the list's contents. If you are careful, you should be able to change the implementation without affecting its public interface or the way the application uses the shape list. That's the power of object-oriented programming.

hue is chosen in a hue, saturation and value color space instead of choosing random red, green and blue values, which would result in lots of boring colors. Colors fade to black over successive generations, so new shapes are easier to see than older shapes.

To tell when a shape has moved off the screen, each shape also keeps track of its bounding box (the smallest rectangle that contains the entire shape). For simplicity, some shapes do not compute the smallest bounding box, but use a small-enough rectangle that is easier to determine. The extra size means a few shapes live longer than necessary, but they do not cause any problems.

Finally, every shape must be able to draw itself on a canvas. `TShape` declares the `Draw` method as an abstract virtual method, so derived classes must override it. Listing 7 shows the declaration for the `TShape` base class.

The simplest shape is the ellipse. It uses the `TCanvas.Ellipse` method to draw itself, after setting the brush and pen colors to the shape's desired color. All other behavior it inherits from `TShape`. Listing 8 shows the `TEllipse` class.

The `TPolygon` class implements a simple polygon, which can have between 3 and 12 vertices. Any more than 12 vertices and the polygon looks more like an ellipse. The number of vertices is chosen randomly when the polygon object is created. The `TPolygon` class is also quite simple, as you can see in Listing 9. Other shapes are just as easy to define. A four-sided `TPolygon` is drawn as a diamond, so a separate `TRectangle` class implements a basic rectangle. See the companion disk for the complete source of `TRectangle` and other shapes.

Displaying The Shapes

Splat starts by taking over the entire screen and displaying a black background. The shape list is initialized with the help message. To make the form cover the entire screen, the `BorderStyle` property is set to `bsNone`, and the form's `OnCreate` handler sets the form size to fill the screen (`Screen.Width` by `Screen.Height`), which makes sure the form covers the task bar. The handler also calls the Windows API function `SetWindowPos`, with `Hwnd_TopMost` as the insert-after parameter. This forces Splat's window to be shown on top of all other windows. Unfortunately, Windows does not strictly enforce the topmost attribute (in part because other windows vie for that status). Splat must play a number of tricks to remain on top in all situations.

The simplest way for Splat to stay on top of things, so to speak, is to call `SetWindowPos` when any other window becomes active. Set the application's `OnDeactivate` event handler. (The form's `OnDeactivate` handler is called when another form in the same application becomes active. You must set the application's `OnDeactivate` handler to learn when another application becomes active.) Splat already has a `TApplicationEvents` component for intercepting keystroke events. Just add an `OnDeactivate` handler (Listing 10).

You run into a problem on Windows 98 and Windows 2000 (the operating system formerly known as NT 5). These operating systems have a feature that is usually friendly, but in this case gets in the way. The problem is that one thread cannot steal the keyboard focus from another thread. In normal working conditions, this is an excellent feature, which

➤ *Listing 6*

```
type
  TShapeList = class(TObjectList)
  private
    fHeight, fWidth: NaturalInt;  // screen size
    function GetShape(Index: NaturalInt): TShape;
  public
    class function AnyShapeClass: TShapeClass;
    constructor Create(Width: NaturalInt = 0; Height: NaturalInt = 0);
    procedure AddShape(X, Y: NaturalInt);  // Create random shape and add to list
    procedure AddHelp;                     // Add Help text to center of screen
    procedure Draw(Canvas: TCanvas);       // Draw all shapes on canvas
    procedure NextGeneration;              // Iterate next generation of shapes
    // Get the shapes in the list
    property Shapes[Index: NaturalInt]: TShape read GetShape; default;
    property Height: NaturalInt read fHeight;
    property Width: NaturalInt read fWidth;
  end;
```

