Wintellect Know how.

Kenn Scribner's CODE Café

The Matrix- Screen Saving Me from Myself

by Kenn Scribner

Man, I'm tired. Not even the usual dose of Diet Dew is helping me these days. I've spent the past year writing some code for an idea I'd had, and I'm spent. It's always delightful creating your own software, but I should have remembered to save some emotional energy for marketing the stuff. Just because you write it doesn't mean anyone will buy it. I guess I should have taken some business courses in graduate school.

Times like this, I tend to dodge into familiar places, hunker down, and recharge. Frankly, some good times and good conversation would do nicely, so I snapped the top of my laptop down and headed out the door. It had been a while since I'd visited the Code Café, and it was high time for dropping in. As the song says, it was time to head somewhere where everybody knew my name.

Angie met me at the door (she's not usually a door-greeter, but I didn't mind). "Johan, look who finally came back." Hmm, I could tell she was a little upset I'd not stopped by for awhile.

"Hi, Angie! Sorry it's been a while…I was writing some code on a tight schedule and just couldn't get out." I figured if anyone would understand being a bit too busy, it would be Angie. And, to my relief, her face relaxed a bit. She understood.

"Well, glad to see ya." And she smiled. Johan, behind the counter serving up some meatloaf, smiled as well.

"The usual?" he called out. Just hearing the question made me feel a lot better. It *is* good to go someplace where people know your name.

"Absolutely!" I said. Okay, maybe a bit more Diet Dew couldn't hurt. I smiled as well. "So what's on the menu today?" I asked Angie.

"Screen savers."

"Screen savers? What's up with that?" I mean, I expected some hefty communications thing, like some code targeted towards the Windows Communication Foundation (WCF). I'd been meaning to get into that a bit deeper. Gimme workflow, personalization providers, distributed applications, XAML, or something! But screen savers?

She looked at me quizzically, hands on hips. "Yeah, and what's wrong with that? You can learn a lot by coding a good screen saver. It's a fundamental component in Windows user interfaces, and coding one correctly isn't as easy as you might at first think." Ah, okay. Roger that. Sitting down and shutting up!

"Cool," I said. "So tell me about them."

Copyright 2005, 2006 – Wintellect, LLC

She began by telling me something interesting. "Screen savers actually depend on the command-line parameters they're given when they start up." Wow, I didn't know that. I thought most Windows applications are all-graphic, with options and settings being assigned by the user from the GUI. While many applications accept command-line parameters, not many actually depend upon them. "Depending upon the parameter, the screen saver starts up in vastly different modes. But the real fun is that you get to write some imaginative graphical code. The whole idea is to have fun doing it."

Now how did she know I needed a "fun break?" Maybe she knows more than just my name. "I like it," I told her. But what to write? Angie said anything is fair game, so I sat and thought a minute while I drank my Dew. Then it came to me… "Hey, Angie, how about doing 'The Matrix'!"

She smiled again. "Exactly" she said. "It should be fun to write, but there is a lot to get right to make it work."

I don't have a Matrix screen saver, so why not?

The Matrix

If you've not see the movie, the hero, Neo, is living in a…well, if you've not seen it, I won't ruin it for you. But in the movie you see computer screens from time to time that have black backgrounds with green characters streaming down from top to bottom. As they stream, they leave trails that fade over time. Some streams move quickly towards the bottom, and some more slowly. Some are bright green while others are a more dull green color. Figure 1 probably shows it better than I can describe it:

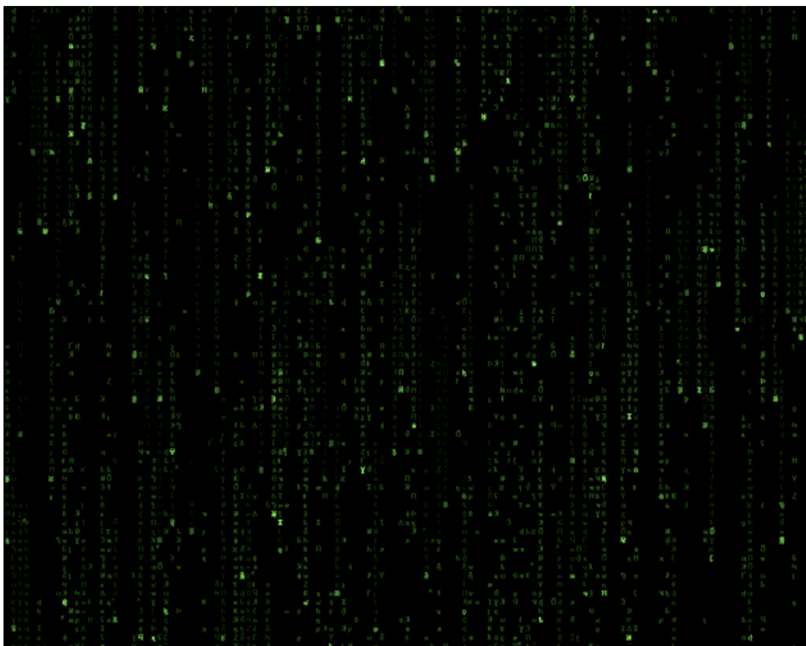

Figure 1. *The Matrix*

While rummaging around the Internet, I found an old but well-done Win32 implementation of a [Matrix screen saver](). It was written in thinly disguised C as C++, but translating the meat of the algorithm wasn't difficult. Interestingly, the only useful part for me of the entire original implementation was a short bit of code that manages some arrays that retain information about the streams. This was such a good starting point it was hard to turn down.

