Volume 9, Number 6—June 2005

www.bcbjournal.com
ISSN 1093-2097

EnCoded Communications Group

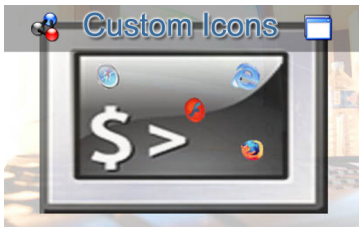
C++BUILDER Developer's Journal

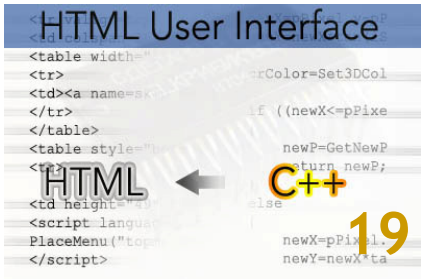
A Monthly Publication Offering Tips & Techniques for Borland C++Builder

Special Issue on User-Interface Development

Edited by D. M. Chandler and M. J. Smith

In this Issue:

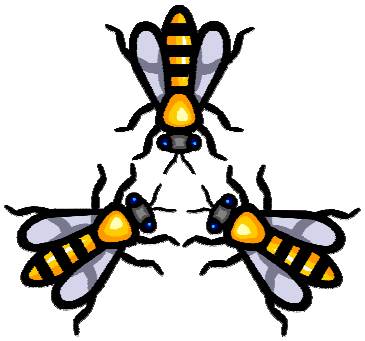




23

This Month's Special Issue

By Damon Chandler



As C++Builder developers, an application's ease-of-use is not only something that we can appreciate—it's something that we have come to *expect* and *rely upon*. In one way to another, I'm sure we all feel that C++Builder—despite its long-standing bugs—is a vital tool for rapid application development. But, what it is about this tool that makes it so useful? The compiler is certainly nothing to write home about; and you'd be hard-pressed to find someone singing the praises of the IDE's stability; so, what exactly is it that we find so attractive about BCB? (See this month's poll question.)

For me, the answer is simple: I find C++Builder's visual designer and component-based framework extremely accessible and intuitive to use. I have come to rely upon this ease-of-use, and I strive to pass it on in my own applications.

This month's special issue contains an interesting mix of articles which cover several unique aspects of application development, but all with one underlying theme in mind: *facilitating an end-user's ability to access and interact with an application*.

This month's special issue contains articles with one underlying theme... facilitating an end-user's ability to interact with an application.

Curtis Krauskopf opens the issue by presenting a tutorial on how to customize the icons of console-mode applications, a task that cannot be accomplished directly via the IDE's Project Options dialog. If you've never before created your own icons, this is an opportunity to learn the basics of Image Editor, and learn how to add a touch of customization to your console applications.

Next, **Don Doerres** discusses how to access command-line parameters in GUI applications. In Windows, there are various methods of passing parameters to an application (e.g., via a shortcut's properties, in addition to using the command line). Often, it's much more convenient to specify this auxiliary information as startup parameters than it is to use the application's various edit controls or dialog boxes.

Contributing Editor **Malcolm Smith** then presents a class—`TConsoleRedirect`—which provides a crucial link between console-mode and GUI applications. Malcolm discusses how to use the `CreateProcess()` function and Windows pipes to launch a console application and redirect its output to a standard edit or memo control.

Mark Finkle demonstrates how to use the `TCppWebBrowser` control to harness the power of HTML in GUI applications. Mark shows how to create GUI elements and controls using HTML, and how to use Dynamic HTML and JavaScript from within a BCB application.

Next, **Malcolm Smith** describes how to get the most out of your screen's real estate by creating a secondary desktop. Malcolm presents a novel DLL-based approach which allows selective forms of a *single* BCB application to span multiple desktops.

Finally, I close the issue by demonstrating how to record, recognize, and respond to mouse gestures, a feature which provides a quick and intuitive means of invoking common tasks in GUI applications.

On behalf of our editorial staff, I sincerely hope you find this issue both informative and enjoyable. As always, your feedback is appreciated. Enjoy!

Special Issue Editors
Damon M. Chandler and Malcolm J. Smith

Custom Icons for Console-Mode Applications

By Curtis Krauskopf

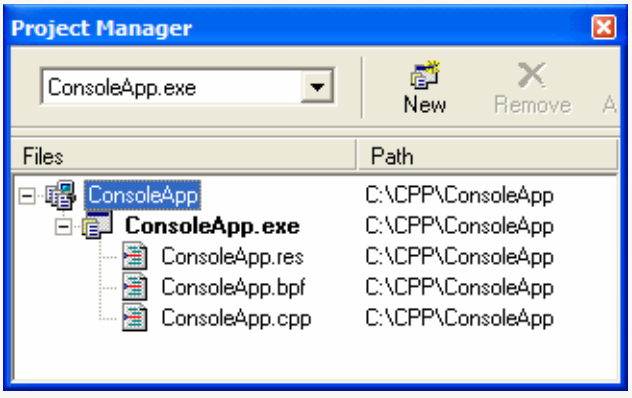
C++Builder provides an icon for console-mode applications. It's a nice icon (see **Figure A**), but at first glance it looks too much like the C++Builder icon. I can't count the number of times I've accidentally clicked on the wrong icon in the Taskbar. Another problem is that if you have two or more console-mode applications running at the same time, they all use the same default icon.

This article provides a tutorial on customizing the icons of console-mode applications. In addition, I talk about creating transparent pixels in a customized icon and creating 16×16-pixel icons using the Borland Image Editor.

Figure A

The default console-mode icon.

Figure B



A typical console-mode application contains a resource file (.res), a Borland Project File (.bpf) and, of course, the C++ source file.

What to do

If you've done any significant Windows or VCL programming using C++Builder, you probably know about the "Load Icon" button on the Applications panel on the project options screen. Unfortunately, for console-mode applications, that button is grayed out.

To change the icon for a console-mode application, first load its resource file into Image Editor, make the changes and then save the modified resource file. The new icon will be incorporated into the executable after the resource file has been re-added to the project or the Borland C++Builder IDE has been restarted.

Where to start

Let's say your console-mode application is called ConsoleApp. The first step is to open the Project Manager view as shown in **Figure B**.

By default, the resource file is located in the same directory as your console-mode application. For the ConsoleApp project in **Figure B**, the resource file you need to open is called CONSOLEAPP.RES. Open the resource file using Borland's Image Editor.

3-2-1… Launch the Image Editor

There are two places where you can find Image Editor:

1. In the Borland C++Builder 6 program menu (via "Start | Programs | Borland C++Builder 6 | Image Editor").

2. In the C++Builder IDE; choose the "Tools" menu and then choose "Image Editor".

Open the resource file

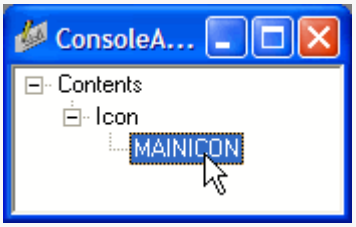
In Image Editor, select "File | Open" and then navigate to the .RES file for your console-mode application.

Open the icon

In a default console-mode application's resource file, the icon in the resource file is called MAINICON. To open the MAINICON icon, expand the Icon tree branch and double-click on MAINICON (see **Figure C**).

The double-click will open the icon editing window (**Figure D**) in Image Editor. The icon editing window has two parts: a pixel editor in the left-hand panel and a full-size icon in the right-hand panel. The default console-mode icon is a 32×32 pixel, 16-color icon.

Figure C



A default console-mode application resource file contains an icon called MAINICON.

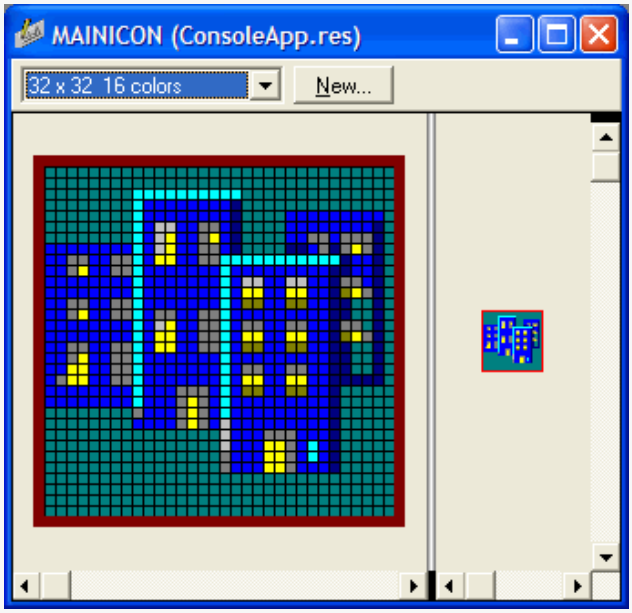
Opening your inner artist

The vertical toolbar on the left side of Image Editor's main window contains all of your graphics-editing tools. Even though Image Editor doesn't have all of the features of a high-priced graphics-editing application, you can still make some pretty amazing icons easily and quickly.

Icon basics

Every pixel in an icon can either be colored or it can be transparent. Image Editor displays transparent pixels using a dark teal (blue-green) color. In the default Borland-supplied icon, the pixels surrounding the buildings are all transparent pixels.

Figure D



The icon editing window contains a pixel editor on the left side and a full-size icon on the right side. Transparent pixels (the blue-green pixels) surround the buildings.

Bulk erase

If you're like me and you like starting with a blank slate, the first thing you'll want to do is delete all of the colored pixels. As a novice, I clicked on the eraser tool and started erasing colored pixels. This was painfully slow because it changed only one pixel at a time.

Next, I tried the Eye Dropper tool and clicked on one of the transparent pixels. I then used the Fill tool (it looks like a pouring paint bucket) to erase large sections of the same color. This was better than using the Eraser tool but it still left lots of scattered pixels of different colors.

Finally, I tried the Brush tool. That was what I needed. Because I had previously picked a transparent pixel with the Eye Dropper tool, I was able to use the Brush tool to quickly erase large sections of the palette.

Later, I discovered I could left-click on the red "S" near the color palette (**Figure E**). This is the same as picking a transparent pixel using the Eye Dropper tool

and it's handy if the icon doesn't have any transparent pixels.

Saving

When you're done customizing your icon, you'll notice on the "File" menu that the "Save" and "Save As" menu choices are grayed-out. To save the icon, you first need to click on the resource window (**Figure C**); you'll then be able to save the icon (and the resource file).

Seeing your icon

If you go back to C++Builder and then build and execute your application, you'll be disappointed to see the old icon. Even if you force the application to build, you'll still see the old icon! The linker apparently caches the .RES file. There are two ways to force a changed resource file to be incorporated into an executable:

1. Remove the resource file from the project and then add it back again.
2. Close the Borland IDE and launch it again.

Creating 16x16-pixel icons

The Windows Taskbar and the application's title bar typically use 16×16-pixel icons. If the only icon available is a 32×32-pixel icon, Windows will shrink its size to 16×16 pixels. Sometimes the 32×32-pixel icon doesn't render well when it's shrunk to 16×16. Likewise, if the resource file defines only a 16x16-pixel icon, it might have a "jagged" (aliased) look when it's rendered as a 32×32-pixel icon. It's therefore a good idea to include both 16×16-pixel and 32×32-pixel icons in your resource file.

On the icon editing window in Image Editor (shown in **Figure D**), click on the "New" button to add a 16×16-pixel icon to the resource. The Icon Properties window will appear.

Choose the 16×16 (small icon) size and then click "OK." A new, empty 16×16-pixel icon is displayed in the icon editing window. You can easily switch back and forth between the icons by clicking on the combo box in the icon editing window (see **Figure D**).

Installing a resource file (.RES) that contains a 16×16-pixel icon is the same as installing it with a 32×32-pixel icon:

Figure E

Click on the red swooping "S" in the color palette to create transparent pixels in the pixel editor.

1. Save the resource file.
2. Remove the resource file from the project.
3. Add it back into the project.
4. Build or make the application.

Making it easier

After you've erased all of the pixels from an icon, save the resource file in an easy-to-find location (I chose the ICONS/ directory in the C++Builder root directory).

The next time you need to customize an icon, launch Image Editor and open the empty resource file that you previously saved. You can safely overwrite the .RES file in your project but you'll still need to use the remove/add trick to make the linker realize that the .RES file has been changed.

Conclusion

Custom icons help to differentiate your console-mode applications at run-time and when they are picked from Windows Explorer. Creating a custom icon is easy. When you're ready to distribute your executable, the Borland C++ compiler incorporates the icon resource into the executable so you don't need to provide any other files.

Contact Curtis at curtis@decompile.com.

Putting it on the Line

By Don Doerres

A colleague at my office and I were working on a project recently that entailed processing groups of files. Each small group of files was handled the same way, namely, dragging the filenames from a directory window to the window that did the processing. He said "Golly, if I know what files I want the program to work on ahead of time, why can't I just tell the program what I want it to do *when it starts*? Why do I have to load the GUI (Graphical User Interface) each time?" Good question indeed. What he was really asking about was the traditional user command line interface. This allows a user to pass parameters to a program as the program begins.

In the pre-Windows days, command lines were routinely used at the console. Computer users used to type something like

```
myprogram param1 param2
```

in a command console window (which used to be the entire screen!) in order to start the program "myprogram.exe" with the two parameters "param1" and "param2."

In the Windows of today, users simply use the mouse to select icons which launch the desired programs. Often, they then need to type additional information in the window or click buttons in the window. However, even Windows retains, in an elegant fashion, the traditional command line which passes information to a program as it starts. Command lines continue to be useful, even in a windowed environment.

The basics of argc and argv

In C and in C++, the start of a program is a function called `main()` with a prototype of

```
int main ( int argc, char *argv[] )
```

which returns an integer to the operating system when the program ends. The first parameter, `argc`, is the count of the number of parameters passed to the program. The second parameter, `argv`, is an array of null-terminated strings which denote the command-line literals. The number of strings (the dimension of `argv[]`) is `argc`. The first string, `argv[0]`, is the full path-name to the called function.

For the earlier example, `argc` would be three, and we would have something like:

```
argv[0] = "C:\MYPROGRAM"
argv[1] = "param1"
argv[2] = "param2"
```

The scope of the values `argc` and `argv` is only inside of the `main()` function:

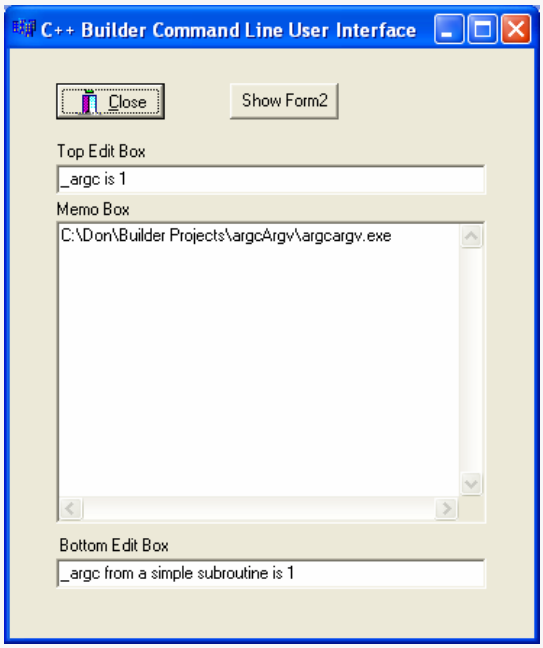
```
int main ( int argc, char *argv[] )
{
  // argc and *argv[] scope only in here
}
```

If you try to use `argc` and `argv` outside of `main()`, a compile-time error will result.