➤ *Listing 7*

```
type
  // Abstract base class for all shapes. Maintain position
  // of shape's center, size, and color. Each generation,
  // fade color and move shape. When color becomes black, or
  // when shape moves off screen, the list deletes it.
  TShape = class
  private
    fColor: TColor;   // Color of shape
    fDelta: TPoint;   // Position change for each generation
    fPosition: TPoint; // Center of shape
    fSize: TSize;
  protected
    constructor Create(Position: TPoint); virtual;
    // Randomly change the shape's color. The default is to
    // fade towards black.
    procedure ChangeColor; virtual;
    // Draw this shape on the canvas, at the current
    // position, using the current color. Derived classes
    // must override this method.
    procedure Draw(Canvas: TCanvas); virtual; abstract;
    // Randomly change the size of the shape.
    procedure ChangeSize; virtual;
    // Generate a random position Delta, can be +ve or -ve
    procedure GenerateDelta; virtual;
    // Get the shape's bounding box.
    procedure BoundingBox(var Rect: TRect); virtual;
    function GetBottom: Integer; virtual;
    function GetLeft: Integer; virtual;
    function GetRight: Integer; virtual;
    function GetTop: Integer; virtual;
    // Return True if the color is not black and if the
    // shape's bounding box is still visible on the screen.
    function IsVisible(Width, Height: NaturalInt): Boolean;
      virtual;
    // Move the shape's position by its delta.
    procedure Move; virtual;
    // Generate the next generation by fading the color.
    procedure NextGeneration(Width, Height: NaturalInt);
      virtual;
  public
    property Color: TColor read fColor write fColor;
    property Delta: TPoint read fDelta write fDelta;
    property Position: TPoint read fPosition
      write fPosition;
    property XPosition: Integer read fPosition.X
      write fPosition.X;
    property YPosition: Integer read fPosition.Y
      write fPosition.Y;
    property Size: TSize read fSize write fSize;
    property XSize: Integer read fSize.CX write fSize.CX;
    property YSize: Integer read fSize.CY write fSize.CY;
    property Left: Integer read GetLeft;
    property Right: Integer read GetRight;
    property Top: Integer read GetTop;
    property Bottom: Integer read GetBottom;
  end;
```

prevents one program from popping up a window and stealing the keystrokes that you are furiously typing into a word processor. Splat, however, is special and needs to keep the keyboard focus no matter what else is going on. The solution is to attach Splat's main thread to the keyboard input of whatever thread happens to have the keyboard input. Then Windows will let Splat become the foreground window, and it can detach the keyboard input from the other thread. Listing 11 shows the `ForceForegroundWindow` function that accomplishes this magic (thanks to Karl E Peterson, who posted this workaround to www.mvps.org/vb/samples.htm).

Activation and keyboard focus are two distinct notions in Windows. Another program can gain the keyboard focus without deactivating Splat. This happens, for example, when the user presses the `Windows` key to bring up the `Start` menu. Splat can retain its active status, but still lose the keyboard focus on Windows 98. There is no simple way for Splat to prevent the `Start` menu from appearing, so instead, it calls `ForceForegroundWindow` each time it redraws its shapes. Even if Windows tries to display the `Start` menu, Splat seizes control and keeps splatting.

Very quickly, you run into another problem. How do you debug an application that forces its window to be on top of all other windows, including Delphi's IDE? A simple solution is to call `IsDebuggerPresent`, and if the application is running under control of a debugger, do not call `SetWindowPos`. The only hitch is that Windows 95 does not have `IsDebuggerPresent`, but that isn't a major problem. Delphi has the `DebugHook` variable, which is non-zero when the app is running in Delphi's integrated debugger. Splat, therefore, implements its own `IsDebuggerPresent` function, which simply returns `True` when `DebugHook` is not zero.

When running in the debugger, Splat does not cover the screen, so you can see the form and the source code at the same time. That's why it's helpful that `TShapeList` keeps track of the size where it is running. Sometimes that's the screen size and sometimes it isn't. The shape list could have kept a reference to the form and used the form's size, but there is no other reason for the shape list to know anything about the form. It's simpler for the shape list to keep track only of the form's size.

The form has a `TTimer` component to create successive generations of shapes. When the `OnTimer` event fires, the event handler first makes sure the application is still the topmost, foreground window (but only if it is not running in the debugger). Next the event handler calls the shape list's `NextGeneration` method. Finally `RedrawShapes` clears the double-buffer bitmap, draws all the shapes on the bitmap, and displays the bitmap on the `TImage`. Listing 12 shows these methods and event handlers.

To initialize the application, Splat's `OnCreate` handler must fetch the list of `WAVE` resources and create the double-buffer bitmap. To ensure the shapes and sounds are different every time the program runs, it also calls `Randomize` to seed the pseudo-random number generator. When running in the debugger, though, it does not call `Randomize`, so you always get a repeatable pattern of sounds and shapes during development.

Recording New Sounds

Now that the basic Splat program runs, it's time to improve the sound effects. As you saw in Listing 5, any key that does not have a matching `WAVE` resource plays a default beep. The goal is to have a different sound for every key. For the educational part of the program, the sounds for letters and digits should be a person saying that letter or digit. You can use any

```
type
  TEllipse = class(TShape)
  public
    procedure Draw(Canvas: TCanvas); override;
  end;
procedure TEllipse.Draw(Canvas: TCanvas);
begin
  Canvas.Brush.Color := Color;
  Canvas.Pen.Color := Color;
  Canvas.Ellipse(Position.X - XSize div 2, Position.Y - YSize,
    Position.X + XSize, Position.Y + YSize);
end;
```