Essentially, Louai's algorithm creates an array of "streams" (not .NET streams but rather an individual character stream on the screen). The streams are assigned to text columns, and their color, speed, current position, and displayed character are controlled and updated when a timer fires. Streams are reused three times down the screen. Each time the stream travels down the screen, the next iteration uses a darker color of green and slows down. After the third time down the screen, the stream is marked as inactive and scheduled for reuse.

The current character position a stream maintains also features trailing characters. This trail must be erased over time--if it isn't, the entire screen would be filled with green characters, reducing the visual impact. So each stream is assigned a number of characters to trail. When the stream is graphically updated (the lead character advances), trailing characters will be randomly erased.

Translating the algorithm didn't involve a direct port to .NET (C#). Louai's original code used console fonts and allocated memory for the maximum number of streams no matter how many were actually to be displayed. I also removed some code to improve the visual effect slightly. But it was a great start, with all of the graphical work essentially complete. Some tweaking, nips and tucks, as well as some refactoring, and it was looking good. I'll describe the algorithm and the graphical process in a bit more detail after discussing some screen saver basics.

A .NET Screen Saver

You might think writing a screen saver is a simple matter in version 2.0 of the .NET Framework. After all, Visual Studio .NET 2005 has a template that creates a basic screen saver for you.

But the Visual Studio template lacks a couple of details that significantly downgrade the value of the screen saver for an end user. The command line doesn't handle a preview operation, so the screen saver totally ignores the preview request when the user selects it in the selection dialog. And the options dialog, if any, should slave itself to the parent window handle provided on the command line, if any. This is more difficult to do since contemporary versions of Windows, for security reasons, do not allow external processes to control their windows (at least not easily). But if you don't deal with this, when the user closes the preview dialog without first closing the screen saver's options dialog, the options dialog remains on screen.

Copyright 2005, 2006 – Wintellect, LLC

There are other details to consider as well. For example, how many of you have more than one monitor connected to your system? Quite a few, I'm sure. Would you consider a screen saver that ran on your primary monitor alone to be very good? Of course not. You expect screen savers to save all of the screens attached to your system. I know one developer that has four monitors attached to his system (and I extend my thanks to him for helping me work through the support of large numbers of monitors). Even with two monitors, the monitors could be (logically) aligned in one of two ways: horizontally (side by side) or vertically (one atop the other). Four or more monitors could be aligned in a linear arrangement, essentially a 1 x 4 array, or in a "video wall" arrangement, a 2 x 2 matrix. For your screen saver to work correctly, you must sense the monitor configuration and respond accordingly.

To write screen savers using .NET, we need to dig into some details a bit deeper than the code-generated screen saver. We'll need to perform a little Win32 "interop," understand what a "window handle" represents, and, at least for this screen saver, deal with some multithreading issues. So let's get started.

Screen Saver Command-line Parameters

"Probably the best place to start when going back to basics with a screen saver is to start with the command line," Angie said. You would think that researching this would be a simple task, but many available resources conflict or give partial information. The parameters I show here are valid for Windows XP and Server 2003. With this in mind, screen savers respond to the following command-line parameters:

- None, meaning the screen saver displays its configurable properties dialog box.
- "-c", meaning show the screen saver's configurable properties dialog box with its parent being the current foreground window. (We'll ignore the last part and just display the dialog on the desktop.)
- "-c:XXXXXXX", meaning show the screen saver's configurable properties dialog box using the provided window handle (known as an HWND) as the parent. We're supposed to treat our dialog as modal to the parent, but we'll sidestep this issue and merely respond to the closure of the parent, at which time we'll close the dialog as well. The HWND is a string representation of an integer value that logically represents an index into a table of window data structures.
- "-p XXXXXXX", meaning show a "preview" using the graphical context of the provided window. Unlike the "-c" option, here the window handle is separated from the command argument by a space rather than a colon. This matters because the command line argument array will have two strings in this case versus one (that must be parsed) for the configuration option.
- "-s", meaning run the screen saver as a screen saver. It should take over the screen, display its content, and shut itself down when the user presses any key or moves/clicks the mouse.

A quality screen saver responds to all of these conditions, but as you'll see, writing it can be a bit tricky. The preview window, in particular, poses an interesting challenge. Let's briefly look at command-line processing, using the Matrix screen saver as an example.

Wintellect Know how.

Kenn Scribner's CODE Café

Interpreting Command Line Parameters

When the Matrix screen saver begins execution, it is essentially a console application. Like any Windows application, it is passed any and all command-line arguments for processing. The arguments come in as an array of strings, with zero or more strings in the array (none or many command-line parameters). The command-line argument set does not include the executable filename as the first parameter (if you were working with Win32 directly, you would normally strip the first parameter because it *would* contain the executable filename--the .NET Framework strips it for you). Given this array, we'll need to see what parameters were provided by examining the various strings and respond accordingly.

There are many ways to read and interpret command-line parameters. I elected to take the most straightforward approach and examine the first command-line argument, if any. Given this, we can decide in what mode to execute very quickly using simple conditional logic. If additional arguments are required, we can dig those out within the various conditions. Perhaps some code will clarify this:

```
string cmd = args.Length > 0 ?
   (args[0].Length > 1 ?
    args[0].ToLower(CultureInfo.InvariantCulture).Trim().Substring(0, 2) :
   args[0].ToLower(CultureInfo.InvariantCulture).Trim().Substring(0, 1)) : "";

if (cmd == "/c" || cmd == "-c"  || cmd == "c" || cmd == "" )
{
   // Options dialog.
} // if
else if (cmd == "/p" || cmd == "-p" || cmd == "p" )
{
   // "Preview" mode.
} // else if
else if (cmd == "/s" || cmd == "-s" || cmd == "s" )
{
   // Run screen saver.
} // else if
else
{
   // Sorry...don't recognize the command-line parameter.
} // else
```

Here we just ask if there are command-line arguments, and if there are, we strip out the first one or two characters and make simple string comparisons for the possible variations (- versus /, or even no delineator at all). If the original command line had no arguments, the command-line argument array will contain null element values. We'll also check for this condition and substitute an empty string if necessary for quick and easy string comparison later.