Builder's version of the command line

With the exception of a pure console application, the days of `main()` are long gone. Builder developers seldom see a `main()` function. However, the command

Figure A



First demo showing values of argc and argv in Form1.

line parameters are still there—they have simply evolved. Their new names are `_argc` and `_argv` (see also [1]).

These names are the same as before with the addition of leading underscores. The scope of the two parameters is global to a C++Builder project. They are available everywhere, not just inside of `main()`.

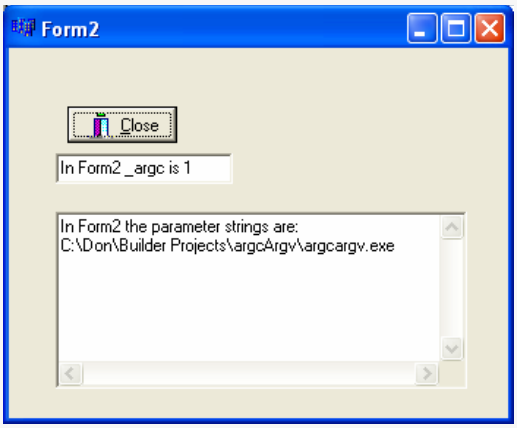
This brings us to a demonstration project. In building and executing the demonstration program associated with this article, one will observe the following: With no command line parameters, the value `_argc` will be one as shown in the top edit box of **Figure A**. The bottom edit box will show that `_argc` is available in subroutines—a major improvement over the old `argc`!

The memo box is filled with the strings from `_argv` by using the following code:

```
for (int i = 0; i < _argc; i++)
   {
   Memo1->Lines->Append(_argv[i]);
   }
```

By starting the program, there is no command line and there are no command-line arguments other than

Figure B



First demo showing values of argc and argv in Form2.

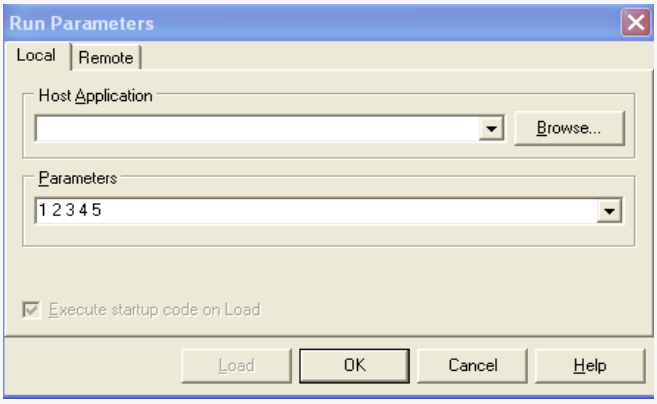
`_argv[0]`, which is the full path name of the executable program.

Pressing the "Show Form2" button yields the output shown in **Figure B**. Notice that `_argc` and `_argv` are truly global in the project.

Four ways to use the command line in Windows

The above is a Windows application. Despite the frequent use of icons and toolbars, there are four ways [2] to instead use a command line in Windows.

Figure C



Specifying parameters via the Run Parameters dialog.

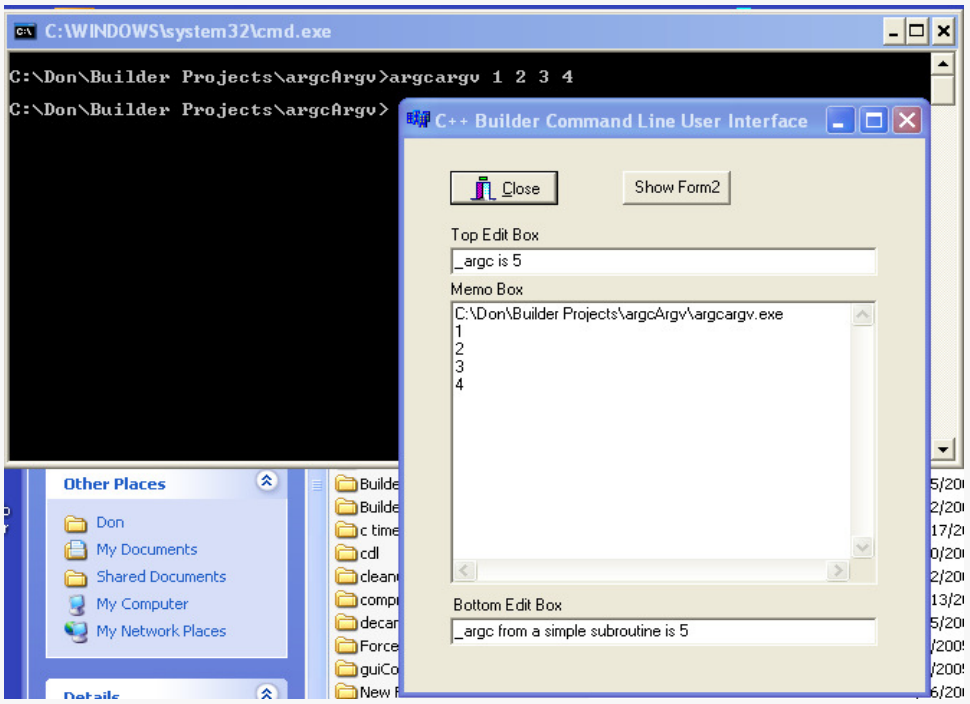
The first way is inside of the Builder IDE. Choosing "Run | Parameters" from the main menu allows the developer to type in command-line parameters into the Run Parameters dialog shown in **Figure C**.

The second way is to enter the command-line parameters after the executable name in a console window as shown in **Figure D**.

The third way is to enter the parameters into the "Open" edit box from the Windows "Start | Run" menu; this approach is demonstrated in **Figure E**. Again, enclose the path to the executable in double quotes.

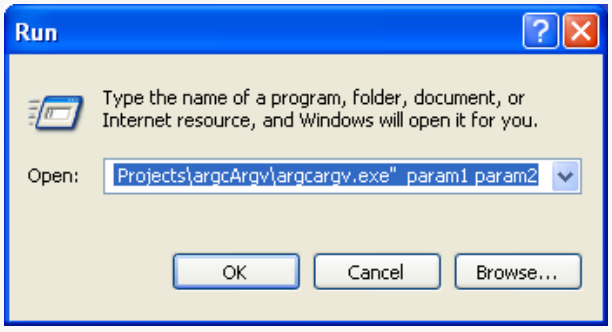
The fourth way is to enter the command-line parameters into the "Target" edit box of a shortcut properties window as shown in **Figure F**. Simply create a shortcut to the executable, and then edit the "Target" box. Notice that the full path name for the executable is in double quote marks. Do not enclose the individual parameters in the same set of quotes or Windows will protest that it cannot find the file.

Figure D



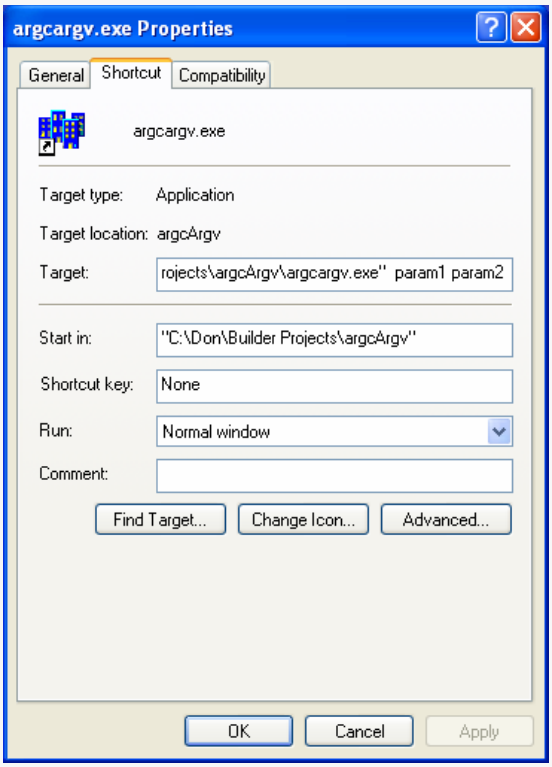
Specifying parameter via the command line.

Figure E



Specifying parameters via the Windows Run dialog.

Figure F



Specifying parameters via a shortcut's properties.

Some Final Comments

It is the case that the `_argc` and `_argv` parameters are available as soon as the project starts. This is where the information in the constructor of the main form is used.

It is difficult to find a reference to Windows command lines, but the rules are fairly simple:

1. The file-name is enclosed in double quotation marks.
2. Individual parameters after the double-quote-delimited file-name are simply separated by white space (spaces or tabs).
3. A multiple-word string must be enclosed in double quotes.

A simple example of all three rules is:

```
"aFile.exe" param1 "param2 string with
spaces" param3
```

Builder command-line parameters work well in applications repetitively. Command-line parameters available in an application are often much easier on the user than refilling edit boxes.

Contact Don at **trundlar@cox.net**.

References

1. See also the `ParamStr()` and `ParamCount()` VCL functions in the BCB help files.
2. A fifth method of specifying command-line parameters is to drop a file directly onto the executable; see also "Secondary Desktops" later in this issue.

Redirecting Console Output

By Malcolm Smith

Complex applications quite often incorporate tools from third-party vendors to perform an external task. As an example, I was once working with a proprietary DBMS that required the execution of several console applications, each with several command-line switches. To ensure the task was performed in a reproducible manner we executed these tasks via a batch file. After spending countless hours designing our award-winning GUI (at least we thought so) the last thing we wanted was the presence of a DOS window flashing on-screen sending numerous lines of text to a scrolling window. What we needed was the ability to run the batch file and redirect the console output to our GUI controls. In this article I'm going to show you how this was accomplished. The source code provided with this article includes a reusable object called `TConsoleRedirect` that you are free to use in your own projects [1].

We'll start by looking at how `TConsoleRedirect` wraps the execution of the batch file as this is where we initiate the redirection of the console output. From there we look at capturing the console output, parsing it, triggering events to the main application, and waiting for the process to complete.

TConsoleRedirect overview

Console applications use three types of standard streams for their input and output:

1. `stdin`—standard input, typically from a keyboard;
2. `stdout`—standard output, typically to the screen;
3. `stderr`—standard error, typically to the screen.

If you've used batch files, then you've probably heard of the term *redirection*. This refers to redirecting the standard input from a file or standard output to a file. Refer to [2] for more information on this.

`TConsoleRedirect` accomplishes redirection of the `stdout` and `stderr` streams by supplying the new process an anonymous (unnamed) pipe. For those not familiar with pipes, think of it as a communication conduit with read and write handles.

Before we go too much further, take a look at **Listing A** for the declaration of `TConsoleRedirect`. As you'll see, it contains only a handful of properties, a single public `Execute()` method, and an event that is triggered every time a new line of text is available. The properties are self-explanatory so I won't be discussing them in detail here. Refer to the source code's comments for a full explanation.

The `Execute()` method performs most of the work, all of which can be broken down into six steps:

1. Initialize security attributes for the pipe we are going to create;
2. Create a pipe to receive data from the process output and error streams;
3. Configure startup information for the new process;
4. Create the new process;
5. Read and parse text received from the created process and raise events in the GUI;
6. Perform a cleanup when the new process has completed executing.

The implementation of `TConsoleRedirect`'s `Execute()` method is a little long to publish in its entirety so I urge you to glance over the provided source code

at this time. Snippets of the implementation will be provided as we cover each of the abovementioned steps.

Each of the above steps center around a Win32 API method called `CreateProcess()`. Explaining each of the above six steps is a little difficult without first briefly covering the parameters passed to this method.

First up: CreateProcess()

`CreateProcess()` is a Win32 API call designed to create a new process and its primary thread. The new process runs in the security context of the calling process. If the calling process is impersonating another user, this context information is ignored. If you need to execute the process as another user then read up on `CreateProcessWithLogonW()` at [3]. This is something I want to look at myself but have not had an immediate need. Drop me a line if you think this would make an interesting topic for discussion.

`CreateProcess()` contains ten parameters in its declaration, which looks like this:

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName,
  LPTSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES
    lpProcessAttributes,
  LPSECURITY_ATTRIBUTES
    lpThreadAttributes,
  BOOL bInheritHandles,
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,
  LPCTSTR lpCurrentDirectory,
  LPSTARTUPINFO lpStartupInfo,
  LPPROCESS_INFORMATION
    lpProcessInformation
);
```

Listing A: Declaration of TConsoleRedirect

```
class PACKAGE TConsoleRedirect
{
private:
  AnsiString        FStartingDir;
  AnsiString        FApplicationName;
  AnsiString        FCommandLine;
  int               FYield;
  TConsoleLineEvent   FOnConsoleLine;
  TConsoleYieldEvent  FOnConsoleYield;

  LPSECURITY_ATTRIBUTES
    InitializeNTSecurityDescriptor(
      SECURITY_DESCRIPTOR &sd,
      SECURITY_ATTRIBUTES &sa);
  void ProcessReadData(TConsoleBuffer &ABuffer);
  void ProcessToken(TConsoleBufferCIter IterP,
    TConsoleBufferCIter IterT);
  void DoConsoleLineEvent(unsigned char *pData,
    unsigned int ALength);

  TConsoleRedirect(const TConsoleRedirect&);
  TConsoleRedirect& operator=(
    const TConsoleRedirect&);

public:
  TConsoleRedirect(void);
  void Execute(void);

