Splat: Part 2, Going International

by Ray Lischner

Last month's article introduced the Splat program, a game for small children to pound on the keyboard and watch interesting shapes and hear fun sounds. This month, Splat goes international, goes on a compression diet, and gains some small but important features.

International Splat

Splat stores all its sounds in a resource file that is linked into Splat.exe. Delphi makes it easy to create language-specific resource files that are linked as separate DLLs, one DLL per language. It seems natural to use this feature in Splat. The sounds for letters and digits are the names of those letters and digits, so these sounds can easily be localized for different languages and countries. Before jumping in, though, take some time to understand how Windows handles keystrokes.

If you remove the printing on top of the keys, all Western keyboards are pretty much the same. Even Greek and Cyrillic keyboards are the same. The only significant differences between these keyboards are the characters printed on top of the keys. When you press a key, the keyboard sends a number, called the *scan code*, to Windows. The scan code that a key generates is the same regardless of what is printed on the key. Windows uses a *code page* to interpret the scan code and map it into a *virtual key code*. For example, in the United States, the key labelled Q (the upper-leftmost letter on the keyboard) generates scan code 16, which corresponds to virtual key code 81 (which is the ANSI value for the letter Q). In France, however, the same key represents the letter A. The scan code is the same (16), but the code page for French maps the scan code to virtual key code 65 (the ANSI value for the letter A).

Rarely does a program work directly with scan codes. Splat, like most programs, handles only virtual key codes. For example, when a French user presses A, Splat plays the sound for the letter A, and when an American user presses Q, Splat plays the sound for the letter Q. Splat doesn't know that both users pressed the same key. As you can see, Splat already does the right thing for handling different keyboard layouts. The next step is to have Splat play a different sound, depending on the virtual key code that it receives. First, though, you need to record the sounds in other languages.

The Recorder program records sounds according to the virtual key code, so you can use it to record sounds for any language, even with your own native keyboard. To make it easier to use, though, you should make the Recorder aware of the desired language, so it can display the correct character for the key being pressed. The display character depends on the code page for the desired language. Before getting too deep in the finer points of upgrading the Recorder program, it's time to examine Delphi's support for multiple languages more closely.

Localization

Internationalization is the process of generalizing an application to suit multiple locales. *Localization* is the act of taking an internationalized application and supplying the details for a specific locale.

The term *internationalization* often has the absurd abbreviation *i18n*. Less often, *localization* is abbreviated *l10n*. (The number tells you many letters have been omitted from the w2d to make the a10n.)

Internationalizing a Delphi application is easy:

- Use `resourcestring` declarations instead of string literals.
- Use position specifiers in `Format` strings (eg, `File %0:s not found: Windows error message is %1:s`) because different languages might need to change the order of the arguments.
- Leave room on a form for prompts and other text that might be a different length in a different language.
- Don't use `Val` or `Str` to convert numbers and strings, but use the routines in the `SysUtils` unit instead. The user can choose formats for dates, times, currency, and more in the `Regional Settings` Control Panel applet, and the `SysUtils` routines heed these settings.

Localizing an application usually involves translating `resourcestrings` and text on forms. Localizing Splat means translating the sounds it makes when you press the letter and digit keys. (You can also translate the initial help message: *'Press ESC to exit the program'*.) Splat stores its sounds as `WAVE` resources, and you can customize the letters and digits for different countries and languages. For example, you might want the Z key to play an American 'zee' in the United States and 'zed' in England. In Russia, the same key would be read as 'ya' (the 'Я' character in Cyrillic).

Delphi performs resource localization by storing locale-specific resources in a separate DLL. Each language, or language and country (to handle local dialects), has a unique file name extension (such as .FRA for standard French and .FRS for Swiss French). A project can have many different resource DLLs, each with its own language-specific extension. At runtime, the application loads a resource DLL based on the Windows locale, and gets its resources from that DLL instead of from the resources in the application's .EXE file.

Splat uses the resource DLL to store language-specific sounds. When Splat starts, Delphi automatically loads the resource

DLL. Splat passes the handle of the resource DLL to `PlaySound` instead of passing `hInstance`. It calls the `FindResourceHInstance` function to get the resource DLL handle.

In order to create the resource DLLs, first you need sound files in another language, which requires some changes to the Recorder.

Internationalizing The Recorder

The first task is to modify the Sound Recorder program to record the sounds for other languages. Two changes are needed: first, let the user choose a different keyboard layout and, second, when displaying the character that is being recorded, the program must display the character from the appropriate character set. You can record sounds for any language, even though you are using your own keyboard with its keytop labels that are appropriate only for your native language. If the user chooses the Russian keyboard layout, for example, pressing the Z key should display Я, so the user knows to say, 'ya'.

The first step is to install support for alternative keyboard layouts. In the `Keyboard` Control Panel applet, select the `Input Locales` tab. (Windows NT and 98 support the `Input Locales` tab. Windows 95 does not.) Click the `Add` button, choose a new locale, and click `OK`. You can add as many different locales as you want. Choose one locale to be the default. The `Keyboard Properties` dialog box also lets you choose a keyboard shortcut that changes the current input locale. Each time you press this keyboard shortcut, Windows selects the next keyboard layout as the current layout. It cycles though all of the locales that you installed. To see which locale is currently active, be sure to check the `Enable indicator on taskbar` checkbox. The taskbar will show a two-letter abbreviation for the current locale. Try firing up WordPad and typing. Change locales using the keyboard shortcut and type some more to see how Windows re-interprets your keystrokes according to the new locale. The Sound Recorder will do the same thing.