The preview mode will require two command-line arguments. The first is the -p itself and the second is the string representation of the window handle for the window in which we're to draw the preview. The other command-line arguments all require just the command itself, or in the case of the configure option, the parent window's handle concatenated to the configure command (for example, -c:1245381). To extract the window handle from the configuration argument, we'll use the string class's Split function using the colon as the separator character.

Configuring the Screen Saver

The screen saver's configuration dialog is simply a form that allows the user to set and change whatever configurable parameters make sense for your screen saver. In the case of the Matrix screen saver, you can establish the number of streams (trails), how long the trails extend, how many characters in the trail are left alone before erasing begins, and a random character padding value that serves to increase or decrease the length of the trails just to vary what we see on the screen a bit.

Figure 2 shows the Matrix screen saver's configuration dialog.

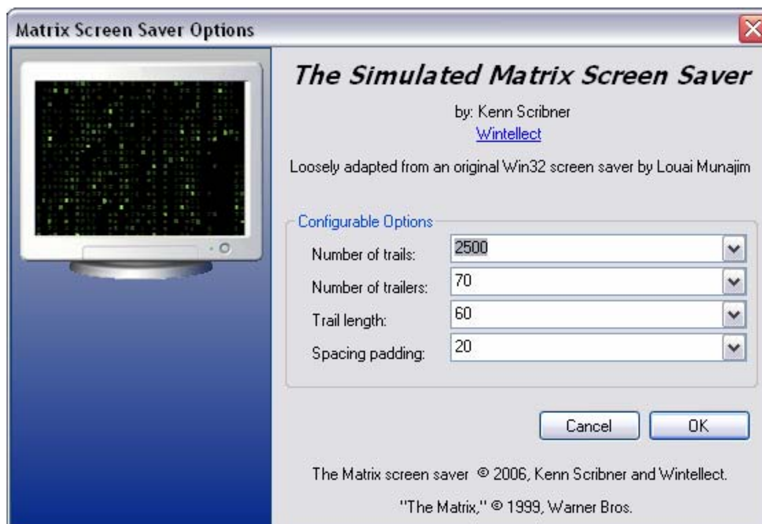

Figure 2. *The Matrix screen saver's configuration dialog.*

While you're free to save the configurable options wherever you choose, isolated storage makes a lot of sense. We could use the system registry, but then we could have permissions issues for weaker accounts. Isolated storage is nice because it's tailored to each user and can be easily backed up by system administrators. Working with isolated storage isn't difficult, and since I described it in detail for my *RssGator* application, I won't revisit it here. I follow the very same pattern here as I did for *RssGator*.

What is interesting is how we determine whether or not the parent window, if any, is closed so that we can close this form as well. To put it another way, if our screen saver is started

Copyright 2005, 2006 – Wintellect, LLC

with the -c option and provided a window handle to use as the parent window, then our dialog box is expected to “take over” the parent window, just as any modal dialog box would do. If we don’t do this, or at least take this into account, the user could close the parent window from underneath us, leaving our configuration dialog box visible and available for user interaction. This is *not* what we want!

Contemporary versions of Windows have security features make this sort of external modal hijacking more difficult to do than earlier versions. If Windows didn’t do this, malicious programs could interact with beneficial applications and wreak havoc on your system.

So we’re going to simulate the behavior we want. Although we won’t be a modal dialog (the user can still close the underlying parent window), we’ll periodically check to see if the parent window is still available. If so, we’ll continue to display our configuration form. If not, we’ll close our configuration form with a cancel operation.

I can think of a couple of ways of doing this, but probably the most effective is to use the old Win32 SendMessageTimeout API call with the WM_NULL message coupled with a Windows Forms timer. SendMessageTimeout will send the WM_NULL message to the parent window, and if the parent window is accepting messages, we’re assured that it’s still available to the user and that our configuration form is appropriately displayed. If the parent window were to be closed, the message would fail (or timeout), and we should close our configuration form prematurely. Here is the code for this:

```
void Timer_Tick(object sender, EventArgs e)
{
   // Stop the timer for the moment
   _timer.Stop();

   // Check for the availability of our parent. We'll send a null message
   // to the parent window. If the parent responds, we restart the timer
   // and continue. If the message times out, our assumption is the
   // parent window is now gone (i.e.: the user canceled the desktop
   // properties dialog without first closing our options dialog). In that
   // case, we close ourselves with a cancel operation.
   UInt32 lpdwResult;
   if (SendMessageTimeout(_hwndParent, WM_NULL, 0, 0, SMTO_BLOCK,
                     SMTODelay, out lpdwResult) == 0)
   {
      // Parent isn't responding, so we'll close ourselves
      this.DialogResult = DialogResult.Cancel;
   } // if
   else
   {
      // Parent is still up and running, so restart the timer
      _timer.Start();
   } // else
}
```