  __property AnsiString StartingDir =
    {read = FStartingDir, write = FStartingDir};
  __property AnsiString ApplicationName =
    {read = FApplicationName,
     write = FApplicationName};
  __property AnsiString CommandLine =
    {read = FCommandLine, write = FCommandLine};
  __property int Yield =
    {read = FYield, write = FYield};
  __property TConsoleLineEvent OnConsoleLine =
    {read = FOnConsoleLine,
     write = FOnConsoleLine};
  __property TConsoleLineEvent OnConsoleYield =
    {read = FOnConsoleYield,
     write = FOnConsoleYield};
};
```

I view these parameters as belonging to three distinct groups:

1. `lpApplicationName`, `lpCommandLine` and `lpCurrentDirectory` relate to the process you want to spawn;
2. `lpProcessAttributes`, `lpThreadAttributes`, `bInheritHandles`, `dwCreationFlags`, `lpEnvironment` and `lpStartupInfo` control how the new process will be created;
3. `lpProcessInformation` contains information about the new process when it is successfully created.

Group 1 parameters

The first group of parameters *identifies* the process to be spawned, the "current directory" from which it is to be spawned, and any command-line information that needs to be passed to the process. For reference,

TConsoleRedirect includes these properties to cover this group:

- ApplicationName
- StartingDir
- CommandLine

Group 2 parameters

The second group dictates *how* the process will be created. We only need to set three of these parameters to non-default values, namely bInheritHandles, dwCreationFlags, and lpStartupInfo.

Console applications use the stdin, stdout, and stderr streams for their input, output, and error-reporting, respectively. Redirecting the console output requires we supply our own output and error handles (via a pipe as you'll see shortly) to the new process. In order to do this, we need to tell CreateProcess() that the new process must inherit our handles. For this reason we will be setting the bInheritHandles parameter to true.

dwCreationFlags controls the priority class for the new process. The priority class is used by the operating system to schedule thread priorities in the new process. In TConsoleRedirect I use the recommended default of NORMAL_PRIORITY_CLASS.

Finally, lpStartupInfo is a pointer to a STARTUPINFO structure [4]. Amongst other things, this structure contains these three handles: hStdInput, hStdOutput, and hStdError. When we create our process we will be setting hStdOutput and hStdError to point to our new pipe.

Group 3 parameters

The last group provides *resultant* information (handles and numerical identifiers) about the new process and its main thread. We use this information to determine when the process has completed its task.

Putting it all together

With the bare-bones information about CreateProcess() out of the way, let's now work through each stage and put the pieces together.

Initializing security attributes

When providing a process with a pipe to replace the standard input, output, or error streams, the new process must inherit the handles of the new pipe, otherwise the process will revert to using the default stdin, stdout, and stderr streams.

As already mentioned, TConsoleRedirect uses CreateProcess() to create the new process. We've also mentioned that we set the bInheritHandles parameter to true. This ensures that all *inheritable* handles are inherited. This implies that when we create our pipe, the handles of that pipe need to be inheritable.

In the second stage (to be discussed shortly) we use another Win32 API method called CreatePipe() to create our anonymous pipe. One parameter to this call is a pointer to a struct of type SECURITY_ATTRIBUTES. This parameter tells CreatePipe() if the handle it returns can be inherited by child processes.

SECURITY_ATTRIBUTES is defined as this:

```
typedef struct _SECURITY_ATTRIBUTES {
  DWORD nLength;
  LPVOID lpSecurityDescriptor;
  BOOL bInheritHandle;
} SECURITY_ATTRIBUTES;
```

TConsoleRedirect initializes this structure as follows:

```
SECURITY_DESCRIPTOR sd = { 0 };
SECURITY_ATTRIBUTES sa = { 0 };
LPSECURITY_ATTRIBUTES lpsa =
  InitNTSecurityDescriptor(sd, sa);
```

where InitNTSecurityDescriptor() is implemented like so:

```
LPSECURITY_ATTRIBUTES
TConsoleRedirect::InitNTSecurityDescriptor(
  SECURITY_DESCRIPTOR &sd,
  SECURITY_ATTRIBUTES &sa)
{
LPSECURITY_ATTRIBUTES lpsa = NULL;

OSVERSIONINFO osv;
osv.dwOSVersionInfoSize =
  sizeof(OSVERSIONINFO);
::GetVersionEx(&osv);

// NT-based platform
if(osv.dwPlatformId ==
  VER_PLATFORM_WIN32_NT)
  {
  ::InitializeSecurityDescriptor(&sd,
    SECURITY_DESCRIPTOR_REVISION);
  ::SetSecurityDescriptorDacl(&sd, true,
    NULL, false);
```

```
  sa.nLength = sizeof(SECURITY_ATTRIBUTES);
  sa.bInheritHandle = true;
  sa.lpSecurityDescriptor = &sd;

  lpsa = &sa;
  }

return lpsa;
}
```

Because `CreatePipe()` needs to create an inheritable handle, the `lpSecurityDescriptor` member of `SECURITY_ATTRIBUTES` will need to point to a security descriptor that indicates this requirement. A full discussion of security descriptors is beyond the context of this article but the main points to make about the above code are the following:

- `TConsoleRedirect` requires an NT-based operating system (please refer to the section "Some finer points of the implementation" for information relating to non-NT-based systems);
- The `bInheritHandle` member of the security attribute is set to `true`.

If you'd like to know more about security descriptors and attributes, refer to [5] at MSDN.

Creating a pipe

Now that we've initialized our security attributes structure, it's time to create the anonymous pipe. This is achieved with these few lines of code:

```
HANDLE hReadPipe = INVALID_HANDLE_VALUE;
HANDLE hWritePipe = INVALID_HANDLE_VALUE;
::CreatePipe(
  &hReadPipe, &hWritePipe, lpsa, 0
  );
```

If `CreatePipe()` is successful it will have created an anonymous pipe and provided the read and write handles of the pipe. The last parameter in `CreatePipe()` indicates the size of the buffer to be used for the pipe. I'm using a value of zero which instructs the operating system to use a default size.

Now we need to prepare the startup information that will be used by `CreateProcess()`.

Setting startup parameters

The `STARTUPINFO struct` is used by `CreateProcess()` to control several aspects about the new process, including the window station, desktop, standard handles, and appearance of the main window in the new process. We are interested only in these latter two items. The initialization of our `STARTUPINFO struct` therefore looks like this:

```
STARTUPINFO si;
memset(&si, 0, sizeof(STARTUPINFO));
si.cb = sizeof(STARTUPINFO);
si.dwFlags = STARTF_USESHOWWINDOW |
             STARTF_USESTDHANDLES;
si.wShowWindow = SW_HIDE;
si.hStdOutput = hWritePipe;
si.hStdError = hWritePipe;
```

The first three lines declare and initialize the contents of the structure. The `dwFlags` member must be initialized with `STARTF_USESSHOWWINDOW` if the `wShowWindow` member is to have an effect. We are making the new process non-visible. Similarly, `dwFlags` must also be initialized with `STARTF_USESTDHANDLES` if we want our pipe to be used in place of the standard output and error streams. This replacement can be seen with the assignment to the `hStdOutput` and `hStdError` members. `STARTUPINFO` also includes a `hStdInput` member but we do not require this for our needs.

Everything required to create the process and redirect the console output to our pipe is now in place. So, let's move onto creating the process.

Creating the process

Listing B provides an extract of code from `TConsoleRedirect` which shows how the pieces are placed together to create the new process.

If the creation of the new process is successful, the local variable `pi` (of type `PROCESS_INFORMATION`) will receive information about the new process. In particular, we will have access to the handle of the process., which allows us to test to see if the process has completed its task. This will be explained in greater detail a little later.

First, it's time to look at how we capture data from the pipe, parse it, and then present it to the GUI via an event handler.

Reading and parsing received data

Capturing data in the pipe involves the use of two simple API calls: `PeekNamedPipe()` and `ReadFile()`. `PeekNamedPipe()` checks to see if there are any data available in the pipe without removing it. If data are

available, then `ReadFile()` is used to read the data and remove the contents from the pipe.

`TConsoleRedirect` uses a `typedef` called `TConsoleBuffer` to represent a `std::vector` of `unsigned char`:

```
typedef std::vector<unsigned char>
  TConsoleBuffer;
```

Two instances of this type are used within the `Execute()` method of `TConsoleRedirect`:

```
TConsoleBuffer CachedLine;
TConsoleBuffer Buffer(BlockSize, ' ');
```

`CachedLine` is used to buffer received data until a full line of text (defined as ending in `\r\n`) is received. This buffer grows dynamically as required.

`Buffer`, on the other hand, is a fixed-size buffer used to store data captured from the pipe. This data is then copied/appended to `CachedLine` and processed as required.

It's important to note here that our `std::vector` of `unsigned char` is equivalent to an array such as:

```
// BlockSize = 1024
unsigned char Buffer[1024];
```

By using a `std::vector` (guaranteed to be implemented as a single contiguous block of memory), we have the advantage of being able to specify the buffer size at run-time (by changing `BlockSize`). Also, if we later decide to modify our design such that `Buffer` can dynamically change size at run-time, then we already have the mechanism in place.

Checking for the availability of data in the pipe is achieved with this line of code:

```
::PeekNamedPipe(
  hReadPipe,     // pipe identifier
  pBuffer,       // pointer to our buffer
  BlockSize,     // available block size
  &BytesRead,    // number of bytes read
  &TotalBytes,   // total bytes available
  NULL           // bytes left (N/A)
  );
```

`PeekNamedPipe()` can be used in two ways. First, it can be used to determine how many bytes of data are available in the pipe without actually reading them. Secondly, it can be used to determine the number of available bytes as well read the available data into a buffer without removing them from the pipe. I'm using this latter approach because it allows me to check for the existence of an end-of-line token (`\r\n`) and read only the data I'm interested in. I could read the entire contents of the pipe, but then the implementation of the parsing would be a little more involved.

Listing B: Creating the new process

```
PROCESS_INFORMATION pi = { 0 };

if(::CreateProcess(
    // application name
    FApplicationName.c_str(),
    // command line
    FCommandLine.c_str(),
    // default process security attributes
    NULL,
    // default thread security attributes
    NULL,
    // inherits handles
    true,
    // creation flags
    NORMAL_PRIORITY_CLASS,
    // environment block
    NULL,
    // starting directory
    FStartingDir.c_str(),
    // STARTUPINFO
    &si,
    // PROCESS_INFORMATION
    &pi))
  {
  // process is now created...
  }
```

Assuming data are available for reading, the following code is executed to read and parse it:

```
// point to first byte of buffer
unsigned char *pBuffer = &(Buffer[0]);

if(BytesRead)
  {
  BytesToRead = BytesRead;
  OrigBytesRead = BytesToRead;
  BytesRead = 0;

  // find where the last \r\n is located
  // and modify the value of BytesToRead
  while( BytesToRead > 0 &&
         ( *(pBuffer + BytesToRead - 1) !=
           '\n' ) )
    --BytesToRead;

  if(BytesToRead == 0)
    {
```

```
    // an incomplete line of text has
    // been captured so we will store
    // it and wait for more
    ::ReadFile(hReadPipe, pBuffer,
      OrigBytesRead, &BytesRead,
      NULL);

    // store contents in CachedLine
    // for future use
    std::copy(
      pBuffer, pBuffer + BytesRead,
      std::back_inserter(CachedLine)
      );
    }
  else
    {
    // we have one or more lines
    // of data to process
    ::ReadFile(hReadPipe, pBuffer,
      BytesToRead, &BytesRead, NULL);

    // append the read line onto CachedLine
    std::copy(
      pBuffer, pBuffer + BytesRead,
      std::back_inserter(CachedLine)
      );

    if(FOnConsoleLine)
      ProcessReadData(CachedLine);
    CachedLine.clear();
    }
  }
```

First up, the `while` loop in this code determines whether there is an end-of-line token. Specifically, starting from the end of the buffer and working backwards, the `while` loop searches for a newline character. If one is not located (the loop exits with `BytesToRead` equal to zero) then we have an incomplete line of text. In this situation, the full buffer is read via the `ReadFile()` Win32 function, and then the read data are stored in `CachedLine` by using the `std::copy()` function.

On the other hand, if the previously mentioned `while` loop located a newline character, then the contents up to and including the newline character are read and appended to `CachedLine`. Next, if an event handler is assigned to `FOnConsoleLine`, the captured data are parsed via `ProcessReadData()`, after which the `CachedLine` buffer is cleared.

`ProcessReadData()` takes `CachedLine` and searches for multiple lines of text. Each line of text located is then processed via `ProcessLine()` where the buffered data are copied to an `AnsiString` and sent to the GUI via the event handler. `ProcessReadData()` and `ProcessLine()` are shown in **Listing C** and **D** respectively.

This procedure continues until no more data are available for reading and the process has completed. As discussed next, detection of process completion is achieved by using `WaitForSingleObject()`.

Waiting for the process to finish

The pseudo-code below represents the internal processing performed by `TConsoleRedirect` once the new process is running:

```
if( CreateProcess() )
  {
  for(;;)
    {
    if( PipeContainsData )
      {
      // Read and parse data
      }
    else
      {
      if(::WaitForSingleObject(pi.hProcess,
          0) == WAIT_OBJECT_0)
        break;
      }
    }
  }
```

There is an infinite loop that continuously checks for data to read from the input handle of the pipe. If there are no data to read, then a check is made to see if the process has completed its task. This is achieved by checking if the return value of `WaitForSingleObject()` equates to the constant `WAIT_OBJECT_0`. When this state is reached, the process is finished and the code breaks from the loop. `TConsoleRedirect` has completed its task.

Some finer points

Due to the amount of code implemented in `TConsoleRedirect`, I cannot publish all of it on these pages. I do however ask that you take a few minutes to look at the code to observe some of the finer details in the implementation—all of which help make the code safer to use.

Limitations

`TConsoleRedirect` is designed primarily for NT-based operating systems. I have not personally tested

TConsoleRedirect under Windows 95 or 98 but from what I've read at [6][7][8] it should work as long as you are executing only 32-bit processes. You might also find the alternative approach at [9] to be informative reading.

Exceptions

In most cases (but not all) I have elected to check the return value from the Win32 API calls and throw a custom exception (as defined in my MJFAF library) in preference over aborting the operation (by just returning). I've made an assumption that once the initial startup pre-conditions are met everything will be fine from that point onward. As is, the code will suit the majority of circumstances.

Guards

The Execute() method of TConsoleRedirect ultimately ends up with four handles that need closing at the completion of the process. Rather than use nested try/__finally blocks to protect these resources, you will find I'm using another class from MJFAF called THandleGuard. This class uses the RAII design pattern to ensure the handle is closed when it goes out of scope.

Terminating the running process

TConsoleRedirect has a OnConsoleYield event that is called at an interval equal to the Yield property (in milliseconds) after calling TApplication::ProcessMessages(). This event serves two purposes:

1. The GUI has a chance to repaint itself.
2. You have an opportunity to abort the running process. The comments in the source code explain this feature.

An exercise for the reader (or me)

I have an extreme dislike for making calls to Application->ProcessMessages(), especially in multithreaded applications. A nice enhancement to the provided code would be to place the execution of the new process within a worker thread. Not only would it eliminate calling TApplication::ProcessMessages() but you would also be capable of running the process asynchronously.

Listing C: Processing data read from the pipe

```
void TConsoleRedirect::ProcessReadData(
  TConsoleBuffer &ABuffer)
{
TConsoleBufferIter IterB = ABuffer.begin();
TConsoleBufferIter IterE = ABuffer.end();
TConsoleBufferIter IterF;

while( (IterB != IterE)  &&
  (IterF = std::find(IterB, IterE, '\r'))
  != IterE )
  {
  ProcessLine(IterB, IterF);

  // update the starting location
  IterB = IterF;

  // skip over the '\r'
  ++IterB;

  // if the next char is a '\n', skip over it;
  // don't skip over others (blank lines)
  if( (IterB != IterE) && (*IterB == '\n') )
    ++IterB;