Delphi's `SysUtils` unit declares the `Languages` object, which stores information about all the different languages and locales that Windows supports. The Sound Recorder needs to work with the locales for which you have installed a keyboard layout, so it iterates through all the languages and tries to load a keyboard layout for each one. When it succeeds, it saves the locale name and the handle for the keyboard layout in a `TComboBox`. When the user chooses a new language from the combobox, the Recorder selects the corresponding keyboard layout. The combobox is preferable to the keyboard shortcut because the Recorder tries to record a sound for every keypress, including the keyboard shortcut for choosing a new language. Listing 1 shows the `GetKeyboardLayout` method and the combobox's `OnChange` event handler.

```
// Load all the keyboard layouts that the user has installed.
// The user can select a new keyboard layout at runtime.
procedure TForm1.GetKeyboardLayouts;
var
  I: Integer;
  Index: Integer;
  Handle: HKL;
begin
  for I := 0 to Languages.Count-1 do begin
    Handle := LoadKeyboardLayout(PChar(IntToHex(Languages.LocaleID[I], 8)),
      Klf_Substitute_OK or Klf_NoTellShell);
    if Handle <> 0 then begin
      Index := KeyboardList.Items.AddObject(Languages.Name[I], TObject(Handle));
      // Pre-select the current keyboard layout.
      if Handle = GetKeyboardLayout(0) then
        KeyboardList.ItemIndex := Index;
    end;
  end;
end;
// When the user selects a new keyboard, tell Windows
// to activate that keyboard layout.
procedure TForm1.KeyboardListChange(Sender: TObject);
var
  Handle: HKL;
begin
  if KeyboardList.ItemIndex >= 0 then begin
    Handle := HKL(KeyboardList.Items.Objects[KeyboardList.ItemIndex]);
    Win32Check(ActivateKeyboardLayout(Handle, 0) <> 0);
  end;
end;
```

➤ *Listing 1*

When the user presses a key to record a sound, the Recorder displays a user-friendly name for the key in the status bar. That task is more complicated now that the character might be from an entirely different character set. If you are using Windows 9x, you must have the correct fonts installed in order to see these characters, and the Recorder must know which font to use. An easier solution is to use Windows NT or 2000, which support the Unicode character set.

Unicode is a 16-bit character set that unifies many different character sets. American, European, Cyrillic, Greek, Arabic, and Asian languages all have representation in the Unicode standard. Mapping a virtual key code to a Unicode character is easy with the `ToUnicode` function (which exists in Windows NT and 2000, but does not exist in Windows 9x). Listing 2 shows the new declaration for

➤ *Listing 2*

```
// Convert a virtual key code to a user-friendly character or key name.
function KeyCodeToDisplay(KeyCode: Word): WideString;
var
  KeyState: TKeyboardState;
  Text: array[0..3] of WideChar;
begin
  GetKeyboardState(KeyState);
  FillChar(Text, SizeOf(Text), 0);
  if ToUnicode(KeyCode, 0, KeyState, Text,
    SizeOf(Text) div SizeOf(WideChar), 0) > 0 then
    Result := Text
  else begin
    Result := KeyCodeToText(KeyCode);
    if (Length(Result) = 6) and (Copy(Result, 1, 4) = 'Char') then
      Result := Result + ' (' + ShortCutToText(KeyCode) + ')';
  end;
end;
```

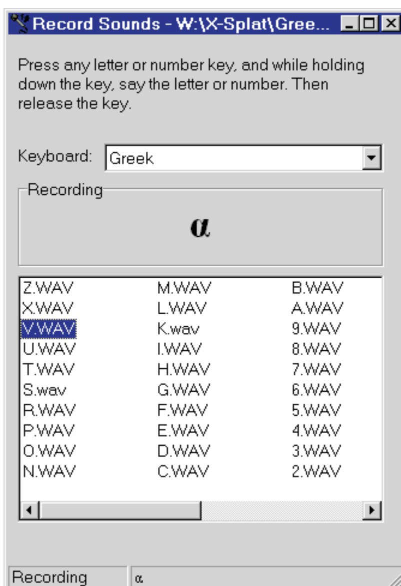
```
type
  // TWideLabel: just like TLabel, but supports wide strings
  TWideLabel = class(TLabel)
  private
    fWideCaption: WideString;
    procedure SetWideCaption(const Value: WideString);
  protected
    procedure DoDrawText(var Rect: TRect; Flags: Longint);
      override;
  published
    property Caption: WideString read fWideCaption
      write SetWideCaption;
  end;
// Copied from TLabel.DoDrawText, but changed to use
// DrawTextW insted of DrawText (which is DrawTextA).
procedure TWideLabel.DoDrawText(
  var Rect: TRect; Flags: Integer);
var
  Text: WideString;
begin
  Text := Caption;
  if (Flags and DT_CALCRECT <> 0) and
    ((Text = '') or ShowAccelChar and (Text[1] = '&') and
    (Text[2] = #0)) then
    Text := Text + ' ';
  if not ShowAccelChar then
    Flags := Flags or DT_NOPREFIX;
  Flags := DrawTextBiDiModeFlags(Flags);
  Canvas.Font := Font;
  if not Enabled then begin
    OffsetRect(Rect, 1, 1);
    Canvas.Font.Color := clBtnHighlight;
    DrawTextW(Canvas.Handle, PWideChar(Text), Length(Text),
      Rect, Flags);
    OffsetRect(Rect, -1, -1);
    Canvas.Font.Color := clBtnShadow;
    DrawTextW(Canvas.Handle, PWideChar(Text), Length(Text),
      Rect, Flags);
  end else
    DrawTextW(Canvas.Handle, PWideChar(Text), Length(Text),
      Rect, Flags);
end;
// When the user changes the text, redraw the control.
procedure TWideLabel.SetWideCaption(
  const Value: WideString);
begin
  fWideCaption := Value;
  AdjustBounds;
  Invalidate;
end;
```

➤ *Listing 3*

the `KeyCodeToDisplay` function. If the virtual key code represents a Unicode character, the function returns that character as a string. Otherwise, it uses the old version of the function to convert the key code to a name, such as `F1` or `PgUp`. Note that the function's return type is now `WideString` instead of plain string.

A `WideString` is similar to an ordinary string (that is, `AnsiString`), except that each character is a `WideChar` instead of `AnsiChar`. (`Char` is currently the same as `AnsiChar`.) A `WideChar` is two bytes long and stores a Unicode character. Delphi can automatically convert between narrow and wide strings, but when converting from wide to narrow strings, you often lose

➤ *Figure 1*



information. If Delphi cannot represent a wide character in the narrow string, it uses a default character (usually `?`). Thus, it is important to keep track of the `WideString` and not let Delphi convert it to a plain string.