Copyright 2005, 2006 – Wintellect, LLC

If you look in the .NET documentation, you won't find a reference to SendMessageTimeout. That's because this method is provided by the unmanaged DLL *User32.dll*. This is an example of Win32 unmanaged DLL interoperability using *platform invoke* (P/Invoke). But since the method isn't part of .NET, for the compiler to be able to generate code to execute the call we'll need to tell the compiler where to find the method and what its method signature looks like (that is, what parameters and such it expects). We'll do this using .NET's DllImport attribute, found in System.Runtime.InteropServices. (Keep interoperability in mind as we'll revisit it when we look at building the preview window.) Here is the definition I created for SendMessageTimeout:

```
/// <summary>
/// Sends a Win32 message to the given window with a timeout, in case
/// the window isn't responding.
/// </summary>
/// <param name="hWnd">Window handle of window to call.</param>
/// <param name="Msg">Win32 message to process.</param>
/// <param name="wParam">WPARAM to process.</param>
/// <param name="lParam">LPARAM to process</param>
/// <param name="fuFlags">Execution flags</param>
/// <param name="uTimeout">Timeout (in milliseconds)</param>
/// <param name="lpdwResult">Value returned from application when Win32
/// message is processed (message dependent).</param>
/// <returns></returns>
[DllImport("user32.dll", CallingConvention = CallingConvention.StdCall,
        EntryPoint = "SendMessageTimeoutW")]
private static extern UInt32 SendMessageTimeout(IntPtr hWnd,
                                                UInt32 Msg,
                                                UInt32 wParam,
                                                UInt32 lParam,
                                                UInt32 fuFlags,
                                                UInt32 uTimeout,
                                                out UInt32 lpdwResult);
```

Basically, what we'll need to do is to pass into SendMessageTimeout the following:

- The receiving window's handle
- The message and any associated parameters
- The operational flags to use (we'll block in this case, which is to say we'll wait for the time duration to expire or the message to be processed)
- The timeout period

Copyright 2005, 2006 – Wintellect, LLC

We'll have returned to us any result from the message, such as an integer or Boolean return value (in the lpdwResult variable). I've defined the values for WM_NULL and SMTO_BLOCK for convenience as well:

```
/// <summary>
/// Win32 API message WM_NULL. WPARAM and LPARAM are unused. Returns
/// zero if application processes the message.
/// </summary>
private const UInt32 WM_NULL = 0x0000;

/// <summary>
/// "Send message timeout" blocking option. Our thread blocks while
/// waiting for the called application to process the given Win32
/// message.
/// </summary>
private const UInt32 SMTO_BLOCK = 0x0001;
```

As for the parameters to DllImport, I'm telling the P/Invoke machinery that we want to execute the SendMessageTimeoutW method found in the *User32.dll* DLL, and we'll be using the standard Win32 calling convention (technical jargon for how the parameters are placed on the CPU's execution call stack). Since .NET is Unicode based, we want to execute the "wide" character version of the method, hence the W on SendMessageTimeoutW. Ah, the good (or bad!) old days of Win32 programming.

Window Handles

If you're a scraggly old Win32 programmer like me, then this section is a review and probably not necessary. But there is a whole crop of developers out there that haven't programmed in any environment but .NET, so for you, this information might be useful, especially when dealing with P/Invoke calls such as we're doing here.

I've mentioned the window handle several times thus far, but I've not described the window handle in any detail. On the one hand, this is what .NET saves us from, so describing it isn't necessary in a .NET programming environment.

But when you dip back into Win32 programming, which we do from time to time in .NET applications, especially Windows Forms applications, you need to know how Windows is working under the covers to successfully negotiate your Win32 method calls. Now, as soon as I say that, the astute developer would tell me that the new version of Windows (Longhorn, or Vista as it's now called) is going to be CLR-based instead of Win32-based. This is true, but we still need to know a bit about Win32 for this application at the very least, so on we go.

A handle is simply a pointer to a pointer. "Okay," you ask, "what's a pointer?" Well, each chunk of information in the computer's memory has an address so that the CPU can request that piece of information for processing. This address is called a pointer in many computer languages (most notably C++, which was used to write most of the Win32 API).

A handle, then, is really an index into a table of pointers, which themselves indicate the location of the data structure that defines a window in memory. This table structure allows the operating system to shift things around in memory, yet the index into the table you use does not change. Many things in Win32 require handles—windows, events, mutexes, and graphics device contexts (akin to .NET's Graphics object) to name a few.

So when Windows hands your screen saver an integer value that represents a window, what you're really getting is a handle, or token of sorts, that you use to access a data structure that represents a window. For our purposes, we'll need to access the window's graphical surface, for drawing, as well as query some of the basic window information, such as its client (drawing surface area) size.

The Preview Window

One of the coolest things I think screen savers do is something a bit unusual in Windows programming. Normally your Windows application is given its own window in which to paint its user interface. But the preview window for screen savers is owned by another window, and to put your screen saver into the preview viewing area, you need to resort to some trickery.

We've already seen that the window in which we're to draw our preview screen saver is provided to us on the command line. The -p command-line argument will be followed by the string representation of the integer window handle. The window in which we're to draw our screen saver is shown in Figure 3.

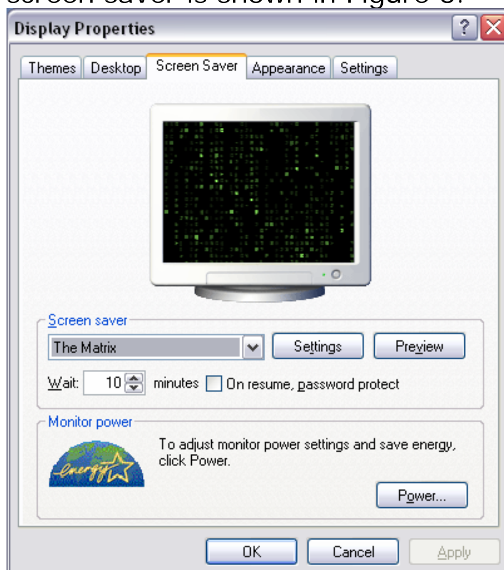

Figure 3. *The Matrix screen saver's preview and parent window.*

The preview is shown in an image of a computer monitor. Pretty cool. But looking at this you probably already noted that we need to know how large the area is in which we're to draw, and we'll need to scale our screen saver to the smaller screen area.