  }

// one last check in case the end of the
// line did not end with /r/n
if(IterB != IterE)
  ProcessLine(IterB, IterE);
}
```

Listing D: Processing a line of text

```
void TConsoleRedirect::ProcessLine(
  TConsoleBufferCIter IterP,
  TConsoleBufferCIter IterT)
{
static AnsiString ALine;

TConsoleBuffer::difference_type Distance =
  std::distance( IterP, IterT );
if(Distance == 0)
  {
  // must have been a blank line
  ALine = "";
  FOnConsoleLine(this, ALine);
  }
else
  {
  ALine.SetLength(Distance);
  memcpy( ALine.c_str(), IterP,
    Distance );
  FOnConsoleLine(this, ALine);
  }
}
```

I initially leave this as an exercise for the reader. If, however, I receive enough requests from readers then I'll make this enhancement and make it available for all subscribers.

Hint, hint: I'd love to hear your comments regarding the topics being covered in recent months. I have a huge list of topics already lined up for the forthcoming months, but please feel free to drop me a line if you have something that would benefit other readers as well.

Conclusion

Even as operating systems become more and more advanced, we sometimes still take advantage of simple features such as batch files. The downside to executing batch files is that they invoke console windows which may not fit into the design and layout of your GUI applications. This article demonstrated how to eliminate this annoyance by redirecting the `stdout` and `stderr` streams, and by providing the captured text to the GUI via an event handle.

We also looked at some of the inner details associated with the Win32 API calls being made. Hopefully this has provided a more complete understanding of what is being performed under the hood, which ultimately helps in your understanding of how certain tasks are achieved within the Windows environment.

Contact Malcolm at **msmith@bcbjournal.com**.

References

1. `TConsoleRedirect` is part of my MJFAF so please read the copyright notice in the source code.
2. Information about the command processor shell and input/output redirection can be found at http://tinyurl.com/5w8z8
3. http://tinyurl.com/5yrhl
4. `STARTUPINFO` is fully detailed at http://tinyurl.com/4yag8
5. http://tinyurl.com/5awg8
6. http://tinyurl.com/3wczm
7. http://tinyurl.com/3uzco
8. http://tinyurl.com/4txg6
9. http://tinyurl.com/guro

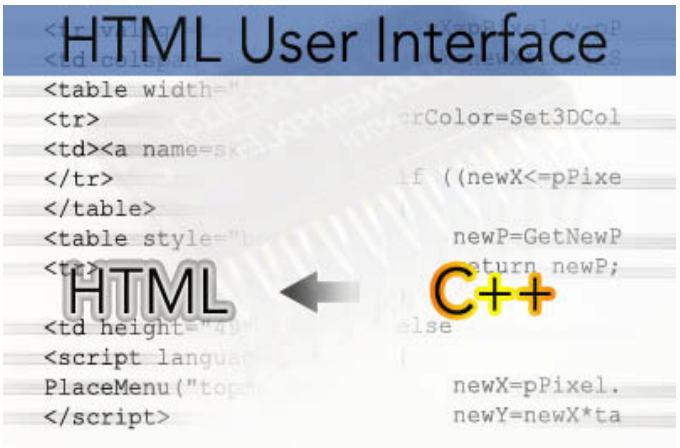
Have a question or comment?
Visit our online forums at
http://forums.bcbjournal.com.

We welcome and appreciate your feedback on this Special Issue!

Please send any comments to editor@bcbjournal.com or post your comments to http://forums.bcbjournal.com.

Creating an HTML User Interface

By Mark Finkle



HTML allows for the creation of clean, simple and usable interfaces. More and more desktop applications are employing HTML in the user interface (UI). Other applications are using modified Windows controls made to look like HTML elements, such as labels that look like hyperlinks.

HTML in desktop applications

There are various levels of HTML interactivity that can be used in a desktop application:

1. Static display: Simple, richly formatted text area used for display-only purposes.
2. Basic interactivity: Richly formatted text with hyperlinks used to execute an action in the application.
3. Deep interactivity: HTML-based forms with controls tied to application code and data.

C++Builder includes the `TCppWebBrowser` component, which allows developers access to the same HTML rendering system used in Internet Explorer. The component is a VCL wrapper around the WebBrowser ActiveX control. As such, it does not contain some features you would expect from a native VCL control. Simple features you might expect to find include:

- Loading/saving the contents using a `TStream`
- Controlling the border
- Controlling the background color
- Access to DHTML events

One of the goals of this article is to create a simple `TFrame`-based wrapper for `TCppWebBrowser` that adds some convenient features which make it easy to create an HTML UI.

Loading HTML

No matter what level of interactivity your applications needs, the first step is usually getting HTML into `TCppWebBrowser`. Static display and basic interactivity can be handled by `TCppWebBrowser` fairly well. If your HTML content is stored in a file, you can always use the `Navigate` or `Navigate2` methods of `TCppWebBrowser` as shown below:

```
wbBrowser->Navigate(
  WideString("file://c\path\text.htm"),
             TVariant(::navNoHistory));
```

The `::navNoHistory` flag is used so the file does not show up in IE's browser history. If your content is dynamically generated, a `LoadHtmlFromStream()` method would be much more convenient. Though not included in `TCppWebBrowser`, it can be implemented fairly easily:

```
HRESULT LoadHtmlFromStream(
  TCppWebBrowser* pBrowser,
  TStream* pStream)
{
  IHTMLDocument2* pHTMLDoc;
  HRESULT hr = pBrowser->Document->
    QueryInterface(IID_IHTMLDocument2,
                   (void**)&pHTMLDoc);
  if (SUCCEEDED(hr)) {
    IPersistStreamInit* pPersist = NULL;
    hr = pHTMLDoc->QueryInterface(
      IID_IPersistStreamInit,
      (void**)&pPersist);
```

```
    if (SUCCEEDED(hr) && pPersist) {
      hr = pPersist->InitNew();
      if (SUCCEEDED(hr)) {
        TStreamAdapter* pAdapter =
          new TStreamAdapter(pStream,
              soReference);
        hr = pPersist->Load(*pAdapter);
      }
      pPersist->Release();
    }
  }
  pHTMLDoc->Release();
  return hr;
}
```

Loading HTML from a resource

There may be circumstances in which you do not want to load HTML from physical files. HTML files can be added to your application as resources and loaded a couple of ways. One way is to access the HTML resource using a `TResourceStream` and then load that stream into a `TCppWebBrowser` using the method described above.

Add your HTML content to your application by including a resource script (.RC) file to your project. Add the following code to the .RC file:

```
IDH_HTMLPAGE_1   HTML  DISCARDABLE
  "htmlpage1.htm"
IDH_HTMLPAGE_2   HTML  DISCARDABLE
  "htmlpage2.htm"
```

Load the HTML resource using `TResourceStream` and use it to load the browser:

```
TResourceStream* pStream =
  new TResourceStream((int)HInstance,
  "IDH_HTMLPAGE_1", "HTML");
pStream->Position = 0;
LoadHtmlFromStream(pMyBrowser, pStream);
delete pStream;
```

A second way of loading the HTML from a resource is using the `res://` protocol. This method is useful for creating `IMG` or `STYLE` links from the main HTML content to external, supporting content, stored as resources:

```
<html><body>
  <img src="res://pathtoexe/HTML/
IDH_RESOURCE">
Click here for help</a>
</body></html>
```

Getting interactive

Using hyperlinks to achieve a basic level of interactivity is easily accomplished by using pseudo-URL references in HTML anchor elements. For example, your HTML could contain the following content:

```
<html><body>
  <a href="app:displayhelp">Click here for
help</a>
</body></html>
```

The `HREF` in the anchor element is fake, but it can be used as a text command for your application to act upon. You can use the `OnBeforeNavigate` event to catch any click on the anchor, extract and execute the command, and cancel the navigation. It's simple, but effective. Below is an example of extracting the command from the `HREF` string.

```
void __fastcall TForm1::OnBeforeNavigate(
  TObject *Sender, LPDISPATCH pDisp,
  Variant *URL, Variant *Flags,
  Variant *TargetFrameName,
  Variant *PostData, Variant *Headers,
  VARIANT_BOOL *Cancel)
{
  WideString sURL = (WideString)*URL;

  // Check for a "command"
  if (sURL.Pos(L"app:") == 1) {
    *Cancel = VARIANT_TRUE;
    WideString sCommand =
      sURL.SubString(5, sURL.Length());
    if (sCommand == "displayhelp") {
      // display help
    }
  }
}
```

Going deeper: Using MSHTML

In order to implement deeper interactivity, we need to explore the HTML document object model (DOM) upon which the WebBrowser ActiveX control is built. Called MSHTML, it is the COM-based system used to implement the client-side Dynamic HTML (DHTML) abilities of Internet Explorer. Typically, these features are used only in client-side JavaScript or VBScript scripting inside Internet Explorer. However, because of the way Microsoft implemented those scripting languages, the HTML DOM is COM and can be accessed by any COM-enabled development language. See [1] for documentation.

The TCppWebBrowser control provides a way to access MSHTML COM objects via the Document property as shown below:

```
IHTMLDocument2* pDoc2 = NULL;
wbBrowser->Document->
  QueryInterface(IID_IHTMLDocument2,
                 (void**)&pDoc2);
```

If the QueryInterface() call is successful (content must be loaded into the control first), the IHTMLDocument2 interface can be used to do a multitude of different actions. Actually there are several versions of the IHTMLDocument interface. Each new version adds different functionality.

Customizing TCppWebBrowser the MSHTML way

Although MSHTML is the system used to build Internet Explorer, you may not want it to act or look like a web browser. By implementing the IDocHostUIHandler interface [2], you can control various aspects of the WebBrowser behavior and appearance. Here is a short list of things you can affect:

- Display of border and scrollbars.
- Use of Windows XP themes on HTML UI elements.
- Selection of text with the mouse.
- Control over the default right-click context menu.
- Expose application specific functions into JavaScript

The code accompanying this article contains a simple implementation of IDocHostUIHandler.

Hooking C++ to DHTML events

The VCL framework makes it rather easy to hook C++ code to UI control events. Each control publishes available events and the developer can hook those events at design-time or run-time. HTML elements also expose events. However, hooking those events is more complicated than the VCL counterparts. The MSDN reference website has all the details on the events each type of HTML element exposes [3].

MSHTML exposes the events through the COM IDispatch interface. The key to hooking C++ code to MSHTML events is making a C++ function act like an IDispatch interface and attaching that interface to the desired event. The code provided with this article uses a helper class, CHtmlEvent<>, to handle the minimum required pieces of IDispatch and forward an event to a provided C++ method. Here is an example of hooking the DHTML onkeydown event to a method of a TForm-derived class:

```
// Assume pElement is the HTML element
// you want to hook

// Get IHTMLElement2 to attachEvent
IHTMLElement2* pElement2;
pElement->
  QueryInterface(IID_IHTMLElement2,
                 (void**)&pElement2);
TVariant vFuncObj =
  CHtmlEvent<TMyForm>::Create(this,
  &TMyForm::OnEventCallback,
  DISPID_KEYDOWN);

// Link onkeydown to OnEventCallback
VARIANT_BOOL bSuccess;
pElement2->
  attachEvent(WideString("onkeydown"),
  vFuncObj, &bSuccess);
pElement2->Release();
```

In this example, anytime the onkeydown event of pElement is fired, TMyForm::OnEventCallback is called.

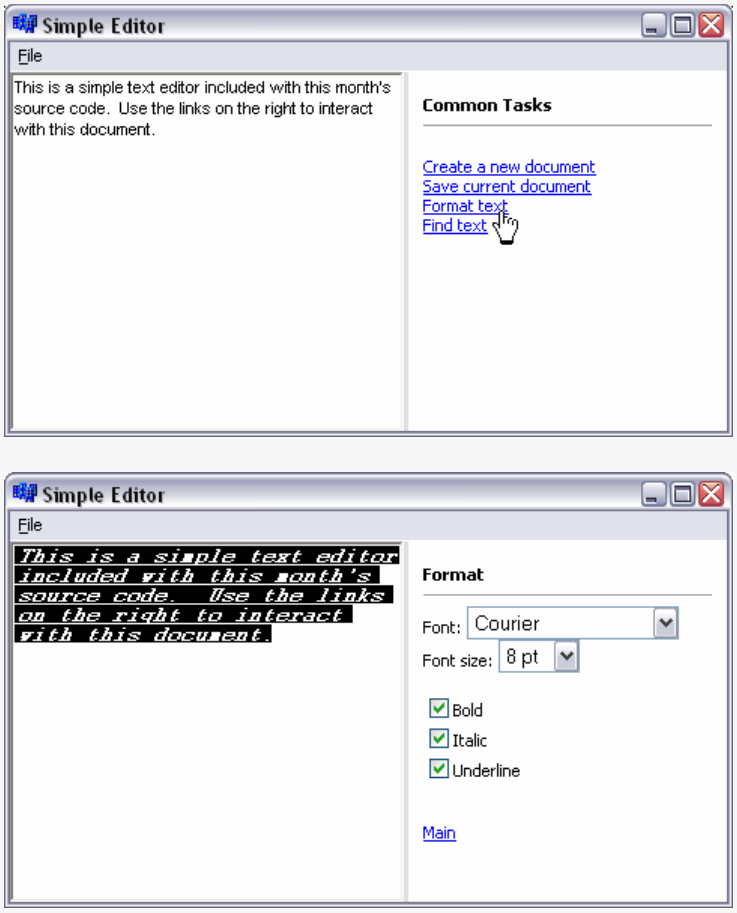
Calling C++ from JavaScript

In DHTML, much of the dynamic capabilities are made possible by JavaScript. Your HTML content can also use JavaScript to implement dynamic features without ever needing to resort to calling C++. You could write as much of the event-based code needed by your UI in JavaScript, where things are simpler and well documented. You would then only need to call into C++ to execute actions or commands that affect the application. MSHTML provides a way to do this via the window.external() method. The trick is exposing your C++ objects as IDispatch-based COM objects and providing the interface through the IDocHostUIHandler::GetExternal() method. The code provided with the article does this using the simple IDocHostUIHandler implementation and the CHtmlExternal<> base class.

First, a C++ `IDispatch` imposter is defined as follows:

```
class CMyExternal :
  public CHtmlExternal< CMyExternal >
{
public:
  void __stdcall setTitle(BSTR sText);
  BSTR __stdcall upperCase(BSTR sText);
  void __stdcall onDataChange();

BEGIN_DISPATCH_MAP(CMyExternal)
  DISP_METHOD1(setTitle, VT_EMPTY,
               VT_BSTR)
  DISP_METHOD1(upperCase, VT_BSTR,
               VT_BSTR)
  DISP_METHOD0(onDataChange, VT_EMPTY)
END_DISPATCH_MAP()
};
```

Figure A



Screenshots of sample application included with this month's code.

A pointer to an instance of this class is provided to the `IDocHostUIHandler` implementation. Then, methods can be called from within JavaScript:

```
<script language="javascript">
function doSomething()
{
  window.external.setTitle("From
    JavaScript");
  window.external.onDataChange();
}
</script>
```

Conclusions

The article source code shows how these different methods can be used in a simple application (**Figure A** shows some screenshots). MSHTML and the WebBrowser ActiveX control are sizeable technologies. We have just begun to scratch the surface of what can be done. Hopefully, this article will help you begin to explore further. The MSDN website is good place to start. It has useful reference and tutorial pages.

Contact Mark at **mark.finkle@gmail.com**.

References

1. MSDN MSHTML reference: http://tinyurl.com/c9csb
2. MSDN IDocHostUIHandler reference: http://tinyurl.com/bvepn
3. MSDN DHTML events reference: http://tinyurl.com/9jyya

Secondary Desktops

By Malcolm Smith

If you're writing a security-oriented or kiosk-type application that demands usage of the entire desktop, hides the windows shell, and prevents other dialogs from polluting the screen, then a secondary desktop is what you require.

A secondary desktop, also known as a private desktop, is actually the combination of a *window station* and a *desktop*. Each of these terms is important to understand so we'll start by looking at what they are.

(Note that secondary desktops are not available on operating systems below Windows NT 3.51. The source code provided with this application has been tested under Windows 2000 and Windows XP.)

Window Stations

A window station is a securable object associated with a process. Each window station contains a clipboard, atom table [1], and one or more desktops. When window stations are created they must be given unique names to differentiate them.

The system contains what is known as the *interactive window station* that goes by the name of Winsta0. This is the only window station capable of displaying a user interface or receiving user input. It is automatically assigned to the logon session of the interactive user, and it contains the keyboard, mouse, and video display.

Window stations created by you will always be non-interactive. In other words, they cannot be used to display user interfaces or receive user input. As we are interested in creating interactive desktops I will not be covering the creation of new window stations. Further information on this subject can be obtained from MSDN [2].

Desktops

A desktop is a securable object which, when created, is associated with the current window station of the calling process and assigned to the calling thread (an important fact to remember for later).

Each desktop has a logical display surface capable of displaying dialogs and other user interface items. If you need to send messages, or hook messages intended for other windows, this can be achieved only between processes that exist on the same desktop.

Desktops associated with WinSta0 can all display a user interface and receive user input, but only one of the desktops can be active at any given time. The currently active desktop is known as the *input desktop*. By default, WinSta0 contains three desktops:

1. *Default*—created when the user logs on;
2. *Disconnect*—used by Terminal Services [3];
3. *Winlogon*—created for users to log on.

Before we begin

The previous sections were short in their description, but they provide enough background information for creating simple secondary-desktop applications. I say "simple" because non-interactive processes, such as services, are trickier to deal with (especially when they are running under a different user account).

There are just over a dozen API functions available for desktop creation and management. Most of them are used in the two demo applications provided with this article. A complete list of functions is provided at MSDN [4].

The first demo, DESKTOPLAUNCH, is a menu-driven interface for creating and switching between secondary desktops. As you create new desktops, they are listed under the "Desktops" menu. This menu is dynamically created and ensures that all desktops created by the current desktop are listed together. Desktops created by another desktop are listed under the "Desktops | Others" menu. This scheme allows you to determine which desktops are children of the current desktop.

The second demo uses a small "desktop stub" executable, called DSTUB.EXE, that is designed to create a secondary desktop and either launch an executable or load a DLL that displays a main form via a predefined interface. The sample DLL provided, VCLKIOSK.DLL, creates a data module and a form that presents a small database application to the user. When the application closes, so does the desktop.

Creating desktops

Creating desktops faultlessly can be problematic to achieve, so I highly recommend you don't have too many other applications running while developing your project (just in case you cannot return to the primary desktop to close them). Whenever I found myself stuck in a situation with a new desktop and no application (because it was on the wrong desktop or it threw an exception before it was shown) my only option was to log off or shutdown. Task Manager is visible only on the *Default* desktop so it cannot be used to terminate any running applications.

When I first started creating new desktops, I found it a bit of a hit-and-miss affair when trying to determine the correct approach to obtain the desired results. For this reason, DESKTOPLAUNCH will be used to explain each of the steps required and their order of execution.

Step 1: Create a worker thread

Each process is associated with the thread in which it is created. Additionally, newly created desktops are associated with the calling thread. This tells us we need to start by creating a new thread that is responsible for creating the secondary desktop and launching the application to be displayed.

Each of the demo applications use a `TThread`-derived class called `TDesktopThread`. This thread is designed to create the new desktop and execute the code (via an event) required to start the new process. Both of these tasks are performed within the context of the thread.

Listing A: Declaration of TDesktopThread

```
class TDesktopThread;
typedef void __fastcall
  (__closure *TOnDesktopThreadEvent)
  (TDesktopThread *Sender,
   const TDesktopData &AData,
   HDESK ADesktopHandle);
typedef void __fastcall
  (__closure *TOnDesktopCreatedEvent)
  (TDesktopThread *Sender,
   const TDesktopData &AData,
   bool ASuccess);

class TDesktopThread : public TThread
{
private:
  const TDesktopData&    FData;
  TOnDesktopThreadEvent  FThreadedEvent;
  TOnDesktopCreatedEvent FOnDesktopCreated;
  bool                   FDesktopCreated;

protected:
  virtual void __fastcall Execute();
  void __fastcall DoOnDesktopCreated();

public:
  explicit __fastcall TDesktopThread(
    bool CreateSuspended,
    TOnDesktopThreadEvent AThreadedEvent,
    const TDesktopData &AData );

  __property TOnDesktopCreatedEvent
    OnDesktopCreated =
    { read = FOnDesktopCreated,
      write = FOnDesktopCreated };
};
```

I chose to execute the threaded code via an event handler—rather than within `TDesktopThread::Execute()`—for several reasons:

- Calling back into the main form (or other creator class) allows access to information that only the creator knows about. This helps to reduce coupling between the thread class and its creator.
- The thread class can be reused for different tasks. This is demonstrated in DSTUB where the thread launches an application or loads a DLL depending on the command-line parameters passed to it.

Listing A shows the declaration of `TDesktopThread` and **Listing B** provides the implementation of its `Execute()` method.

`TDesktopThread`'s explicit constructor requires a pointer to the event handler to be called once the desktop has been created, as well as some context information. This context information is sent to the event handler provided in the constructor.

`TDesktopData` is a base class containing three properties: `DesktopName`, `DesktopFlags`, and `DesiredAccess`. This information is used by `TDesktopThread::Execute()` to create the desktop which we'll look at next.

Step 2: Create the desktop

The first job performed by `TDesktopThread` is the creation of the secondary desktop via the API function `CreateDesktop()`. Only three parameters are required: a unique name and two flags that control the process hooking and security attributes of the desktop.

The flags are made up of standard and generic access rights. The standard access rights correspond to operations specific to the desktop, and the generic access rights relate to objects contained within the interactive window station of the user's logon session (see [6] and [7]). DESKTOPLAUNCH includes a menu option to create a secondary desktop based on custom flag values.

Once the desktop has been created, the `OnDesktopCreated` event is called. The context data and a Boolean indicating if the desktop was successfully created are passed as parameters to the event handler.

The final steps involve switching to the new desktop and calling the event you will use to launch an application (or load a DLL). Switching to the new desktop is performed via a class called `TDesktopGuard`, which implements the RAII design pattern to ensure the previous desktop is restored during destruction. Here's the `TDesktopGuard` constructor:

```
TDesktopGuard::TDesktopGuard(
  HDESK ANewDesktop)
  : FNewDesktop(ANewDesktop)
{
FOriginalDeskThread =
  ::GetThreadDesktop(GetCurrentThreadId());
FOriginalDeskInput =
  ::OpenInputDesktop(0, false,
    DESKTOP_SWITCHDESKTOP);
```

Listing B: TDesktopThread::Execute()

```
void __fastcall TDesktopThread::Execute()
{
HDESK hNewDesktop = ::CreateDesktop(
  FData.DesktopName.c_str(),
  NULL, NULL, FData.DesktopFlags,
  FData.DesiredAccess, NULL);

FDesktopCreated = ( NULL != hNewDesktop );
if( FOnDesktopCreated )
  Synchronize(DoOnDesktopCreated);

if(FThreadedEvent)
  {
  TDesktopGuard AGuardedDesktop(hNewDesktop);
  FThreadedEvent(this, FData, hNewDesktop);
  }
}
```

```
FSwitched = (
  ::SetThreadDesktop(FNewDesktop) &&
  ::SwitchDesktop(FNewDesktop) );
}
```

The constructor starts by obtaining a handle to the desktop associated with the current thread and storing it in `FOriginalDeskThread`. `FOriginalDeskInput` is then initialized with a handle to the desktop that is the current *input desktop*. Finally, `SetThreadDesktop()` is used to associate the newly created desktop with the current thread, and then `SwitchDesktop()` is used to switch to the new desktop.

Step 3: Launch another process

At this point, launching another executable will result in that process being associated with the calling thread and the new input desktop. This is exactly what happens when the user-provided event is called via this line in `TDesktopThread::Execute()`:

```
FThreadedEvent(this, FData, hNewDesktop);
```

Refer to `DoDesktopLaunchApplication()` in the main form of DESKTOPLAUNCH to see how the application is launched.

Step 4: Restore everything

When `FThreadedEvent` returns (which must not occur until the called process returns), `TDesktopGuard`'s

destructor is executed, resulting in everything being restored. Here's the destructor:

```
TDesktopGuard::~TDesktopGuard()
{
if( NULL != FOriginalDeskInput )
  ::SwitchDesktop(FOriginalDeskInput);

if( NULL != FOriginalDeskThread )
  ::SetThreadDesktop(
    FOriginalDeskThread);

if( NULL != FNewDesktop && FSwitched )
  ::CloseDesktop(FNewDesktop);
}
```

The destructor switches back to the previously known input desktop, then re-associates the thread with the original desktop, and then closes the secondary desktop.

It's important to note that if you leave any processes running when you close the desktop, the operation will fail. That is, the desktop will hang around until all processes close. Unless you're using interprocess communications to direct them to close, you'll need to use the Task Manager (or re-create a new desktop with the same name) to terminate the processes.

Using an in-proc approach

Everything covered so far has dealt with launching another application. This may not be suitable for your requirements, especially if you want to share information between the desktops.

When I first started playing with secondary desktops, I attempted to create a VCL form and place it on the new desktop. Unfortunately this does not work due to the close ties between VCL forms and `TApplication`. Don't despair, however, as I have thought of an alternative: *use a DLL*.

A VCL-based DLL has its own instance of `TApplication`. This means that you can create your form within a DLL and show it modally on the new desktop (it must be modal because the desktop will be closed as soon as the function-call returns). The subject of creating a DLL is beyond the scope of this article but will be covered in some near-future articles. If you need an immediate reference, then check out [5].

This month's download includes two projects that, together, show how to launch a form onto a new desktop via a DLL. The demo projects are called DSTUB.EXE and VCLKIOSK.DLL:

- DSTUB is used create a new desktop and then either launch an executable or call a function within a DLL depending on the file-name it is passed.
- VCLKIOSK exports a function with a known signature and is responsible for creating a form and showing it modally. This DLL performs no desktop operations since it is all handled by DSTUB.

Calling a function from the DLL

DSTUB is a console application requiring one or two command-line parameters. If one parameter is provided, then it must be the name of an executable or DLL. If two parameters are given, then the first parameter is the name of the desktop to create and the second must be the name of the executable or DLL. (If no desktop name is provided, then a default of "StubDesktop" is used. Refer to the source code for further details.)

After examining its command-line parameters, DSTUB checks to ensure the requested desktop does not already exist (refer to the source code for how this is done). Finally, DSTUB calls these two lines:

```
TStubHelper Stub( DesktopName, FileName );
Stub.Execute();   // waits for return
```

`TStubHelper` is a helper class whose `Execute()` method creates a new thread, creates the desktop, and then launches an application or calls a function in a DLL. Here's how the `TStubHelper::Execute()` method is defined:

```
int TStubHelper::Execute()
{
TOnDesktopThreadEvent ThreadEvent = NULL;
AnsiString Extension =
  ExtractFileExt(FFileName);

if( Extension.AnsiCompareIC(".dll") == 0 )
  ThreadEvent = OnDesktopDLLThread;
if( Extension.AnsiCompareIC(".exe") == 0 )
  ThreadEvent = OnDesktopEXEThread;

if( !ThreadEvent )
  FErrorCode =
    Nstuberrors::ERROR_INVALID_FILE_EXT;
```

```
if( Nstuberrors::ERROR_OK == FErrorCode )
  {
  std::auto_ptr<TDesktopThread>
    Thread(new TDesktopThread( true,
      ThreadEvent, FDesktopData ));
  if( Thread.get() )
    {
    Thread->OnDesktopCreated =
      OnDesktopCreated;
    Thread->FreeOnTerminate = false;
    Thread->Resume();
    Thread->WaitFor();
    }
  }
return FErrorCode;
}
```

This code creates the new `TDesktopThread` (`Thread`), and assigns its `ThreadEvent` member a different function depending on the extension of the provided filename (.DLL or .EXE). When the thread is resumed, the desktop is created (refer to **Listing B**), and then the appropriate `ThreadEvent` is called.

If a DLL is provided to DSTUB, the following code is called (with error-handling removed for brevity):

```
typedef void __stdcall
  (*TShowVCLForm)(void);

void __fastcall TStubHelper::
  OnDesktopDLLThread(
  TDesktopThread *Sender,
  const TDesktopData &AData,
  HDESK ADesktopHandle)
{
HINSTANCE hLib = ::LoadLibrary(
  FFileName.c_str() );

TShowVCLForm ShowVCLForm =
 (TShowVCLForm) ::GetProcAddress(
    (HMODULE)hLib, "ShowVCLForm" );
ShowVCLForm();

::FreeLibrary( hLib );
}
```

The `TShowVCLForm typedef` indicates that the DLL is required to export a function of the same signature. If you look at the source code for VCLKIOSK.DLL you will find `ShowVCLForm()` implemented as follows:

```
void __fastcall ShowVCLForm()
{
std::auto_ptr<TfrmKiosk>
  frm( new TfrmKiosk(
    static_cast<TComponent*>(NULL)) );
if( frm.get() )
  frm->ShowModal();
}
```

Pretty simple. As for what your form performs, that's up to you. The provided demo DLL creates a main form (`TfrmKiosk`) and a data module. The form presents a small database application to store some password information. It isn't very flashy but it does demonstrate that you can perform anything you like on the secondary desktop.

The easiest way to test your DLLs is to drag and drop them onto DSTUB.EXE. DSTUB will be passed the name of the file as its first parameter. In fact, as a test, drop a normal executable on top of DSTUB and watch it appear on a secondary desktop.

Name-mangling issues

As a quick and final note about the function exported from the DLL, if you export using the `__stdcall` calling convention your function name will not be mangled. For example:

```
extern "C"
{
  DLLAPI void __stdcall ShowVCLForm();
}
```

If you export using the `__fastcall` calling convention, then the name will be prefixed with an `@` symbol. This means your call to `GetProcAddress()` will become:

```
TShowVCLForm ShowVCLForm =
  (TShowVCLForm) ::GetProcAddress(
    (HMODULE)hLib, "@ShowVCLForm" );
```

A final warning

Working with secondary desktops is very difficult to debug when things go wrong. As it is very easy to lose control, especially when you cannot get back to the default desktop, I highly recommend you save all important work before testing your applications. I've had rare occasions in which I lost total control and needed to power down the computer without safely logging off.

To help prevent losing sight of the *Default* input desktop, this month's code includes a small watchdog application that performs 10-second-interval checks to

see if the input desktop is empty. If it is, then it will switch you back to the default desktop. This did not save me from all disasters, but it did save me from many failures.

Also, the LAUNCHDESKTOP demo application provides the ability to create an "Advanced Desktop" where you specify the exact flags you want to use. I have coded this section based on the information at MSDN, but I have not performed extensive testing. Use at your own risk, but please let me know if you find any issues or bugs so that I can pass the information on to other readers.

Conclusions

Secondary desktops provide you the ability to launch applications or show modal forms on an alternate desktop that does not allow the user to access the windows shell (unless you allow them to execute EXPLORER.EXE which will automatically create the shell). This article described the steps required to perform this task and is accompanied by two sample applications (and a DLL) that demonstrates how to put the pieces together.

Contact Malcolm at **msmith@bcbjournal.com**.

Acknowledgements

The code for the watchdog application provided with this month's download is a translation from code I found after reading "Private Desktops & Windows XP," published by Dr. Dobbs Journal January 2003 [8].

References

1. Information about atom tables can be found at http://tinyurl.com/4zx72
2. http://tinyurl.com/8m3tq
3. I could not find this documented on MSDN but I observed it on my Windows 2000 and Windows XP machines. MSDN mentions the third desktop as being called *screen-saver* which is created when a secured screen saver is activated, while non-secured screen savers run on WinSta0\default. Since the screen saver is not active while the user is interactive with the desktop, I did not check further to confirm *screen-saver* was present.
4. http://tinyurl.com/7fkn3
5. http://tinyurl.com/awjlz
6. http://tinyurl.com/a75ex
7. http://tinyurl.com/aef4p
8. http://tinyurl.com/bposc

Mouse Gestures

By Damon Chandler

For many Windows applications, the mouse serves as the primary input device: We use it to press on-screen buttons, select menus, navigate to different windows, and select controls within a window, just to name a few uses. Granted, these operations can also be achieved (sometimes, more quickly) via the keyboard; but, for many tasks, working with the mouse is often more convenient and more intuitive. For example, imagine browsing the Web using only your keyboard.

In fact, browsing the Web is a good example of a situation in which using the mouse is sometimes more convenient and sometimes more wearisome than using the keyboard. On the one hand, navigating to different links within a web page is—by far—much easier by using the mouse. On the other hand (and when this hand is free), I often find it more convenient to press the Backspace key rather than moving the mouse to the top of the browser to press the "Back" button. And, what about copying and pasting text/images from a web page into a document? Sometimes I find this task to be more easily accomplished via the keyboard; other times, I prefer OLE-based drag-and-drop [1] (as long as I don't have to drag the mouse too far).

It seems that the degree of convenience provided by the mouse is inversely related to the distance that the mouse cursor has to travel to perform a task. This is especially true for commonly performed tasks such as scrolling. It's not difficult to *use* a window's scrollbars; the tedious part is moving the mouse cursor all the way to the edge of a window to get *to* the scrollbars. Indeed, modern mice have addressed this issue by including a scroll wheel.

Hardware-based modifications aren't the only solution. One can also use short, swift mouse movements—called *mouse gestures*—to quickly perform common tasks [2]. In a sense, any task-based mouse movement can be considered a mouse gesture (e.g., dragging a file in Explorer). Here, when I say a "mouse gesture," I'm referring to a mouse movement that's short, quick, easily performed, and—most importantly—doesn't require interaction with a particular GUI element. So, in this article, I'll show you how to add basic mouse-gesture support to your applications.

Overview

As it turns out, creating a C++Builder component that supports mouse gestures is both extremely simple and somewhat complex. The three main steps involved in this process are:

1. *Recording*—The first step involves capturing and storing the coordinates of the mouse cursor as the end-user makes the gesture.
2. *Recognizing*—The next step is to compare the gesture from Step 1 with a list of known gestures.
3. *Responding*—The final step is to performing an action in response to the recognized gesture.

Steps 1 and 3 (recording and responding) are easily handled by using the `TApplicationEvents` and `TBasicAction` classes, respectively. Specifically, we can use the `TApplicationEvents::OnMessage` event to access the mouse-related messages needed to track and record the end-user's mouse movements. And, we can use the `TBasicAction::Execute()` method to invoke a recognized gesture's corresponding action.

Step 2 (recognition), on the other hand, is not quite as easy, simply because humans and computer mice aren't perfect—there will always be slight variations in the mouse movements that comprise a user-input mouse gesture. Accordingly, we'll need to create a recognition system that provides a decent trade-off between *sensitivity* (i.e., the ability to correctly recognize an intended gesture) and *specificity* (the ability to correctly ignore unintended gestures) [3]. Furthermore, the recognition process has to be *fast*: There's little advantage to using mouse gestures if the recognition process is so slow that the end-user can more quickly invoke the desired action by using traditional input methods (e.g., via menu commands or keystrokes).

The following section describes the recognition system that's used in this month's code. Later, I'll present three VCL classes—`TMouseGesture`, `TMouseGestures`, and `TMouseGestureManager`—and I'll show how these classes perform the steps of recording, recognizing, and responding to mouse gestures.

Recognizing gestures

Suppose that we're given a list of (*x*,*y*) coordinates corresponding to a mouse gesture input by the end-user; each (*x*,*y*) pair in the list denotes the position of the mouse cursor relative to the top-left corner of the screen. Also suppose that we have a collection of predefined gestures, each with its own unique list of (*x*,*y*) coordinates. The goal of gesture recognition is to determine which gesture in the predefined collection, if any, corresponds to the gesture input by the end-user.

For example, if the input gesture is as shown in the left-hand side of **Figure A**, and the predefined gestures are as shown in right-hand side of **Figure A**, we'd expect the input gesture to be recognized as the predefined "circle" gesture.

Invariant recognition

Of course, for a human, recognizing the correct gesture is trivial; our visual system has an extraordinary ability to perform recognition of objects, even of objects that have undergone various transformations (e.g., a change in size, or small perturbations in shape). On the other hand, getting a computer to perform this form of invariant recognition is not nearly as trivial.

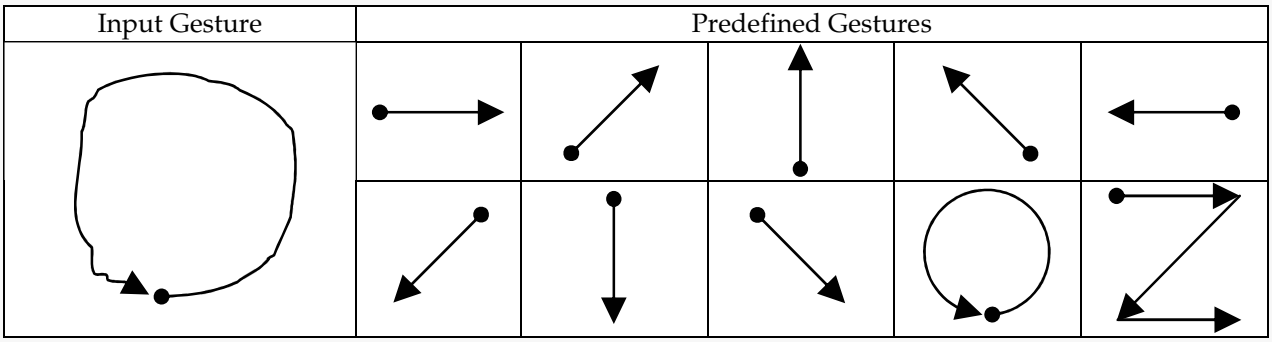
There are various techniques of gesture recognition, the most common of which uses a grid-based method that operates by quantizing the (*x*,*y*) coordinates to a scaled and indexed grid, and then performing the matching with respect to the grid indices [4][5]. Other, more computationally intensive approaches involve statistical learning algorithms (e.g., neural nets) [6].

Here, I've taken a different approach: We account for size differences by first converting the (*x*,*y*) coordinates to polar angles [7], then interpolating the list of angles to a common length. And, we account for small perturbations in shape by performing a quasi-winner-take-all match between lists of angles.

Converting to polar coordinates

Recall that an (*x*,*y*) coordinate can be viewed as a vector in a two-dimensional vector space as shown in **Figure B**. The (*x*,*y*) coordinates specify the vector's

Figure A



Example input gesture (left) and a set a predefined gestures (right). Finding the best match is trivial for a human, but it's not so easily accomplished via code.

head; and, for simplicity, assume the vector's tail is located at the origin (0,0). Alternatively, we can uniquely specify the vector by its length (denoted by symbol *r*), and by its angle relative to the *x*-axis (denoted by symbol θ); see **Figure B**.

In other words, we can transform the (*x*,*y*) rectangular coordinates to (r,θ) *polar* coordinates via

$$r = \sqrt{x^2 + y^2}$$
$$\theta = \arctan(y / x)$$

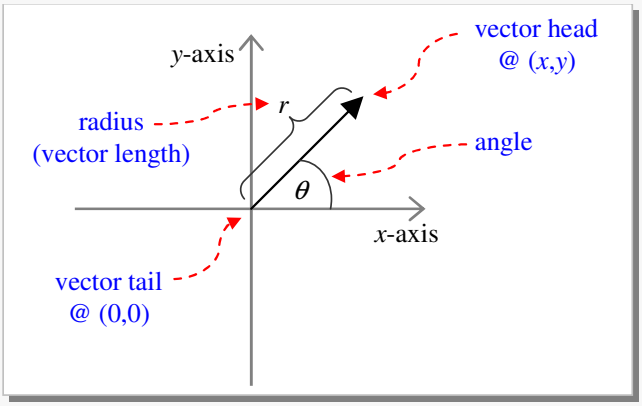
Note that when $x = 0$, θ is either +90 degrees (+π/2 radians) or -90 degrees (-π/2 radians), depending on the sign of *y*; that is, the vector is either pointing straight up or straight down, depending on whether *y* is positive or negative.

When working with a list of (*x*,*y*) coordinates, we can assume (again, for simplicity) that the vectors are connected in a "head-to-tail" fashion—i.e., the head of one vector is also the tail of the next vector. As an example, consider the gesture shown in **Figure C**. This gesture consists of five (*x*,*y*) coordinates, and thus four vectors. The angles of these four vectors, starting with the bottom-most vector, are: 180 degrees (π radians), 90 degrees (π/2 radians), 45 degrees (π/4 radians), and 135 degrees (3π/4 radians). Furthermore, because we're interested in the shape of the gesture, regardless of its size, we can ignore the vector lengths (and assume that they're all $r = 1$).

Here's a function that will convert a list of (*x*,*y*) coordinates to a list of angles:

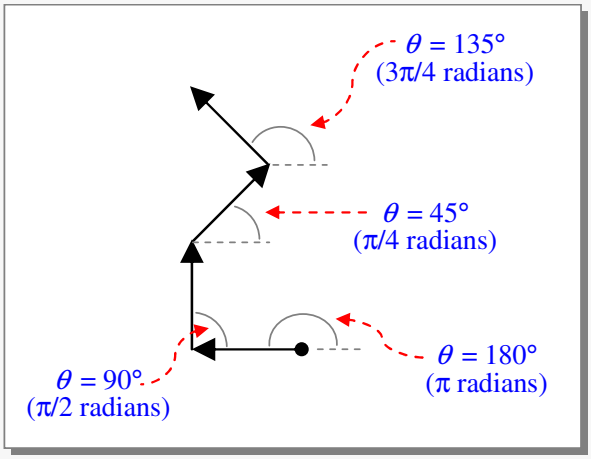
```
typedef std::vector<float> TAngles;
typedef std::vector<TPoint> TPoints;

void PointsToAngles(
  TPoints const& Points,
  TAngles& Angles)
{
  int const num_angles =
    Points.size() - 1;
  if (num_angles < 1)
  {
    throw Exception("Invalid points");
  }

  Angles.clear();
  Angles.reserve(num_angles);
  for (int idx = 0; idx < num_angles;
    ++idx)
  {
    TPoint const& P0 = Points[idx];
    TPoint const& P1 = Points[idx + 1];

    float const dy = P1.y - P0.y;
    float const dx = P1.x - P0.x;
    if (std::abs(dy) > 0 ||
      std::abs(dx) > 0)
    {
      Angles.push_back(
        std::atan2(-dy, dx)
        );
    }
  }
}
```

Figure B



Vector-space representation of rectangular and polar coordinates.

Figure C



Polar angles of an example gesture.

Note that the `atan2()` function (as opposed to the `atan()` function) correctly handles the case in which the second parameter (`dx`) is zero. Also note that I've passed `-dy` as the first parameter to `atan2()` because in screen coordinates, the upward direction corresponds to a decrease in the *y* coordinate.

Interpolating to a common length

After converting the list of (*x*,*y*) coordinates to a list of angles, we can compare the input gesture with each predefined gesture (each of which also has been specified in terms of its angles). As I'll discuss shortly, the comparison method is extremely simple: We just scan the list of predefined gestures, and locate the one whose list of angles most closely matches the input gesture's list of angles (where "closest match" is defined as the minimum sum-squared error between lists of angles).

However, before we can compute the sum-squared error between two lists of angles, we need to address the issue of length. Namely, because the input gesture is provided by the end-user, the length of the input gesture's list of angles might be as small as a single angle, or it might be as large as, say, a length-1000 list of angles. How should these angles be compared to the predefined gestures which consist of, say, length-30 lists? We'll need to ensure that all lists of angles have a common length.

Here's a function that will resize a `std::vector` of angles (`TAngles`) to a specified size using nearest-neighbor interpolation:

```
void ResizeAngles(
  TAngles const& Angles,
  TAngles& ResizedAngles,
  int dst_len)
{
  // check the existing lengths
  int const src_len = Angles.size();
  if (src_len == 0)
  {
    throw Exception("Invalid angles");
  }
  else if (src_len == dst_len)
  {
    ResizedAngles = Angles;
    return;
  }

  // compute the resizing ratio
  float const ratio_len =
    static_cast<float>(src_len) /
    static_cast<float>(dst_len);