To avoid converting the wide string to a narrow string, you must use controls that support wide characters. Windows NT and 2000 (but not Windows 9x) support wide controls, that is, controls that work with Unicode characters and strings. Delphi's VCL doesn't use these controls, though, to maintain compatibility with Windows 9x. Thus, to display Unicode text, you cannot use `TLabel` or `TStaticText`, but must use entirely different functions to create the window, send messages to it, and so on. Instead of `CreateWindow`, you must use `CreateWindowW`, for example. (`CreateWindow` is a synonym for `CreateWindowA`, where the `A` means ANSI. Most API functions come in `A` and `W` flavors, and Delphi maps the plain name to the `A` variety.)

It's easy to make a `TWideLabel` control that inherits from `TLabel`. The only difference is that `TWideLabel` has a new `Caption` property whose type is `WideString`. It overrides the `DoDrawText` method to draw the wide string by calling the Windows API function `DrawTextW` instead of `TLabel`'s use

```
// Set the status information in the right-hand panel of the status bar.
procedure TForm1.SetStatusInfo(const Info: WideString);
begin
  // Do this: "StatusBar.Panels[1].Text := Info;" but using Unicode
  SendMessageW(StatusBar.Handle, Sb_SetTextW, 1, LParam(PWideChar(Info)));
end;
```

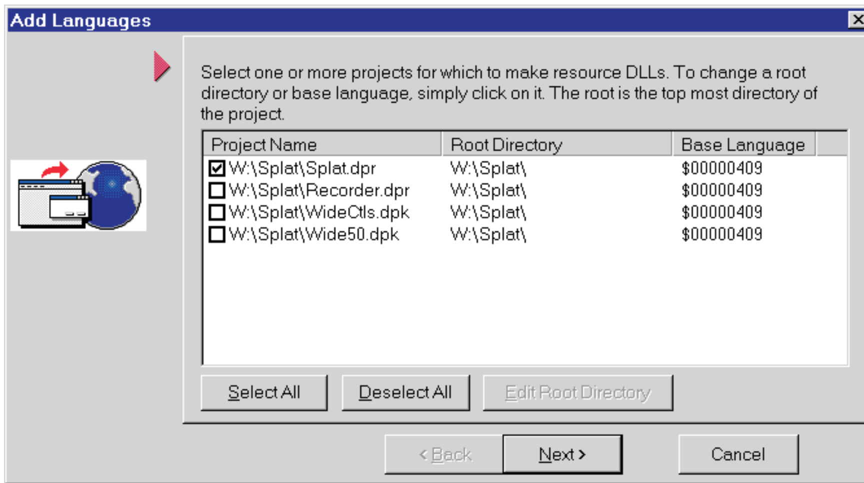
➤ *Listing 4*

of `DrawText` (which is really `DrawTextA`). Listing 3 shows the `TWideLabel` control. Note that Delphi does not store wide strings in a .DFM file, but converts the wide string to an ANSI string. It doesn't matter in this application because the Recorder doesn't need to set the `Caption` at design-time, only at runtime. (To use the `TWideLabel` component, install the `WideCtls` package in Delphi's IDE. The component is added to the Component Palette's `Splat` tab.)

To display the character being recorded, add a group box and a `TWideLabel` in the group box, with the alignment set to `alClient`. When the user presses a key, set the wide label's caption to the key's display text.

For the sake of completeness, the status bar also shows the key. The Windows status bar control can handle Unicode even if it is created as an ANSI control. The only difference is that you must set the status bar panel's caption by calling `SendMessageW`, as shown in Listing 4.

You can now use the updated Recorder to record sounds for other languages. As you will learn in the next section, it is most convenient to store each language's



➤ *Figure 2*

sounds in a separate directory, using a language and country code as the directory name. For example, `RUS` for Russian, `ENU` for English in the United States, `ENG` for English in England, and so on. Figure 1 shows the new Recorder recording a sound for α (*Alpha*), which is the Greek character for the key labelled `A` on American keyboards.

Localizing Splat

To load the sounds for a particular language, you need to create a resource DLL and store the localized sound resources in that DLL. Start by running Delphi's Resource DLL Wizard (which is available in the Professional and Enterprise editions, but not Standard). Choose `File|New...` and invoke the Resource DLL Wizard. Click `Next` to get past the introductory screen and into the Wizard (Figure 2). If you have the Enterprise edition, you can also run the Resource DLL Wizard from the menu bar: choose `Project|Languages|Add...` to run the Wizard, skipping the introductory screen.

The Resource DLL Wizard presents you with a list of the projects in your project group. By default all the projects are checked. Only Splat needs localization, so uncheck the Recorder. If you cannot see the project name, change the column widths until you can read the entire project path. For each language, Delphi will create a subdirectory from the root directory. By default, the root directory is the one that contains the project. The last column shows the current language, which is the default language that Delphi uses when running the program and it cannot find a locale-specific resource to match the user's locale.

If you load the Splat project in Delphi, you must set the root directory to the directory that contains the project. The default is the directory where the project was originally developed, which makes sense on my system, but not on yours.

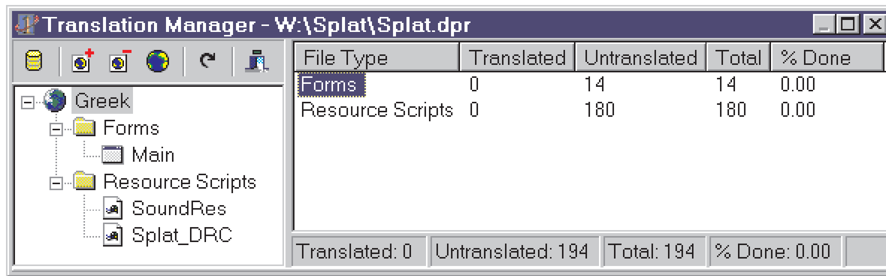
Click the `Next` button to see the list of languages that Windows supports. Scroll down the list and check the languages you want to add. This example uses Greek as an additional language for Splat. Click `Next` to see the languages you chose. Delphi uses the language and country name as the extension for the resource DLL and as the default name for the subdirectory. You can change the directory name, if you wish. Again, if the path is too long, resize the column header until you can read the entire path name. Click `Next` again to see the language status: `New` for a new language or `Update` if the language directory already exists. Click `Next` one more time to see a summary of what Delphi is about to do. Click `Finish` to have Delphi perform these actions, namely creating new projects for the resource DLLs or updating existing projects. Delphi compiles the Splat project to make sure it has all the `resourcestring` declarations and forms updated. Then it copies the necessary files to the new language subdirectory and adds the resource DLL projects to the project group.