I've created a special class, MatrixPreviewWindow, to deal with our screen saver's unique requirements. In this class you'll see a lot of the same code you will see in the main screen saver's custom panel control. But the preview operates differently than the actual screen saver, at least for this screen saver, so the functionality differs slightly. For example, the screen saver will terminate whenever a key is pressed, the mouse moves, or the mouse is clicked. If none of these things happen, the screen saver will run forever. The screen saver is nothing more than a Windows application (a form), and like any application it'll run until you close it.

The preview window is tied to its parent, so even though it will also run forever, or at least as long as the parent is visible, the preview must cease operation when the parent ceases operation. The preview window executes in a tight while loop, where it sleeps for a short duration and then checks exit conditions. We'll see this code shortly.

The primary method in the preview class, Run, executes the following logical operations:

- Determine the size of the parent's drawing area
- Loop until the parent window is visible
- While the parent window is visible, display the preview
- When the parent window is no longer visible, quit and clean up.
-

Two of these logical operations require P/Invoke calls: GetClientRect and IsWindowVisible. Requesting the client rectangle, which is to say the area in which we're to draw our preview, will require the use of a Windows-based structure, RECT. We'll therefore also need to define that as well:

```
/// <summary>
/// RECT structure, which is expected by GetClientRect.
/// </summary>
[StructLayout(LayoutKind.Sequential)]
public struct RECT
{
   public Int32 left;
   public Int32 top;
   public Int32 right;
   public Int32 bottom;

   public RECT(Int32 l, Int32 t, Int32 r, Int32 b)
   {
      this.left = l;
      this.top = t;
      this.right = r;
      this.bottom = b;
   }
}
```

```
/// <summary>
/// Retrieves client area drawing rectangle for given window.
/// </summary>
/// <param name="hWnd">Window handle for window to query.</param>
/// <param name="rect">Returned rectangle value.</param>
/// <returns>True if rectangle structure was completed, false if error.</returns>
[DllImport("user32.dll")]
private static extern bool GetClientRect(IntPtr hWnd, ref RECT rect);

/// <summary>
/// Determines if given window is visible (if so, we know the
/// graphics surface is valid, allowing us to draw our preview).
/// </summary>
/// <param name="hWnd">Window handle for window to query.</param>
/// <returns>True if window is visible, false if not.</returns>
[DllImport("user32.dll")]
private static extern bool IsWindowVisible(IntPtr hWnd);
```

At first glance we could resort to using SendMessageTimeout to determine when we're to start and stop drawing. But windows begin life long before they become visible on the screen, so it's more reliable for our purposes here to query the parent's visible status. Processing WM_NULL doesn't necessarily mean we're free to begin drawing on the parent's drawing surface.

Now that we have a native way to determine how large of an area we have in which to draw, and we can figure out if we have a window in which we can render our output, we'll need to scale our output to the size of the provided drawing area. This is purely visual aesthetics. In this case, you might not find it surprising to find the characters drawn on the screen are drawn using a .NET Font object, so scaling for this screen saver is merely a matter of using a much smaller font. The visual effect is pretty dramatic, though. Other screen savers might have to take more drastic measures.

Putting all of this together, the Run method looks like this:

```
public void Run(IntPtr hwndParent)
{
   try
   {
      // We'll need to know how large the preview area
      // is supposed to be so we can generate the proper
      // canvas bitmap.
      RECT rect = new RECT();
      if ( GetClientRect(hwndParent, ref rect) )
      {
         // Place the drawing rectangle and set some
         // other parameters.
         Location = new Point(rect.left, rect.top);
```

```
Size = new Size(rect.right - rect.left, rect.bottom - rect.top);
Font = new Font("Microsoft Sans Serif", 2.75f,
    System.Drawing.FontStyle.Bold,
    System.Drawing.GraphicsUnit.Point,
    ((System.Byte)(0)));

// Since we'll break our event loop when our parent
// window becomes not visible, we have to make sure
// it's actually visible before we start. There is
// a delay in the system from the time we're told to
// display the preview and the time the preview window
// becomes visible. So we'll wait here until it
// becomes visible. To prevent an infinite loop, we'll
// wait a predetremined amount of time (20 seconds)
// for the preview window to become active. If for
// some reason it isn't activated in time, we'll
// break the loop and return, resulting in no
// preview (but also no infinite loop situation
// either).
DateTime dtTimeout = DateTime.Now.AddSeconds(20.0);
while ( !IsWindowVisible(hwndParent) )
{
    // While we wait, process events...
    Application.DoEvents();

    // Now check the timeout
    if ( DateTime.Now > dtTimeout )
    {
        // Hmm...some problem, so end the loop and
        // quit...
        return;
    } // if
} // while

// Now the parent window is visible, so initiate
// graphics ops. While the window is visible, we'll
// run our special previewer object...
while ( IsWindowVisible(hwndParent) )
{
    try
    {
        // Delay awhile...
        System.Threading.Thread.Sleep(_period);

        // Reset the updating event so we can't dispose
        // of ourselves prior to update completion.
        _updating.Reset();
```