  // if the source and destination
  // vectors refer to the same memory,
  // we need to use a temporary vector
  if (&Angles == &ResizedAngles)
  {
    TAngles TempResizedAngles(dst_len);
    for (int dst_idx = 0;
      dst_idx < dst_len; ++dst_idx)
    {
      TempResizedAngles[dst_idx] =
        Angles.at(dst_idx * ratio_len);
    }
    ResizedAngles = TempResizedAngles;
  }
  // source and destination vectors use
  // separate memory, so just use them
  else
  {
    ResizedAngles.resize(dst_len);
    for (int dst_idx = 0;
      dst_idx < dst_len; ++dst_idx)
    {
      ResizedAngles[dst_idx] =
        Angles.at(dst_idx * ratio_len);
    }
  }
}
```

Winner-take-all matching

After the lists of angles for all gestures have been resized to a common length, the final step in the recognition process is to locate the closest match. As I mentioned, the closest match is defined in the sum-squared-error sense. Here's a function that will compute the sum-squared error between two `std::vectors` of angles:

```
float AnglesSSE(
  TAngles const& Angles1,
  TAngles const& Angles2,
  float current_min_sse)
{
  float sse = 0;
  int const num_angles = std::min(
    Angles1.size(), Angles2.size()
    );
  for (int idx = 0; idx < num_angles;
    ++idx)
  {
    float err = std::abs(
      Angles1[idx] - Angles2[idx]
      );
    // ensure that the absolute error
    // between two angles never exceeds
    // 180 degrees (pi radians)
    if (err > M_PI)
    {
      err = M_2PI - err;
```

```
    }
    sse += (err * err);

    // if the running sum-squared error
    // at this point is already greater
    // than the current minimum sum-
    // squared error (as specified by
    // the current_min_sse parameter),
    // there's no need to keep computing
    if (sse > current_min_sse)
    {
      return 1E10; // error punt
    }
  }
  return sse;
}
```

And, here's some pseudo-code which demonstrates how the `AnglesSSE()` function is used in the matching procedure:

```
min_sse = 1E10;
for each predefined gesture
{
  sse = AnglesSSE(
    InputAngles, PredefinedAngles,
    min_sse
    );
  if (sse < min_sse && sse <= tolerance)
  {
    min_sse = sse;
  }
}
return (
  predefined gesture with min_sse ||
  no gesture if min_sse is 1E10
  )
```

This search procedure is similar to a winner-take-all scheme [8] in which the input gesture is mapped to (or "recognized as") the predefined gesture whose angles most closely match the input gesture's angles. However, this is not a true winner-take-all approach because the `if(sse<tolerance)` check in the above code allows for the possibility that no one wins. Such a scheme provides a reasonable tradeoff between sensitivity (true-positive rate) and specificity (false-positive rate).

Later, I'll show exactly how the `AnglesSSE()` function is used to find the closest matching gesture (see "Finding a match," later in this article). But, first, let me introduce the three mouse-gesture-related VCL classes: `TMouseGesture`, `TMouseGestures`, and `TMouseGestureManager`.

Listing A: TMouseGesture declaration

```
class PACKAGE TMouseGesture :
  public TCollectionItem
{
__published:
  __property TBasicAction* Action =
    {read=GetAction, write=SetAction};
  __property float Tolerance =
    {read=Tolerance_, write=Tolerance_};

public:
  __property TMouseGestureManager* Manager =
    {read=GetManager};

public:
  __fastcall TMouseGesture(
    TCollection* Collection);

public:
  TAngles& Angles() { return Angles_; }

public:
  virtual float Error(TAngles const& Angles,
    float current_min_err);
  virtual void Draw(TCanvas& Canvas,
    int cx, int cy, int padding = 4,
    int arrow_size = 8);

private:
  TBasicAction* __fastcall GetAction();
  void __fastcall SetAction(
    TBasicAction* NewAction);
  TMouseGestureManager* __fastcall GetManager();

private:
  std::auto_ptr<TActionLink> ActionLink_;
  float Tolerance_;
  TAngles Angles_;
};
```

TMouseGesture

The `TMouseGesture` class is a `TCollectionItem` descendant that's designed to represent a predefined gesture and its associated action; **Listing A** shows the class declaration. In particular, notice from the `private` section of the class declaration that the `TMouseGesture` class has three private members:

- `ActionLink_`, which is a `TActionLink` that supports the `TMouseGesture::Action` property (see [9] for information on actions and action links);

- `Tolerance_`, which is a floating-point value that specifies the maximum sum-squared error allowed for the gesture to be recognized; this member is the back-end to the `Tolerance` property.
- `Angles_`, which is a `std::vector` of the polar angles that defines the gesture's shape.

Here's the code for the `TMouseGesture` constructor, which initializes the `ActionLink_` and `Tolerance_` members:

```
__fastcall TMouseGesture::TMouseGesture(
  TCollection* Collection) :
    TCollectionItem(Collection),
    ActionLink_(new TActionLink(this)),
    Tolerance_(25)
{
}
```

Again, `TMouseGesture` is a `TCollectionItem` descendant, which is designed to be contained within a `TOwnedCollection`. A pointer to this `TOwnedCollection` (specifically, `TMouseGestures`, described shortly) is passed to the class constructor. In turn, the `TOwnedCollection` will be contained within a `TComponent` (`TMouseGestureManager`), a pointer to which is returned via the `TMouseGesture::Manager` property:

```
inline TMouseGestureManager* __fastcall
  TMouseGesture::GetManager()
{
  return static_cast
    <TMouseGestureManager*>(
      Collection->Owner()
      );
}
```

The action and action link

Notice from **Listing A** that the `TMouseGesture` class contains an `Action` property. As I mentioned, this action is maintained via the private `ActionLink_` member:

```
TBasicAction* __fastcall
  TMouseGesture::GetAction()
{
  return ActionLink_->Action;
}
```

```
void __fastcall TMouseGesture::
  SetAction(TBasicAction* NewAction)
{
  ActionLink_->Action = NewAction;
  if (NewAction != NULL)
  {
    NewAction->
      FreeNotification(Manager);
  }
}
```

Note that the call to `FreeNotification()` is required so that we receive a notification when the action is destroyed; this allows us to update the `Action` property accordingly. However, `FreeNotification()` requires a `TComponent*` as its single parameter, meaning that only `TComponent` descendants are notified of the action's destruction. Because `TMouseGesture` is a `TCollectionItem` (and not a `TComponent`), I've passed `Manager` (rather than the `this` pointer) to `FreeNotification()`. Later, I'll show you how the `TMouseGestureManager` class responds to the "action destroyed" notification.

The angles

Of course, the most important element of the `TMouseGesture` class is its angles which define the underlying gesture. As shown in **Listing A**, these angles are retrieved/specified via the `Angles()` method, which returns a (non-`const`) reference to the `std::vector` vector of angles (`Angles_`).

I've also added a method called `Error()`, which is designed to return the error between a specified list of angles and the gesture's list of angles:

```
float TMouseGesture::Error(
  TAngles const& Angles,
  float current_min_err)
{
  return AnglesSSE(
    Angles_, Angles, current_min_err
    );
}
```

Here, the `Error()` method simply calls the previously described `AnglesSSE()` function. Note, however, that the `Error()` method is `virtual`, so if you later choose to use a different error metric, you'd just override the `Error()` method.

In addition, the TMouseGesture class contains a method called Draw() for drawing the gesture as a series of arrows to a TCanvas. I'll refer you to this month's source code for the definition of that method.

TMouseGestures

Now that we have a class that represents a single mouse gesture, let's look at the container class—TMouseGestures—which is designed to hold these gestures.

The TMouseGestures class, whose declaration is provided in **Listing B**, is a TOwnedCollection descendant. As I mentioned last month [10], TOwnedCollection is a TCollection that maintains information about its owner (in this case, TMouseGestureManager). A pointer to this owner is specified via the class constructor:

```
__fastcall TMouseGestures::
  TMouseGestures(TPersistent* AOwner)
    : TOwnedCollection(AOwner,
      __classid(TMouseGesture)),
    GestureLength_(16)
{
}
```

Note that in addition to passing AOwner to the TOwnedCollection constructor, we also pass information about the type of the to-be-contained items (TMouseGesture).

GestureLength

In addition to storing multiple TMouseGesture objects, the TMouseGestures class is also responsible for resizing each gesture's list of angles to a common length. This common length is specified via the GestureLength property (and the private GestureLength_ member), and it is used in the NormalizeGestures() method, like so:

```
void TMouseGestures::NormalizeGestures()
{
  int const num_gestures = Count;
  for (int idx = 0; idx < num_gestures;
    ++idx)
  {
    TAngles& Angles =
      Gesture[idx]->Angles();
    ResizeAngles(
      Angles, Angles, GestureLength_
      );
  }
}
```

Here, we simply use the ResizeAngles() function that I presented earlier to resize each gesture's list of angles to length GestureLength using nearest-neighbor interpolation.

Listing B: TMouseGestures declaration

```
class PACKAGE TMouseGestures :
  public TOwnedCollection
{
public:
  __property int GestureLength =
    {read=GestureLength_,
     write=GestureLength_};
  __property TMouseGesture* Gesture[int Index] =
    {read=GetItem, write=SetItem};

public:
  __fastcall TMouseGestures(TPersistent* AOwner);

public:
  virtual void NormalizeGestures();
  virtual TMouseGesture* FindGesture(
    TAngles const& Angles);
  TMouseGesture* __fastcall Add()
    {
      return static_cast<TMouseGesture*>(
        TOwnedCollection::Add()
        );
    }

protected:
  TMouseGesture* __fastcall GetItem(int Index)
    {
      return static_cast<TMouseGesture*>(
        TOwnedCollection::GetItem(Index)
        );
    }
  void __fastcall SetItem(int Index,
    TMouseGesture* AGesture)
    {
      TOwnedCollection::SetItem(Index, AGesture);
    }

private:
  int GestureLength_;
};
```

Finding a match

The final duty of the TMouseGestures class is to perform the search for the closest-matching gesture, given a (user-input) list of angles. Here's the code for that:

```
TMouseGesture* TMouseGestures::
  FindGesture(TAngles const& Angles)
{
  // find the index of the gesture
  // whose angles are closest to the
  // supplied angles (and also within
  // the gesture's tolerable error)
  int min_idx = -1;
  float min_sse = 1E10;
  int const num_gestures = Count;
  for (int idx = 0; idx < num_gestures;
    ++idx)
  {
    float const sse = Gesture[idx]->
      Error(Angles, min_sse);
    if (sse < min_sse &&
      sse <= Gesture[idx]->Tolerance)
    {
      min_sse = sse;
      min_idx = idx;
    }
  }

  // return either a pointer to the
  // closest gesture or NULL if no
  // gesture was close enough
  if (min_idx >= 0)
  {
    return Gesture[min_idx];
  }
  return NULL;
}
```

TMouseGestureManager

The last component of the mouse-gestures framework is the TMouseGestureManager class, which is declared in **Listing C**.

As its name suggests, TMouseGestureManager is responsible for managing multiple TMouseGesture objects. Specifically, the TMouseGestureManager class—with help from TMouseGesture and TMouseGestures—performs the recording, recognizing, and responding steps mentioned earlier.

Here's an overview of TMouseGestureManager's properties and events:

Listing C: TMouseGestureManager declaration

```
typedef void __fastcall
  (__closure *TMouseGestureEvent)(
    TObject* Sender, TMouseGesture* Gesture,
    const TAngles& Angles, bool& GesHandled);

class PACKAGE TMouseGestureManager :
  public TComponent
{
__published:
  __property bool Active =
    {read=Active_, write=SetActive,
     default=true};
  __property bool AutoExecute =
    {read=AutoExecute_, write=AutoExecute_,
     default=true};
  __property TMouseGestures* Items =
    {read=GetItems, write=SetItems};
  __property TMouseGestureEvent OnGesture =
    {read=OnGesture_, write=OnGesture_};

public:
  __fastcall TMouseGestureManager(
    TComponent* Owner);

protected: // inherited
  virtual void __fastcall Notification(
    TComponent* Component, TOperation Operation);

protected: // introduced
  virtual void __fastcall DoAppMessage(
    MSG& Msg, bool& MsgHandled);
  virtual void DoRecognize(
    TAngles const& Angles);

private:
  void __fastcall SetActive(bool NewActive);
  TMouseGestures* __fastcall GetItems()
    {
      return Gestures_.get();
    }
  void __fastcall SetItems(
    TMouseGestures* NewGestures)
    {
      Gestures_->Assign(NewGestures);
    }

private: // property-related members
  bool Active_;
  bool AutoExecute_;
  std::auto_ptr<TMouseGestures> Gestures_;
  TMouseGestureEvent OnGesture_;

private: // recording-related members
  bool Recording_;
  std::vector<TPoint> mouse_coords_;
  TApplicationEvents* AppEvents_;
};
```

- The `Active` property, which specifies whether or not gestures should be recorded and recognized.
- The `AutoExecute` property, which specifies whether or not recognized gestures should be responded to (i.e., whether or not the corresponding action of a recognized gesture should be invoked upon recognition).
- The `Items` property, which provides access to the individual gestures.
- The `OnGesture` event, which is invoked in response to both successful and unsuccessful recognitions.

And, here's the constructor of the `TMouseGestureManager` class:

```
__fastcall TMouseGestureManager::
  TMouseGestureManager(
    TComponent* AOwner) :
    TComponent(AOwner),
    AutoExecute_(true),
    Gestures_(new TMouseGestures(this)),
    AppEvents_(new
      TApplicationEvents(this))
{
  SetActive(true);
}
```

As described next, the `AppEvents_` member is a `TApplicationEvents` instance whose `OnMessage` event provides access to the (application-wide) mouse messages needed for recording user-input gestures. The `SetActive()` method, which is defined as follows, simply toggles the `OnMessage` property:

```
void __fastcall TMouseGestureManager::
  SetActive(bool NewActive)
{
  if (NewActive)
  {
    AppEvents_->OnMessage =
      DoAppMessage;
  }
  else
  {
    AppEvents_->OnMessage = NULL;
  }
  Active_ = NewActive;
}
```

Recording

Before we can recognize a mouse gesture, we first need to record the (*x*,*y*) coordinates input by the end-user. Specifically, the end-user initiates a "gesture recording" session by (1) pressing and holding down the right mouse button, then (2) moving the mouse to input the gesture, and then (3) releasing the right mouse button [11].

Here, I've used a `TApplicationEvents` object—and in particular, its `OnMessage` event—to gain access to the mouse messages corresponding to these mouse-related actions. By using the `OnMessage` event (as opposed to the `OnMouseDown`, `OnMouseMove`, and `OnMouseUp` events of a particular window), we're able to record gestures anywhere in the application.

If the `Active` property is `true`, the `OnMessage` event is assigned the following event handler:

```
void __fastcall
  TMouseGestureManager::DoAppMessage(
    MSG& Msg, bool& MsgHandled)
{
  switch (Msg.message)
  {
    case WM_RBUTTONDOWN:
    {
      // capture mouse messages
      SetCapture(Msg.hwnd);
      // set the mouse-tracking flag
      Recording_ = true;
      break;
    }
    case WM_MOUSEMOVE:
    {
      // if we're recording...
      if (Recording_ &&
        (Msg.wParam & MK_RBUTTON))
      {
        // grab the cursor location
        TPoint PMouse(
          LOWORD(Msg.lParam),
          HIWORD(Msg.lParam)
          );
        // translate to screen coords
        ClientToScreen(
          Msg.hwnd, &PMouse
          );
        // store the coordinates
        mouse_coords_.push_back(PMouse);
        MsgHandled = true;
      }
      break;
    }
```