Finally, Delphi shows you a list of all the resources that it knows about (Figure 3). You can translate individual captions on forms, translate resourcestrings, and so on. If you know Greek, you can translate *Press ESC to exit the program*. Splat doesn't have any other text that needs to be translated, so the real work lies in making new sound resources.

Delphi creates the ELL subdirectory to contain a copy of the files that need localizing. It copies all the .RC and .DFM files, creates a new .RC file for resourcestrings, and creates a new .DPR for the resource DLL itself. The file that is

Root Directory Settings

Delphi stores the root directory in the project options (.DOF) file. If you move the project or rename the directory, the stored root directory will be wrong. When you open the project, Delphi might notice the error and ask you to enter a new root directory. If Delphi doesn't notice the problem, or if you cancel the dialog box, the next time you use the Resource DLL Wizard, Delphi reports a mysterious *File not found* error, without telling you which file it couldn't find. If you have multiple projects in your project group, you can get the *File not found* error if *any* project has an invalid root directory, even if that project is not checked in the Wizard dialog box. Note that the Resource DLL Wizard does not display the erroneous root directory in its dialog box, but shows the project directory instead. Thus, you cannot see which project is in error. So if you get the *File not found* error, manually set the root directory for every project. If necessary, you can edit the .DOF file manually. Look for the [Resource DLL Wizard] section and make sure the `RootDir` entry is empty (if you don't need any localization) or is set to the directory that contains the project.



➤ *Figure 3*

most important for Splat is SoundRes.rc. As you recall from last month's article, this resource script lists all the sound effects for letter, digits, and other keys. You need to make sure the file uses relative path names and that the ELL directory has all the necessary sound files. Use the new Recorder to record the Greek letters in the ELL\Sounds directory.

You might not want to duplicate all the sounds, though. Some sound effects are not language-specific. To avoid including certain sounds in the Greek resource DLL, you need to modify SoundRes.rc. Don't modify the copy in the ELL directory, Delphi creates that automatically. Modify the master copy in the root directory by separating SoundRes.rc into three parts: alphabet.rc, digits.rc and fx.rc. The first can contain language-specific letters. The second contains language-specific digits. The third contains the sound effects that are not language-specific. Use `#include` directives in SoundRes.rc to include the other files. In the ELL directory, create an empty fx.rc file. Set the alphabet.rc and digits.rc files to include the sound resources that you have localized. Different languages have different alphabets, so each alphabet.rc file will be slightly different. The new SoundRes.rc is now quite simple:

```
#include "alphabet.rc"
#include "digits.rc"
#include "fx.rc"
```

After making any change to the main project, you should re-invoke the Resource DLL Wizard in order to update the resource DLLs. Even though the Wizard is invoked from `File|New...`, you are not creating anything new, just updating the existing project. (In Delphi Enterprise, choose `Project | Language | Update Resource DLLs`.)

If you make a mistake in an .RC file, Delphi doesn't give you a useful error message. It tells you which line is erroneous, but you need to figure out the error yourself. The mistake is usually obvious. For example, if the error line is a `#include` line, the included file probably doesn't exist. Once you work out all the .RC errors, Delphi can build the main project and update the resource DLLs.

Running The Localized Splat

When an application starts, Delphi tries to load a localized resource DLL. A resource DLL sits in the same directory as the application, but its extension is that of a language or language and country (such as .ELL for Greek). Delphi tries three ways to identify the resource DLL extension.

First, Delphi looks under the registry key `HKEY_CURRENT_USER\Software\Borland\Locales\`, using the full path (with expanded long file names) of the application as the entry name and the resource DLL extension as the entry value.

Next, Delphi looks under the key `HKEY_CURRENT_USER\Software\Borland\Delphi\Locales\` using the full path of the application as the entry name. This key is obsolete, and you should always use the plain `Borland\Locales` key for new projects.

Thirdly, if no registry entry specifies the resource DLL extension, Delphi checks the Windows locale. It first looks for a language and country code (such as `FRS` for Swiss French), and if that doesn't work, it tries a plain language code (such as `FR` for French).

At each step, the application tries to load the DLL, and if it succeeds, it skips the remaining steps. If the application loads a resource DLL, it loads forms and `resourcestring` resources from the resource DLL. You can load other resources (such as the sound resources) from this DLL by calling `FindResourceHInstance` to obtain the instance handle of the resource DLL. Pass that instance handle to any function that loads a resource (such as `PlaySound`).

Not all the sounds are localized, so some sounds are loaded from the resource DLL and some from the application. Because all the calls to `PlaySound` are handled in the `PlayWave` and `PlayRandomWave` functions, it is easy to modify them. They first try to load the sound from the resource DLL, and if that fails, the function tries to load the sound from the application. As before, if `PlaySound` fails even to play a default sound, Splat plays the system beep. Listing 5 shows the new `PlayWave` function.

To test the new Greek sounds, run Regedit and create the appropriate registry key for your Splat.exe file. Set the registry value to ELL. Run Splat and press letters to hear Greek. (My apologies to all Greek readers. The sound files that accompany this article are based on how I learned Greek letters in math and science classes in the