Copyright 2005, 2006 – Wintellect, LLC

```
                // ...and update the screen.
                Refresh();
            } // try
            catch
            {
                // Some error, so simply break from
                // loop and clean up.
                break;
            } // catch
            finally
            {
                // Okay to dispose
                _updating.Set();
            } // finally

            // Keep processing events while we wait
            Application.DoEvents();
        } // while
    } // if
} // try
catch
{
    // Do nothing...
} // catch
}
```

The code you see follows the basic set of logical operations I described. We first try to obtain the drawing rectangle:

```
RECT rect = new RECT();
if ( GetClientRect(hwndParent, ref rect) )
{
   ...
}
```

Assuming we do obtain the drawing rectangle, we loop until the parent window is visible:

```
DateTime dtTimeout = DateTime.Now.AddSeconds(20.0);
while ( !IsWindowVisible(hwndParent) )
{
   // While we wait, process events...
   Application.DoEvents();

   // Now check the timeout
   if ( DateTime.Now > dtTimeout )
   {
      // Hmm...some problem, so end the loop and
      // quit...
```

Copyright 2005, 2006 – Wintellect, LLC

```
        return;
    } // if
} // while
```

When the parent window does become visible, we enter another loop that waits for the parent window to become non-visible:

```
while ( IsWindowVisible(hwndParent) )
{
    ...
}
```

The heart of this loop consists of a brief delay

```
System.Threading.Thread.Sleep(_period);
```

coupled with a display refresh, after resetting a manual reset event to prevent early disposal (more on that in a moment):

```
_updating.Reset();
...
Refresh();
```

Of course, something has to call the Run method. Moreover, something needs to accept the parent window handle and from that derive a graphical context into which our preview will paint. This is accomplished by the following code, which executes in the main thread once we've determined we're to display the preview window:

```
try
{
    // Pull the parent handle
    UInt32 hwndParent = UInt32.Parse(args[1]);

    // Start a new instance, with our parent providing us
    // with a graphics surface onto which we'll paint...
    MatrixPreviewWindow preview = null;
    Graphics gfx = Graphics.FromHwnd((IntPtr)hwndParent);
    if (gfx != null)
    {
        try
        {
            // Create the special preview version
            preview = new MatrixPreviewWindow(gfx);

            // Run the preview
            preview.Run((IntPtr)hwndParent);
        } // try
```

```
        catch
        {
            // Do nothing...we'll just have no preview...
        } // catch
        finally
        {
            // Dispose of the preview object
            if (preview != null) preview.Dispose();

            // Be sure to dispose of the parent's graphics
            // object...
            if (gfx != null) gfx.Dispose();
        } // finally
    } // if
} // try
catch
{
    // Do nothing...we'll just have no preview...
} // catch
```

The parent window handle is parsed as an integer and held in a local variable:

```
UInt32 hwndParent = UInt32.Parse(args[1]);
```

Using the integer version of the parent's window handle, we request that window's Graphics object:

```
Graphics gfx = Graphics.FromHwnd((IntPtr)hwndParent);
```

If we can't obtain the parent's Graphics object, we'll simply not show a preview. But assuming we can, we'll create an instance of our special preview class and then run the preview:

```
preview = new MatrixPreviewWindow(gfx);
...
preview.Run((IntPtr)hwndParent);
```

When the preview window senses the parent window has been dismissed, the Run method will terminate. Once that happens, a finally block will execute to dispose of both the preview window and the parent's graphics object:

```
if (preview != null) preview.Dispose();
...
if (gfx != null) gfx.Dispose();
```

I need to mention two details. First, so that we don't loop forever waiting for the parent window to become available, I set an arbitrary time limit of 20 seconds. If after 20 seconds the parent window doesn't activate and become visible, the preview loop self-terminates.

Copyright 2005, 2006 – Wintellect, LLC

The other detail is one I didn't originally foresee, both for the preview window and for the screen saver itself. In this case, when the preview loop terminates, ultimately the application will be garbage collected and the class' Dispose method called. In Dispose, I release the graphics bitmaps I'm using to draw the Matrix. Once released, bitmaps can't be reused, and any attempt to do so results in an exception.
As it happens I had a *race condition*. Sometimes Dispose was called after the Matrix code was finished with the graphics bitmaps and sometimes it wasn't. Since different threads were working with the application, it was possible to dispose of the graphical bitmaps before the thread rendering the Matrix output on the screen was truly through with its rendering process. This is bad.

So I added a manual reset event, _updating, that is reset when the code rendering the Matrix is executing. If Dispose is called and _updating is still reset, Dispose waits until the event triggers before it continues disposing of the underlying graphical bitmaps. No more race condition!

(As luck would have it, I never saw this problem on my system. However, as soon as Angie got hold of it, the bug reared its ugly head. It's always the way.)

Finally, you probably saw the calls to Application.DoEvents(). Since we're in a while loop, the thread we're using is completely tied up during our loop execution. Calling Application.DoEvents() allows the graphical user interface presented by the preview dialog to continue to accept user input so that to the user it doesn't appear to be hung.

The Main Event

And now what we've all been waiting for…the screen saver itself!

I chose to implement the screen saver as a Windows Forms form, using a custom panel control as the drawing surface. This lets me reuse the panel in other forms if I want to (you can, in fact, use the panel as a true panel and place form controls on its surface in your own applications). I felt it neatly encapsulated the functionality of a panel combined with my custom drawing code.

When the screen saver executes, the form resizes itself to fit the screen after setting its borders to "none." The panel that renders the Matrix is docked to the four edges of the enclosing form, so it automatically resizes as well, effectively filling the screen with imagery.