```
    case WM_RBUTTONUP:
    {
      // release mouse capture
      ReleaseCapture();

      // if the recorded gesture is of
      // sufficient length...
      if (Recording_ &&
        mouse_coords_.size() > 6)
      {
        Recording_ = false;
        // convert to angles
        TAngles Angles;
        PointsToAngles(
          mouse_coords_, Angles
          );
        // attempt recognition
        DoRecognize(Angles);
        MsgHandled = true;
      }

      // clean up
      mouse_coords_.clear();
      break;
    }
  }
}
```

This code: (1) stores the (*x*,*y*) coordinates of the recorded gestures in the private `mouse_coords_` member (a `std::vector`); then (2) converts the (*x*,*y*) coordinates to a corresponding std::vector of polar angles via the previously mentioned `PointsToAngles()` function; and then (3) passes this std::vector of angles on to the `DoRecognize()` method.

Recognizing and responding

The `DoRecognize()` method is responsible for recognizing the user-input gesture; here's the code:

```
void TMouseGestureManager::
  DoRecognize(TAngles const& Angles)
{
  // resize the user-input angles
  // to the common length
  TAngles ResAngles;
  ResizeAngles(Angles, ResAngles,
    Gestures_->GestureLength);

  // find the matching gesture (if any)
  TMouseGesture* const Gesture =
    Gestures_->FindGesture(ResAngles);

  // invoke the OnGesture event handler
  bool GesHandled = false;
  if (OnGesture_ != NULL)
  {
    OnGesture_(this, Gesture,
      ResAngles, GesHandled);
  }

  // if required, invoke the gesture's
  // associated action
  if (!GesHandled && AutoExecute_ &&
    Gesture != NULL &&
    Gesture->Action != NULL)
  {
    Gesture->Action->Execute();
  }
}
```

Here, we simply punt to the `TMouseGestures::FindGesture()` method to find the matching gesture in the predefined collection. Note that this call to `FindGesture()` will return `NULL` in the case of no match.

Following the recognition attempt, the `OnGesture` event is invoked, which provides an opportunity to inform the end-user of the recognition results (e.g., you could display a sound or draw the gesture). It's especially important to inform the end-user of a unsuccessful recognition attempt—you certainly don't want the end-user waiting for an action that'll never happen.

The `OnGesture` event also contains a Boolean reference-type parameter, `GesHandled`, which allows you to selectively filter out certain gestures from within the event handler. Specifically, if the `TMouseGestureManager::AutoExecute` property is `true`, and the gesture was recognized, the gesture's associated action is invoked via the `TBasicAction::Execute()` method. If you don't want this action to be invoked for a *particular* gesture, you can set the `GesHandled` parameter to `true` from within a handler for the `OnGesture` event. (And, if you don't want *any* gesture's action to be invoked, you can simply set `AutoExecute` to `false`.)

The Notification() method

Earlier, when describing the `TMouseGesture::SetAction()` method (see "The action and action link"), I mentioned that the `TMouseGestureManager` component is notified of the destruction of each gesture's corresponding `TBasicAction`. Recall that this setup is required because the `TMouseGesture` class is not a `TComponent` descendant and therefore cannot receive notification of destruction of its action. `TMouseGestureManager`, on the other hand, *is* a `TComponent` de-

scendant, and has therefore been delegated the task of updating each gesture's `Action` property:

```
void __fastcall TMouseGestureManager::
  Notification(TComponent* Component,
    TOperation Operation)
{
  // if an action is being destroyed
  if (Operation == opRemove &&
    dynamic_cast<TBasicAction*>
      (Component) != NULL)
  {
    // scan the list of gestures to see
    // if one of their actions is the
    // deleted action (and then nullify
    // its Action property)
    int const num_gestures =
      Items->Count;
    for (int idx = 0;
      idx < num_gestures; ++idx)
    {
      if (Component ==
        Items->Gesture[idx]->Action)
      {
        Items->Gesture[idx]->
          Action = NULL;
      }
    }
  }

  // call the inherited version
  TComponent::
    Notification(Component, Operation);
}
```

Using and extending the code

This month's code includes a sample application that demonstrates how to use the `TMouseGesture`, `TMouseGestures`, and `TMouseGestureManager` classes. Using the `TMouseGestureManager` class is extremely simple: you just drop a `TMouseGestureManager` instance on your form at design-time, and then add your predefined `TMouseGesture` objects using the standard Collection Editor.

Unfortunately, there's a catch: you must manually (i.e., via code) specify each predefined gesture's list of angles. For example, here's the constructor of the form used in the demo project, which defines the 10 predefined gestures shown in **Figure A**:

```
__fastcall TForm1::TForm1(
  TComponent* Owner) : TForm(Owner)
{
  for (int idx = 0; idx < 10; ++idx)
  {
    // grab a pointer to the gesture
    TMouseGesture* Gesture =
      MouseGestureManager1->
        Items->Gesture[idx];
    // grab a reference to its angles
    TAngles& Angles = Gesture->Angles();

    // define straight-line gestures...
    if (idx < 8)
    {
      float angle;
      switch (idx)
      {
        // 0 degrees
        case 0: angle = 0.0f;
          break;
        // 45 degrees
        case 1: angle = M_PI/4.0f;
          break;
        // 90 degrees
        case 2: angle = M_PI/2.0f;
          break;
        // 135 degrees
        case 3: angle = 3.0f*M_PI/4.0f;
          break;
        // 180 degrees
        case 4: angle = M_PI;
          break;
        // -135 degrees (225 degrees)
        case 5: angle = -3.0f*M_PI/4.0f;
          break;
        // -90 degrees (270 degrees)
        case 6: angle = -M_PI/2.0f;
          break;
        // -45 degrees (315 degrees)
        case 7: angle = -M_PI/4.0f;
          break;
      }

      for (int pos = 0; pos < 2; ++pos)
      {
        Angles.push_back(angle);
      }
    }

    // define a counter-clockwise circle
    // gesture starting at 6 o'clock...
    else if (idx == 8)
    {
      for (float angle = 0;
        angle < M_PI; angle += M_PI/50.0f)
      {
        Angles.push_back(angle);
      }
      for (float angle = -M_PI;
        angle < 0; angle += M_PI/50.0f)
      {
        Angles.push_back(angle);
      }
    }
```

```
    // define a Z-shaped gesture...
    else // (idx == 9)
    {
      Angles.push_back(0.0f);
      Angles.push_back(0.0f);
      Angles.push_back(-3.0f*M_PI/4.0f);
      Angles.push_back(-3.0f*M_PI/4.0f);
      Angles.push_back(-3.0f*M_PI/4.0f);
      Angles.push_back(0.0f);
      Angles.push_back(0.0f);
    }
  }

  // common length of 21
  MouseGestureManager1->
    Items->GestureLength = 21;
  // resize all gestures
  MouseGestureManager1->
    Items->NormalizeGestures();
}
```

Figure D shows a screenshot of the application upon successful recognition of the "circle" gesture.

For these relatively simply shapes, specifying the angles via code isn't too difficult; however, this process can quickly become tedious for more complex shapes. This is certainly an area which can use some improvement. In fact, it wouldn't be too difficult to write a utility application in BCB which allows you to graphically define the gestures. A similar approach might also be used to allow the end-user to specify custom, user-tuned gestures.

Conclusions

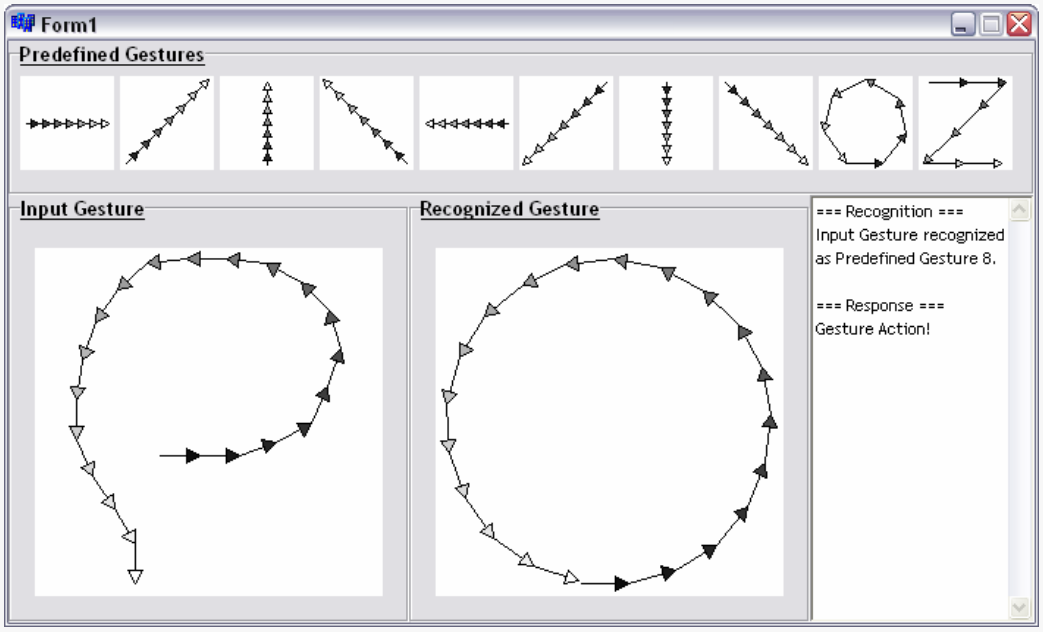
In this article, I described how to record, recognize, and respond to mouse gestures. In particular, I presented a basic approach to gesture recognition which operates based on polar angles and quasi-winner-take-all matching. I encourage you to also take at look at [4], [5] and [6] for other, more sophisticated approaches to recognition.

Contact Damon at **editor@bcbjournal.com**.

References

1. D. Chandler, "OLE Drag and Drop," *C++Builder Dev. Journal*, **6** (11), 2002.
2. http://en.wikipedia.org/wiki/Mouse_gesture
3. http://tinyurl.com/8x394
4. http://www.etla.net/libstroke
5. http://xstroke.org
6. http://tinyurl.com/bsost
7. http://tinyurl.com/cabad
8. Winner-take-all matching is often referred to as *vector quantization*; see http://tinyurl.com/b8mtz
9. D. Chandler, "Using Actions and Action Lists, Part I," *C++Builder Dev. Journal*, **6** (9), 2002.
10. D. Chandler, "More Run-time Moving and Resizing, Part II," *C++Builder Dev. Journal*, **9** (5), 2005.
11. Note that there are other ways to initiate a gesture-recording session, e.g., by holding down the Control key instead of the right mouse button; this I leave as an exercise for the reader.

Figure D



Screenshot of this month's demo application.

This Month's Developer's Poll

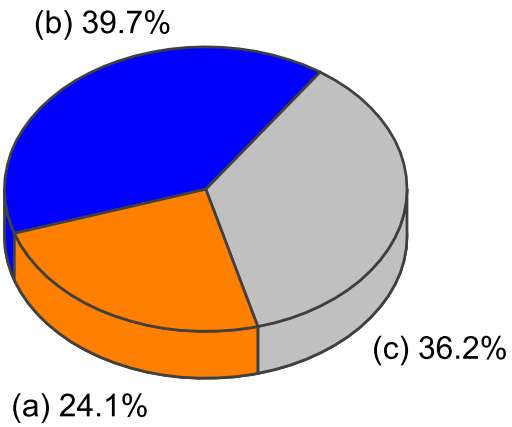
Each month, the Journal's Developer's Poll allows you to compare your opinions to those of other members of the C++Builder Developer's Journal community. The results of this month's poll will be published in next month's issue, and we will forward the results to the C++ folks at Borland.

Last month's poll question was:

How often do you refer to the VCL source code while programming?

a. Very frequently.
b. Occasionally.
c. Very rarely or never.

The results are shown in the following chart:



We thank the 58 readers who voted.

This month's poll question is:

What feature of C++Builder do you find most attractive for facilitating application development?

a. The visual designer/VCL.
b. The overall IDE.
c. The compiler.
d. Other (please specify).

Cast your vote online at
http://polls.bcbjournal.com

This Month's Contributors

Curtis Krauskopf

Curtis Krauskopf is a software engineer and the principle of The Database Managers (www.decompile.com). He has been writing code professionally for over 20 years. Curtis has a bachelor's degree in Computer and Electrical Engineering from Purdue University. His prior projects include multiple web e-commerce applications, decompilers for the DataFlex language, aircraft simulators, an automated Y2K conversion program for over 3,000,000 compiled DataFlex programs, and inventory control projects. Curtis has spoken at many domestic and international DataFlex developer conferences and has been published in FlexLines Online, JavaPro Magazine and C/C++ Users Journal. Curtis can be contacted at **curtis@decompile.com**.

Don Doerres

Don Doerres is a long-time embedded systems engineer with expertise in both hardware and software. His day job is Chief Software Engineer at Broad Reach Engineering, a small firm that develops spacecraft. Don can be contacted at **trundlar@cox.net**.

Mark Finkle

Mark Finkle is a Senior Developer and Project Lead with Minitab Inc., a commercial software company focusing on statistical software and related products. He also maintains a blog with related information at www.weborama.blogspot.com. Mark can be contacted at **mark.finkle@gmail.com**.

Malcolm Smith

Contributing Editor Malcolm Smith is owner of MJ Freelancing, which develops custom components and bespoke projects. Malcolm is also a contributing author of the C++Builder 5 Developer's Guide and he is Chief Analyst Programmer for Comvision Pty Ltd. designing and implementing security management systems, concentrating on the integration of disparate CCTV and alarm systems as well as streaming digital video into security control rooms. Malcolm can be contacted at **msmith@bcbjournal.com**.

Damon Chandler

Damon Chandler develops image processing and graphics-based applications in the Visual Communications Lab at Cornell University, where his research focuses on image compression algorithms. Damon is a co-author of the Windows 2000 Graphics API Black Book, a contributing author of the C++Builder 5 Developer's Guide, and a member of Team Borland (www.teamb.com). Damon can be contacted at **editor@bcbjournal.com**.

C++Builder Developer's Journal (ISSN 1093-2097) is published online monthly by Encoded Communications Group, 66 Lois Lane, Ithaca, NY 14850.

Customer Service: support@bcbjournal.com

Customer Relations (Voice)	(607) 227-3757
Customer Relations (Fax)	(707) 238-3031

Send all written correspondence to:

EnCoded Communications Group
66 Lois Lane
Ithaca, NY 14850

Editorial: editor@bcbjournal.com

Editor-in-Chief	Damon Chandler
Managing Editor	Jared Bish
Contributing Editors	Bob Swart
	Brent Knigge
	Malcolm Smith

Price

Personal	$49/year
Personal with email PDF delivery	$52/year
Corporate/Library/Institutional	$79/year