➤ *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, FindResourceHInstance(hInstance),
    Snd_Resource or Snd_Async or Snd_NoDefault) then
    if not PlaySound(ResName, hInstance, Snd_Resource or Snd_ASync) then
      Beep;
end;
```

USA, and do not necessarily have any relationship to correct Greek pronunciation.)

Adding More Shapes

Now that you have lots of sounds in multiple languages, you might want to add more shapes. The Splat framework makes it easy to add new shape classes. The program's name comes from a shape that looks like a paint splat. The shape is drawn by a Windows metafile. A Windows metafile is a binary file that contains graphical commands. The commands correspond to the graphical function calls that you can use to draw on a Windows device context. From one point of view, a metafile is simply a persistent form of drawing on a canvas. The `TMeta- fileShape` class in Splat draws the metafile in the `Draw` method. (Windows has old metafiles and newer metafiles, called enhanced metafiles. Splat supports only enhanced metafiles, which are device-independent. The basic principles apply to both kinds.)

One of the key differences between a metafile and the other shapes is that Splat draws a metafile with its original aspect ratio. Windows lets you draw a metafile with any aspect ratio, but Splat preserves the aspect ratio to avoid distorting the shape. `TMetafileShape` overrides `ChangeSize` so the *x* and *y* sizes grow at the same rate, instead of the default behavior which lets them grow at different rates.

The other interesting twist that metafiles introduce is that a metafile's colors are static, as preserved in the file. Having shapes change color is part of the fun of Splat, though, so Splat plays a trick with the metafile. `TMetafileShape` changes the metafile's use of black to the shape's random color. Black would not ordinarily show up against the black background anyway, so it seems like a safe choice for mapping to the shape's color. If you have a shape that needs to be partially black, you can use a color that is almost black, such as `$010101`. The user will see the color as black, but Splat will not and so will not try to adjust the color.

In order to implement the color trick, you need to understand the structure of a Windows metafile. A metafile is a stream of records. The first record is a header that contains information about the metafile. After the header record, each metafile record represents a Windows API function for displaying graphical objects. Drawing a metafile is called 'playing' the metafile, which means interpreting each record in succession and calling the API function that the record represents. Windows has API functions for playing back an entire metafile at once, or for examining and playing a metafile one record at a time. Splat uses the latter function so it can adjust the colors when needed.

The metafile records that must be changed are those that deal with colors. A quick look through the Windows API documentation reveals the following functions: `SetTextColor`, `SetBkColor`, `CreatePen`, `CreatePenIndirect`, and `CreateBrush`. `CreatePen` and `CreatePenIndirect` are just variations of the same function, so they have a single metafile record to represent both functions. Thus, Splat needs to alter four different kinds of metafile records. For each record, it must check the color, and if the color is black, change the color to the shape's color. To examine every record in a metafile, call `EnumEnhMetafile`. It takes a callback function as an argument, and Windows calls the callback, passing each metafile record to the function. The metafile record begins with a size and a record type. The callback function uses the record type to decide what to do with the record.

➤ *Figure 4*

Splat could modify the record in one of two ways. First, Splat could make a temporary copy of the record, change the color in the copy, and play the copy. Second, Splat could change the color in the original record and restore the original color after playing the record.

The advantage of the first choice is that you don't risk modifying the existing metafile. On the other hand, the record size is variable, and you would need to use dynamic memory allocation for the copy, so the second method is faster and simpler. The color change affects only the in-memory copy of the metafile and has no effect on the actual file on disk.

A `case` statement in the callback function checks for the four color-related records and finds the color field in each record. Other records are left alone. Palette colors, for example, are left untouched for the sake of simplicity.

To make it dead simple to create metafile shapes, the `TMetafileShape` class automatically loads a metafile resource whose name is the same as the class name (minus the leading `T`). Thus, you can trivially add new metafile shapes to Splat by adding the metafile resource and declaring a one-line class that inherits from `TMetafileShape`.

The Splat program gets its name from the metafile shape `TSplat`, which is a fun shape that looks like a paint splat. Its class declaration is trivial:

```
// automatically loads the
// "Splat" metafile resource
type TSplat =
  class(TMetafileShape);
```

The `TMetafileShape` class loads the resource, but before doing so, the shape object must be fully initialized and ready to go. `TMetafileShape` cannot guarantee anything about classes that inherit from it, so the safest way to load the resource is to wait until after the constructors have finished their work. The `AfterConstruction` method serves this purpose neatly and effectively. Delphi automatically calls the virtual method `AfterConstruction` after the constructor has returned. `TMetafileShape` overrides `AfterConstruction` to load the resource. A derived class can take care of any and all initialization in its constructor without any concerns about when to call the inherited constructor. Listing 6 shows the `TMetafileShape`

➤ *Listing 6*