Supporting Multiple Monitors

But this calls into question another piece of magic we'll need to address. How large is the screen? Or to be more precise, what if I have multiple screens (monitors) attached to my system? Surely the screen saver must be able to deal with this condition!

Well, originally it didn't. I personally don't use multiple monitors, although after writing this and hearing how retro I am, I'll probably make the investment. I do like my big screens.

Copyright 2005, 2006 – Wintellect, LLC

My original implementation simply examined the size of the primary screen, which I pulled from the Windows Forms Screen class, Screen.PrimaryScreen. The associated screen object has a Bounds property that represents the resolution of the primary screen, or in other words the screen's display size, in pixels.

But again, as soon as I gave it to Angie, the screen saver fell on its face. Angie uses a dual monitor system and kindly informed me I was a dork. Actually, Angie didn't *say* I was a dork, but she sure looked like she was thinking it.

If you look at the Screen object in System.Windows.Forms, you'll see that in addition to PrimaryScreen it has an array of screens available, AllScreens. AllScreens tells you not only how many monitors are present on the system but also their resolutions and relative logical location through each element's Bounds property.

My next iteration to Angie now took her second monitor into account. If AllScreens indicated two monitors were present, I added the horizontal resolutions and created my internal graphics bitmap (my "canvas") such that it became the width of both monitors combined. This worked for Angie, and I was happy.

I was so happy that for the fun of it I gave the screen saver to Johan. Wouldn't you know? Johan has FOUR monitors attached to his system, and I was again a dork. Clearly my quick fix for Angie wasn't going to work for everyone, so I needed to solve the multiple monitor mystery once and for all.

The ultimate solution is not necessarily suitable for all screen savers. My solution is to examine AllScreens and total the horizontal and vertical resolutions for the monitors I find there. There is a bit more intelligence built in than that, as the monitors could be in a horizontal, linear arrangement (or vertical for that matter). Simply adding all vertical resolutions would result in a tremendous waste of memory. But essentially I add the resolutions and build one big bitmap in memory onto which I paint the Matrix. This creates a large in-memory bitmap, consuming a good chunk of memory, but it allows for continuous Matrix flow between vertically-arranged monitors (such as for a video wall). If I didn't do this, the trails would be disjointed as they moved vertically from screen to screen, reducing the effect. Other screen savers could possibly deal with each monitor as a single entity, depending upon the graphical effects in place for that screen saver. You'd need just as much memory, but the memory would be allocated and blocked a bit more efficiently since it wouldn't be one big blob.

In code, then, my algorithm takes this form:

```
// Then set up the monitors
Int32 totWidth = 0;
Int32 totHeight = 0;
for (Int32 i = 0; i < Screen.AllScreens.Length; i++)
{
   // Check the dimensions of this screen
   if ( Screen.AllScreens[i].Bounds.X == 0 )
   {
```

```
        // We simply compare the width of this screen to
        // the total width we've accumulated, and if greater,
        // use this screen's width. Otherwise, we're covered.
        if ( Screen.AllScreens[i].Bounds.Width > totWidth )
            totWidth = Screen.AllScreens[i].Bounds.Width;
    } // if
    else
    {
        // The widths are additive...
        totWidth += Screen.AllScreens[i].Bounds.Width;
    } // else

    // Height
    if (Screen.AllScreens[i].Bounds.Y == 0)
    {
        // We simply compare the height of this screen, similar
        // to what we did for width.
        if (Screen.AllScreens[i].Bounds.Height > totHeight)
            totHeight = Screen.AllScreens[i].Bounds.Height;
    } // if
    else
    {
        // The heights are additive...
        totHeight += Screen.AllScreens[i].Bounds.Height;
    } // else
} // for

// totWidth and totHeight are then used to create the canvas
// bitmap used to render the Matrix.
```

Seems simple looking at it now, but it wasn't obvious before I realized how many possible configurations there might be! Now, though, the Matrix should cover each and every screen attached to the system no matter the resolution. There may yet be disjointed effects if monitors of differing resolutions are combined, but this should be a good, general case solution for nearly everyone.

The Matrix

Let's now look at the creation of the on-screen Matrix. When the form MatrixForm loads, it first checks to see if any other instances of the Matrix are currently running. It does this by creating and demanding ownership of a named mutex object. If the demand is made, this instance is the only executing instance. If not, another instance has created the mutex already and retains ownership, so the instance in question simply shuts down.

Assuming the instance in question is indeed the only executing instance, the form pulls the saved option values from isolated storage using the Options class in its Load event handler. For convenience, this is the same class that maintains the optional values for the configuration dialog.

We then ascertain the size of the off-screen bitmap using the multiple monitor algorithm we've just seen and adjust the number of trails upward to account for multiple monitors. We wouldn't want to see a sparse Matrix. (Sorry, I couldn't help myself.)

Once the preliminaries are complete, the Matrix panel's Start method is called. MatrixPanel. Start simply then starts a timer. Each time the timer ticks, the panel updates the positions of the visible trails by re-rendering each trail to the off-screen bitmap. The off-screen bitmap is then copied to the screen for viewing.

The off-screen bitmap is necessary; without it, we would lose the trail for each stream. You would only see the characters rendered for that given pass (the brighter lead character and the less bright initial trailing character). Any previously rendered characters would be erased, completely destroying the effect we're looking for.