➤ *Above: Listing 8*

➤ *Below: Listing 9*

```
type
  // Regular polygon with 3-12 vertices, starts small, grows with each generation
  TPoints = array of TPoint;
  TPolygon = class(TShape)
  private
    fNumVertices: PositiveInt;
  public
    constructor Create(Position: TPoint); override;
    procedure Draw(Canvas: TCanvas); override;
    property NumVertices: PositiveInt read fNumVertices;
  end;
const
  MinVertices = 3;
  MaxVertices = 12;
// Bounding box not always smallest bounding box, but an adequate approximation
constructor TPolygon.Create(Position: TPoint);
begin
  inherited;
  fNumVertices := Random(MaxVertices - MinVertices + 1) + MinVertices;
end;
procedure TPolygon.Draw(Canvas: TCanvas);
var
  I: Integer;
  Pt: TPoint;
  Points: array of TPoint;
  Angle: Single;
begin
  SetLength(Points, NumVertices);
  for I := Low(Points) to High(Points) do begin
    Angle := 2*Pi * I / Length(Points);
    Pt.X := Round(Position.X + XSize * Cos(Angle));
    Pt.Y := Round(Position.Y + YSize * Sin(Angle));
    Points[I] := Pt;
  end;
  Canvas.Pen.Color := Color;
  Canvas.Brush.Color := Color;
  Canvas.Polygon(Points);
end;
```

WAVE capture tool to record the sounds, but that's inconvenient when you want to record 36 short sound files. It's easier and more fun to write a custom sound recorder.

Windows has several ways to record .WAV files. You can use low-level wave audio functions, but it's easier to use the media control interface (MCI). MCI provides a small number of powerful functions for recording and playing back various multimedia formats, including sounds (waveaudio in MCI terms). The MCI functions (like PlaySound) are part of Windows' multimedia system, declared in the MMSystem unit.

The recorder waits for the user to press a key. The OnKeyDown event starts recording a waveform for that key. When you release the key, recording stops and the waveform is saved to a .WAV file. You can quickly record sounds for all the letters and digits. The recorder calls the same KeyCodeToText function shown in Listing 4 when it creates file names, so you know that the file names match the resource names that Splat expects.

The easiest way to use MCI is to call mciSendString, which takes a string parameter containing a command and arguments. For example, to open a new WAV file, use open new type waveaudio (ie, open a new MCI stream of type waveaudio).

The record command starts recording. You can work with multiple devices at once, so MCI requires a device with each command. The easiest way to identify a device is to supply an alias when you open it. Use the alias to identify the open device in subsequent commands. To create an alias, use the alias keyword at the end of the open command string (for example: open new type waveaudio alias keywave). After the alias keyword, list the alias name you want to use. You can then use that alias name in record, close, and other commands (eg, record keywave).

To remind the user what is being recorded, the recorder's main window has a status bar displaying the key name and the current mode (eg Recording). Listing 13 shows the key down/up event handlers, which start and stop recording.

The user *can* press multiple keys at once. In that case, the first key pressed is the one being recorded. The user can press and release any other keys, but recording continues until the first key is released. After recording stops, the user can press and record another key.

The rest of the recorder program manages the display of the WAV files in the current directory. The user can choose a detail view to see the size and modification date of each file. In the detail view, the user can sort by name, size, or date. See the source code on the companion disk for the complete details. Figure 2 shows the recorder in action.

After you have recorded your .WAV files, edit the SoundRes.rc file to include the new sound resources.

What's Next?

Now Splat is ready to use. You have sounds and shapes. The program takes over the screen and lets you play. We have yet to

➤ *Above: Listing 10*

```
// If another application tries to take control, bring attention back here.
procedure TMainForm.AppEventsDeactivate(Sender: TObject);
begin
  if not IsDebuggerPresent then
    Win32Check(SetWindowPos(Handle, Hwnd_Top, 0, 0, 0, 0,
      Swp_NoSize or Swp_NoMove));
end;
```

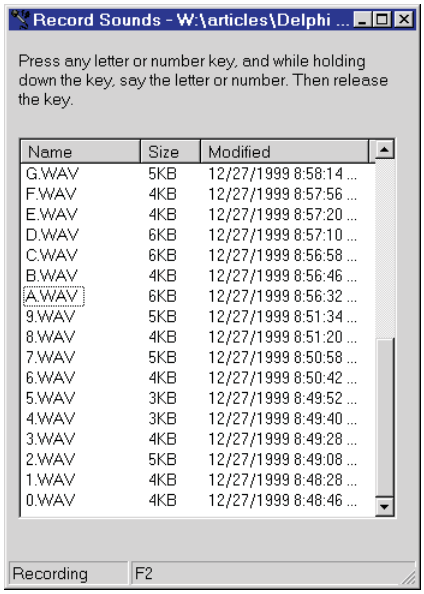
➤ *Below: Listing 11*

```
function ForceForegroundWindow(Handle: HWND): Boolean;
var
  Foreground: HWND;
  ForegroundThreadID, ThisThreadID: DWORD;
begin
  Foreground := GetForegroundWindow;
  if Foreground = Handle then
    Result := True
  else begin
    ForegroundThreadID := GetWindowThreadProcessId(Foreground, nil);
    ThisThreadID := GetWindowThreadProcessId(Handle, nil);
    AttachThreadInput(ThisThreadID, ForegroundThreadID, True);
    SetForegroundWindow(Handle);
    AttachThreadInput(ThisThreadID, ForegroundThreadID, False);
    Result := GetForegroundWindow = Handle;    // Return True if trick worked
  end;
end;
```