```
type
  // Metafile shape. The shape is stored as a metafile
  // resource. The initial size is small and increases with
  // each generation. When the metafile is played, the basic
  // colors are changed the shape's main color.
  //
  // Derived classes must call LoadMetafileResource or
  // otherwise create the resource in the Metafile property.
  //
  // TMetafileShape is an abstract class (although it lacks
  // abstract methods). Concrete classes derive from
  // TMetafileShape and provide an actual metafile resource.
  // By default, the class name is the resource name
  // (after removing the leading T).
  TMetafileShape = class(TShape)
  private
    fMetafile: TMetafile;
    fBounds: TRect;
  protected
    procedure LoadMetafileResource(
     const ResID, ResType: PChar);
    function ResourceType: PChar; virtual;
    function ResourceName: string; virtual;
  public
    constructor Create(Position: TPoint); override;
    destructor Destroy; override;
    procedure AfterConstruction; override;
    procedure Draw(Canvas: TCanvas); override;
    procedure ChangeSize; override;
    property Metafile: TMetafile read fMetafile;
    property Bounds: TRect read fBounds;
    property Left: Integer read fBounds.Left
      write fBounds.Left;
    property Right: Integer read fBounds.Right
      write fBounds.Right;
    property Top: Integer read fBounds.Top
      write fBounds.Top;
    property Bottom: Integer read fBounds.Bottom
      write fBounds.Bottom;
  end;
procedure TMetafileShape.AfterConstruction;
begin
  inherited;
  LoadMetafileResource(PChar(ResourceName), ResourceType);
end;
constructor TMetafileShape.Create(Position: TPoint);
begin
  inherited;
  fMetafile := TMetafile.Create;
end;
destructor TMetafileShape.Destroy;
begin
  FreeAndNil(fMetafile);
  inherited;
end;
// Playback a single metafile record, changing the
// background color to the shape's own color. Do not change
// the actual metafile record.
function EnumFunc(DC: HDC; Table: PHandleTable; Emfr:
  PEnhMetaRecord; NumObjects: DWord; Self: TMetafileShape):
  LongBool; stdcall;
var ColorPtr: ^COLORREF;
begin
  ColorPtr := nil;
  case Emfr.iType of
  Emr_SetTextColor:
    with PEmrSetTextColor(Emfr)^ do
      if crColor = BackgroundColor then
        ColorPtr := @crColor;
  Emr_SetBkColor:
    with PEmrSetBkColor(Emfr)^ do
      if crColor = BackgroundColor then
        ColorPtr := @crColor;
  Emr_CreateBrushIndirect:
    with PEmrCreateBrushIndirect(Emfr)^ do
      if lb.lbColor = BackgroundColor then
        ColorPtr := @lb.lbColor;
  Emr_CreatePen:
    with PEmrCreatePen(Emfr)^ do
      if lopn.lopnColor = BackgroundColor then
        ColorPtr := @lopn.lopnColor;
  else
    ; // Otherwise, leave the record alone.
  end;
  // Set the metafile color to the shape's color.
  if ColorPtr <> nil then
    ColorPtr^ := Self.Color;
  Win32Check(PlayEnhMetaFileRecord(DC, Table^, Emfr^,
    NumObjects));
  // Restore the record's original color.
  if ColorPtr <> nil then
    ColorPtr^ := BackgroundColor;
  Result := True;
end;
// Draw a metafile by enumerating the metafile records.
procedure TMetafileShape.Draw(Canvas: TCanvas);
var
  OldPalette, NewPalette: HPalette;
  Rect: TRect;
begin
  BoundingBox(Rect);
  // Metafile bounds include right and bottom do decrement
  Dec(Rect.Right);
  // the TRect bounds, which ordinarily do not include them.
  Dec(Rect.Bottom);
  OldPalette := 0;
  NewPalette := Metafile.Palette;
  if NewPalette <> 0 then begin
    OldPalette :=
      SelectPalette(Canvas.Handle, NewPalette, True);
    RealizePalette(Canvas.Handle);
  end;
  Win32Check(EnumEnhMetaFile(Canvas.Handle, Metafile.Handle,
    @EnumFunc, Self, Rect));
  if NewPalette <> 0 then
    SelectPalette(Canvas.Handle, OldPalette, True);
end;
// Load a metafile resource. The resource might be in the
// resource DLL or the main application.
procedure TMetafileShape.LoadMetafileResource(const ResID,
  ResType: PChar);
var
  ResInstance: THandle;
  Stream: TResourceStream;
begin
  ResInstance := FindResourceHInstance(hInstance);
  if FindResource(ResInstance, ResID, ResType) = 0 then
    ResInstance := hInstance;
  Stream := TResourceStream.CreateFromID(ResInstance,
    Integer(ResID), ResType);
  try
    Metafile.LoadFromStream(Stream);
  finally
    Stream.Free;
  end;
  // The initial size is square--keep the original aspect
  // ratio by shrinking the smaller dimension to match the
  // metafile.
  if Metafile.Width > Metafile.Height then
    YSize := MulDiv(XSize, Metafile.Height, Metafile.Width)
  else
    XSize := MulDiv(YSize, Metafile.Width, Metafile.Height);
end;
// Compute the next size, trying to maintain the metafile's
// aspect ratio.
procedure TMetafileShape.ChangeSize;
var
  Delta: Integer;
begin
  Delta := Random(DeltaDimension);
  if Metafile.Width > Metafile.Height then begin
    XSize := XSize + Delta;
    YSize := YSize +
      MulDiv(Delta, Metafile.Height, Metafile.Width);
  end else begin
    XSize := XSize +
      MulDiv(Delta, Metafile.Width, Metafile.Height);
    YSize := YSize + Delta;
  end;
end;
// Default resource name is the same as the class name,
// minus the leading 'T'.
function TMetafileShape.ResourceName: string;
begin
  Result := Copy(ClassName, 2, MaxInt);
end;
// Default resource type is 'Metafile'. The resource type
// is not case sensitive.
function TMetafileShape.ResourceType: PChar;
begin
  Result := 'Metafile';
end;
```

class, and Figure 4 shows a Splat game in progress with several splat shapes evident.

Compressing The Sound Resources

Splat is a fat program. The more sound effects you add, the larger the Splat.exe file grows. You can shrink the file size by about 30% by compressing the .WAV resources. Delphi comes with compression software in the `ZLIB` unit. The source is in the Info\Extras directory on the Delphi CD-ROM. The simplest way to compress data is to create a `TCompressionStream`, which reads from another stream and compresses data on the fly. `TDecompressStream` expands data that it reads from a stream.

The first step is to create compressed .WAV files by modifying the Recorder. ZLIB-compressed .WAV files are not a standard format, so there is no convention for the file name extension. The Recorder uses .ZWAV, which requires some minor bookkeeping changes, such as listing `*.ZWAV` files in the list view.

The major change is to save a .ZWAV file. The simplest way is to ask Windows to save a .WAV file, and then compress the file. This approach requires the fewest changes to the Recorder. The `FormKeyUp` method still saves the .WAV file, but after saving the file, `FormKeyUp` calls the `Compress` procedure to compress the file to a .ZWAV file, and then it deletes the .WAV file. The `Compress` procedure uses `TCompressStream` and `TFileStream` to read one file, compress it, and write to another file. Listing 7 shows the `Compress` procedures that compress files and streams.

```
// Compress the WAVE data from InStream onto OutStream.
procedure Compress(InStream, OutStream: TStream);
var
  InBuffer, OutBuffer: Pointer;
  OutSize: LongInt;
begin
  InBuffer := nil;
  OutBuffer := nil;
  try
    GetMem(InBuffer, InStream.Size);
    InStream.ReadBuffer(InBuffer^, InStream.Size);
    CompressBuf(InBuffer, InStream.Size, OutBuffer, OutSize);
    OutStream.WriteBuffer(OutSize, SizeOf(OutSize));
    OutStream.WriteBuffer(OutBuffer^, OutSize);
  finally
    FreeMem(InBuffer);
    FreeMem(OutBuffer);
  end;
end;
// Compress the WAVE data from InFile to OutFile.
procedure Compress(const InFile, OutFile: string);
var
  InStream, OutStream: TFileStream;
begin
  InStream := nil;
  OutStream := nil;
  try
    InStream := TFileStream.Create(InFile, fmOpenRead or fmShareDenyWrite);
    OutStream := TFileStream.Create(OutFile, fmCreate);
    Compress(InStream, OutStream);
  finally
    InStream.Free;
    OutStream.Free;
  end;
end;
```

➤ *Listing 7*

That's it. The Recorder is now ready to use for recording compressed .ZWAV files. An exercise for the reader is to write a program that compresses existing .WAV files to .ZWAV files (by calling `Compress`, of course).

The next step is to modify Splat. Edit the alphabet.rc, digits.rc, and