➤ *Listing 12*

```
// Periodically transform all the shapes into the next generation
// and redraw the shapes. Typically shapes grow and fade color.
procedure TMainForm.TimerTimer(Sender: TObject);
begin
  if not IsDebuggerPresent then begin
    ForceForegroundWindow(Handle);
    if Handle <> GetTopWindow(0) then
      SetWindowPos(Handle, Hwnd_Top, 0, 0, 0, 0, Swp_NoSize or Swp_NoMove);
  end;
  ShapeList.NextGeneration;
  RedrawShapes;
end;
// Draw all the shapes to a background bitmap, and replace the image's bitmap
// with the other bitmap. This use of a double buffer minimizes screen flicker.
procedure TMainForm.RedrawShapes;
begin
  DoubleBuffer.Canvas.Brush.Color := clBlack;  // Clear bitmap
  DoubleBuffer.Canvas.FillRect(Image.BoundsRect);
  ShapeList.Draw(DoubleBuffer.Canvas);         // Draw the shapes
  Image.Picture.Bitmap := DoubleBuffer;        // Display new bitmap
end;
```

➤ *Figure 2*



explore the full potential of Splat, though. For example, Delphi makes it easy to add language- specific resources to a project. Take advantage of this feature to substitute different languages when Splat plays letter and digit sounds. All the sound resources make the Splat.exe file rather large. The ZLIB compression unit that comes with Delphi lets you easily compress all the WAVE resources and shrink the executable to less than two-thirds of its original size. Splat needs other refinements, too. Press Windows+M, for example, to see one situation where Splat misbehaves.

These and other features will be the subject of next month's article.

Ray Lischner is the author of *Delphi In A Nutshell* and a number of other books and articles about Delphi (and Shakespeare). He also teaches computer science at Oregon State University. You can email Ray at delphi@tempest-sw.com

```
// Start recording a wave file for key the user has pressed.
// Record only first key pressed until user releases key.
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word; Shift: TShiftState);
resourcestring
  sCannotOpen = 'Cannot open WAV recorder:'#13#10;
  sCannotRecord = 'Cannot record WAV file:'#13#10;
begin
  if RecordKey = 0 then begin
    // Not already recording a sound, so start recording.
    MciCheck(mciSendString('open new type waveaudio alias wave',
      nil, 0, 0), sCannotOpen);
    MciCheck(mciSendString('record wave', nil, 0, 0), sCannotRecord);
    // Remember which key is being recorded, and update the status bar.
    RecordKey := Key;
    SetMode(sRecording);
    SetStatusInfo(KeyCodeToDisplay(Key));
  end;
end;
// Stop recording when user releases key. Make sure user is releasing key that is
// being recorded (in case user presses multiple keys).
procedure TForm1.FormKeyUp(Sender: TObject; var Key: Word; Shift: TShiftState);
resourcestring
  sCannotStop = 'Cannot stop recording WAV file:'#13#10;
  sCannotSave = 'Cannot save WAV file (%s):'#13#10;
  sCannotClose = 'Cannot close WAV recorder:'#13#10;
var
  FileName: string;
  Item: TListItem;
begin
  if Key = RecordKey then begin
    MciCheck(mciSendString('stop wave', nil, 0, 0), sCannotStop);
    // Save the waveform to a file.
    FileName := KeyCodeToText(RecordKey) + '.wav';
    MciCheck(mciSendString(PChar('save wave ' + FileName),
       nil, 0, 0), Format(sCannotSave, [FileName]));
    MciCheck(mciSendString('close wave', nil, 0, 0), sCannotClose);
    RecordKey := 0;
    SetMode('');
    SetStatusInfo(Format(sRecorded, [FileName]));
    // If the file is not already in the list, add it.
    Item := WaveList.FindCaption(0, FileName, False, True, True);
    if Item = nil then
      Item := AddWaveFile(FileName);
    Item.Selected := True;
  end;
end;
```

➤ *Listing 13*