```
// Load the sound data and decompress it into OutBuf.
// Return True for success, False if the sound resource or
// file could not be loaded.
function LoadSound(pszSound: PChar; hmod: HINST; fdwSound:
  Cardinal; var OutBuf: Pointer): Boolean;
var
  InBuf: PZWaveData;
  FreeInBuf: Boolean;
  OutSize: Integer;
begin
  InBuf := nil;
  OutBuf := nil;
  FreeInBuf := False;
  try
    if (fdwSound and Snd_FileName) = Snd_FileName then begin
      // pszSound is a file name. PlayCompressedSound must
      // free InBuf.
      if not GetFileContents(pszSound, InBuf) then begin
        Result := False;
        Exit;
      end;
      FreeInBuf := True;
    end else if (fdwSound and Snd_Resource) = Snd_Resource
      then begin
      // pszSound is a resource name. Windows frees InBuf,so
      // PlayCompressedSound must not free it.
      if not GetResourceContents(hmod, pszSound, InBuf)
        then begin
        Result := False;
        Exit;
      end;
    end else if (fdwSound and Snd_Memory) = Snd_Memory
      then begin
      // pszSound points to the sound data in memory.
      // PlayCompressedSound must not free the memory--that
      // is the caller's responsibility.
      InBuf := PZWaveData(pszSound);
    end else begin
      // Must be a registry alias (Snd_Alias), or something
      // else, such as Snd_Purge. Let PlaySound handle this
      // case. In particular, aliases cannot be compressed
      // because they are used by other programs that don't
      // know about ZWAVEs.
      Result := PlaySound(pszSound, hmod, fdwSound);
      Exit;
    end;
    // Decompress the data. The estimated size is twice the
    // input size. Most ZWAVE files are about 50-60% of the
    // original size.
    DecompressBuf(@InBuf.Data, InBuf.Size, 2*InBuf.Size,
      OutBuf, OutSize);
    // Remember this sound.
    Cache.Add(pszSound, hmod, fdwSound, OutBuf);
  finally
    if FreeInBuf then
      FreeMem(InBuf);
  end;
  Result := True;
end;
// Look up a compressed sound in the cache. If it isn't
// present, load and decompress the sound data. Then play
// the decompressed sound.
function PlayCompressedSound(pszSound: PChar; hmod: HINST;
  fdwSound: Cardinal): LongBool;
var
  Buffer: Pointer;
begin
  Buffer := Cache.Lookup(pszSound, hmod, fdwSound);
  if Buffer = nil then begin
    Result := LoadSound(pszSound, hmod, fdwSound, Buffer);
    if Buffer = nil then
      Exit;
  end;
  // Play the sound from memory.
  fdwSound := (fdwSound and not (Snd_Resource or
    Snd_FileName)) or Snd_Memory;
  Result := PlaySound(Buffer, 0, fdwSound);
end;
```

➤ *Listing 8*

```
type
  // A ZWAVE file or resource has the following format: the
  // first four bytes contain the size of the compressed
  // data, which follow immediately after the size. Resource
  // should always contain an explicit size field because
  // Windows pads resource data to fit on longword
  // boundaries.
  PZWaveData = ^TZWaveData;
  TZWaveData = packed record
    Size: 0..MaxInt;
    Data: TByteArray;
  end;
  PZWaveCacheNode = ^TZWaveCacheNode;
  TZWaveCacheNode = record
    // Save the arguments to PlayCompressedSound to look for
    // a match.
    pszSound: Pointer;
    strSound: string;
    hmod: HINST;
    fdwSound: DWORD;
    Data: PZWaveData;
    Next: PZWaveCacheNode;
    Prev: PZWaveCacheNode;
  end;
  // Cache the most recently used ZWAVE data, to avoid
  // repeatedly uncompressing the same ZWAVE file or
  // resource. Zero means no caching (except that Snd_Async
  // requires a cache size of at least 1). The cache is
  // searched linearly, so don't use a large cache size.
  // To avoid problems, the maximum size is set arbitrarily
  // to 100.
  TZWaveCacheSize = 0..100;
  TZWaveCache = class
  private
    fHead, fTail: PZWaveCacheNode;
    fCount: TZWaveCacheSize;
    fCapacity: TZWaveCacheSize;
    procedure SetCapacity(NewCapacity: TZWaveCacheSize);
  protected
    function Invariant: Boolean;
    procedure Add(pszSound: PChar; hmod: HINST; fdwSound:
      DWORD; Buffer: PZWaveData);
    procedure FreeNode(Node: PZWaveCacheNode);
    function Lookup(pszSound: PChar; hmod: HINST; fdwSound:
      DWORD): PZWaveData;
    property Head: PZWaveCacheNode read fHead;
    property Tail: PZWaveCacheNode read fTail;
  public
    constructor Create;
    destructor Destroy; override;
    property Count: TZWaveCacheSize read fCount;
    property Capacity: TZWaveCacheSize read fCapacity
      write SetCapacity default 1;
  end;
```

➤ *Listing 9*

fx.rc files to use compressed .ZWAV files: change the resource type from `WAVE` to `ZWAVE`, and change the file extensions from .WAV to .ZWAV.

The final step is to modify Splat to play back the compressed sound resources. Change the references to the `WAVE` resource type to `ZWAVE`. Change the `PlaySound` calls to `PlayCompressedSound`. The `PlayCompressedSound` function takes the same arguments as `PlaySound`, but it plays a `ZWAVE` compressed sound from a file or resource. It does so by loading the compressed sound, decompressing it into memory, and calling `PlaySound` with the `Snd_Memory flag`. Other flags (such as `Snd_Async`) are passed to `PlaySound`.

On an older, slower, system, all this decompressing can be time consuming. Imagine a child leaning on one key (my son likes the space bar), repeatedly playing that sound. It is wasteful to decompress the sound every time the user presses the key. Instead, `PlayCompressedSound` keeps a cache of the most recently decompressed sounds. Listing 8 shows how `PlayCompressedSound` works.

The cache is implemented by the `TZWaveCache` class. It keeps a linked list of `TZWaveCacheNode` records, in most-recently-used order. Each record stores a copy of the arguments that were passed to `PlayCompressedSound`. Each node also keeps a copy of the

```
// Return True if Node matches the arguments pszSound, hmod, and fdwSound
// which are from a call to PlayCompressedSound. Return False if the
// arguments differ at all.
function SameSound(Node: PZWaveCacheNode; pszSound: PChar; hmod: HINST; fdwSound:
  DWORD): Boolean;
begin
  if Node.fdwSound <> fdwSound then
    // Flags must match exactly.
    Result := False
  else if Node.hmod <> hmod then
    // Module handle must match exactly. If the handle is not used,
    // the caller must use 0.
    Result := False
  else if (Node.pszSound = nil) and (Node.strSound <> '') then
    // If the stored sound has a string name (file or resource name),
    // compare the strings.
    Result := SameText(Node.strSound, pszSound)
  else
    // Otherwise, the sound pointer is a resource ID or memory pointer,
    // both of which can be compared verbatim.
    Result := pszSound = Node.pszSound
end;
```

➤ *Above: Listing 10*   ➤ *Below: Listing 11*

```
procedure TMainForm.WMShowWindow(var Message: TWMShowWindow);
begin
  // The user cannot minimize Splat, but pressing Windows+M
  // minimizes all windows. Prevent Splat from minimizing
  // by intercepting the Wm_ShowWindow message.
  if not Message.Show and (Message.Status = Sw_ParentClosing) then
    Message.Result := 0
  else
    inherited;
end;
```

decompressed sound. When `PlayCompressedSound` is called, it checks the cache for identical arguments, and if it finds a match, it moves that node to the start of the list, and plays the sound. Repeatedly pressing the same key results in a fast response. Because the cache uses a linked list, its size should be small. If you need to cache many sounds, you should change the linked list to a hash table and devise a suitable hash function. Listing 9 shows the declarations of `TZWaveCache` and its associated types. Most of the code that implements the cache is for managing the linked list; consult the complete code on the disk for details.

To give you an idea of how the cache works, Listing 10 shows the `SameSound` function, which determines whether a call to `PlayCompressedSound` is a cache hit or miss.

Splat And Polish

Splat still has some rough edges. For example, press the `Windows+M` key combination to minimize all applications, and Splat minimizes, too. Disabling the Minimize icon doesn't help in this case. Instead, Splat must intervene when it receives a `Wm_ShowWindow` message that says its parent window is closing. Listing 11 shows the `WmShowWindow` message handler.

```
// Return True if the key with virtual key code KeyCode is in the toggled (down)
// state. The caller supplies the keyboard state so IsKeyToggled doesn't have to
// call GetKeyboardState repeatedly.
function IsKeyToggled(const KeyState: TKeyboardState; KeyCode: Word): Boolean;
begin
  Result := (KeyState[KeyCode] and 1) <> 0;
end;
procedure TMainForm.FormCreate(Sender: TObject);
var
  KeyState: TKeyboardState;
begin
  ...
  // Get the keyboard state to determine the status of Caps Lock, Num Lock, and
  // Scroll Lock.
  Win32Check(GetKeyboardState(KeyState));
  CapsLock   := IsKeyToggled(KeyState, Vk_Capital);
  NumLock    := IsKeyToggled(KeyState, Vk_NumLock);
  ScrollLock := IsKeyToggled(KeyState, Vk_Scroll);
  ...
```

➤ Above: Listing 12

➤ Below: Listing 13

```
// Simulate a press of the key with the given virtual key code and scan code.
procedure SetKeyState(KeyCode, ScanCode: Word);
begin
  keybd_event(KeyCode, ScanCode, KeyEventF_ExtendedKey, 0);
  keybd_event(KeyCode, ScanCode, KeyEventF_ExtendedKey or KeyEventF_KeyUp, 0);
end;
procedure TMainForm.FormDestroy(Sender: TObject);
const
  CapsLock_ScanCode = $3A;
  NumLock_ScanCode = $45;
  ScrollLock_ScanCode = $46;
var
  KeyState: TKeyboardState;
begin
  // Restore the Caps Lock, Num Lock, and Scroll Lock keys.
  Win32Check(GetKeyboardState(KeyState));
  if IsKeyToggled(KeyState, Vk_Capital) <> CapsLock then
    SetKeyState(Vk_Capital, CapsLock_ScanCode);
  if IsKeyToggled(KeyState, Vk_NumLock) <> NumLock then
    SetKeyState(Vk_NumLock, NumLock_ScanCode);
  if IsKeyToggled(KeyState, Vk_Scroll) <> ScrollLock then
    SetKeyState(Vk_Scroll,  ScrollLock_ScanCode);
  ...
```

Another problem occurs when the user presses the `Caps Lock`, `Num Lock`, or `Scroll Lock` keys. When Splat exits, the keyboard state has been changed. The intended user is a child who doesn't know one key from another, and it is likely that these keys will be pressed. The polite thing to do is for Splat to restore the original settings of the keyboard state. Unfortunately, Windows does not have an API function to set the keyboard state at a global level, only for a single process. The solution is to pretend the user pressed the `Caps Lock` key etc. By simulating a keypress, you can change the global keyboard state. Call `keybd_event` to simulate pressing a key. (Note that Windows 95 does not let `keybd_event` change the `Num Lock` key, only `Caps Lock` and `Scroll Lock`.)

The `FormCreate` method saves the state of the toggle keys in `Boolean` fields: `CapsLock`, `NumLock`, and `ScrollLock`. It does so by calling `IsKeyToggled` for `Vk_CapsLock`, etc, as shown in Listing 12. The `FormDestroy` method restores the state of each key. If the original state is different from the current state, Splat calls `keydb_event` to simulate a keypress. Listing 13 shows `SetKeyState` and how it is called. Windows does not declare any constants for the standard scan codes, so you must declare them explicitly. (The simplest way to learn a key's scan code is to write a small program that displays the scan code of any key you press.)

Put all the pieces together and rebuild Splat. You now have a game the whole family can play. Splat and enjoy!

Ray Lischner is the author of *Delphi in a Nutshell* and other books and articles about Delphi. He also teaches computer science at Oregon State University. You can reach Ray at delphi@tempest-sw.com