The characters themselves are characters selected from the Microsoft Sans Serif font. If you look at the characters included with the Microsoft Sans Serif font, you'll see a great many visually interesting characters in the Cyrillic, Hebrew, Greek, Latin, and Thai alphabets. I chose characters I thought creative and placed them into an array of Unicode character values:

```
/// <summary>
/// Printable stream characters (Unicode).
/// </summary>
protected static char[] _chars =
{
   '\u046c', ... ,'\u046a',
   '\u0472', ... ,'\u019b',
   ...
   '\u0166', ... ,'\u0255',
   '\u026e', ... ,'\u0464',
};
```

A random character is selected from this array and is then displayed on the screen in an appropriate position, based upon the stream's current positional information. The Matrix panel itself allows you to select a different font (but not a different character set). For the screen saver I left this permanently assigned to the default font, Microsoft Sans Serif, rather than let the user select a new font.

Characters, of course, have width and height. Normally when rendering characters to the screen you want the entire character displayed. That is, you typically don't want the character to be clipped (chopped off). It would make sense, then, to create the Matrix streams such that the widest character dictates the width of the vertical character trails. However, some of the characters I selected are quite wide, causing too few trails to be displayed. So I elected to arbitrarily select the width of the letter N as the stream width, resulting in a greater number of vertical trails at the expense of clipped characters.

Copyright 2005, 2006 – Wintellect, LLC

The Matrix panel updates the position of all of the trails when the internal panel timer fires. This is a nontrivial task: there could be as many as 10,000 trails requiring status update. Moreover, for each active trail, the rendering code will loop three times. The first time through, the lead character is rendered in the brightest green color. The second time, the first trailing character is rendered (the character just above the current lead character, which effectively erases the former brightly colored lead character). The third time though, a random trail character is erased using the default background color (black). The following code actually places the characters on the screen:

```
/// <summary>
/// Draws the individual streams on the canvas we created when
/// we sized the window. Three sets of characters are drawn:
///
/// 0: Leading character (brightest color)
/// 1: First trailing character (subdued color)
/// 2: Trail erasure (black to eliminate trail over time)
/// </summary>
/// <param name="gfx">Graphics object used for drawing.</param>
void DisplayStreams(Graphics gfx)
{
   // Pre-allocate some memory
   float x; // X coordinate for drawn character
   float y; // Y coordinate for drawn character
   Int32 incR; // color increments for color differentiation
   Int32 incG;
   Int32 incB;
   RectangleF rect; // Character rectangle

   // For each stream...
   for (Int32 i = 0; i < _numStreams; i++)
   {
      // ...for each active stream...
      if ( _streamStatus[i] )
         {
         // ...pull the starting coordinates.
         x = (float)_startX[i];
         y = (float)_startY[i];

         // Cycle through the drawing states
         for (Int32 j = 0; j < 3; j++)
         {
            if ( j == 0 )
            {
               // Draw the lead character in the brightest color
               incR = _r/(_advanceDelay+1);
               incG = _g/(_advanceDelay+1);
               incB = _b/(_advanceDelay+1);
```

```
            Brush brush =
               new SolidBrush(Color.FromArgb(_r-_streamOrigSpeed[i]*incR,
                                _g-_streamOrigSpeed[i]*incG,
                                _b-_streamOrigSpeed[i]*incB));
            rect = new RectangleF(x,y,(float)_textWidth,(float)_textHeight);
            gfx.DrawString(String.Format("{0}",
             (_chars[(_rand.Next(MaxRand)*_chars.Length)/MaxRand])),
             this.Font,brush,rect,_drawFormat);
            brush.Dispose();
         } // if
         else if ( j == 1 )
         {
            // Draw the first trailing character in a subdued color
            incR = _r/3/(_advanceDelay+1);
            incG = _g/3/(_advanceDelay+1);
            incB = _b/3/(_advanceDelay+1);

            Brush brush = Brushes.Black; // no disposal required
            rect = new RectangleF(x,
                                 y-(float)_textHeight,
                                 (float)_textWidth,
                                 (float)_textHeight);
            gfx.FillRectangle(brush,rect);

            brush =
             new SolidBrush(Color.FromArgb(_r/3-_streamOrigSpeed[i]*incR,
                                _g/3-_streamOrigSpeed[i]*incG,
                                _b/3-_streamOrigSpeed[i]*incB));
            gfx.DrawString(String.Format("{0}",
              (_chars[(_rand.Next(MaxRand)*_chars.Length)/MaxRand])),
              this.Font,brush,rect,_drawFormat);
            brush.Dispose();
         } // else if
         else
         {
            // Erase additional trailing characters if possible/necessary
            Brush brush = Brushes.Black; // no disposal required

            rect = new RectangleF(x,
                        y-(float)((((_rand.Next(MaxRand)*_spacePad)/MaxRand)+
                           _trailLength)*_textHeight),
                        (float)_textWidth,
                        (float)_textHeight);
            gfx.FillRectangle(brush,rect);
         } // else
      } // for
   } // if
```

```
        // Since this loop might take awhile, allow events to
        // process (such as mouse and key events). If we don't
        // do this, the panel will (could) take a long time to
        // respond to the user input (it appears to be locked).
        Application.DoEvents();
    } // for
}
```

Add all of this looping to more loops to update the status of each stream (currently active or marked for reuse), and you have a lot of looping. And, as you know, looping takes time. Therefore, when the timer fires, we disable the timer, update the screen, and then re-enable the timer (sadly, another bug Angie's system uncovered for me).

The rendering code might at first seem daunting, but it's truly very simple. For each rendering pass, 1 – 3, we decide where the character (or black rectangle) is to be drawn and in what color. The "where" is assigned to a drawing rectangle, for example:

```
rect = new RectangleF(x,y,(float)_textWidth,(float)_textHeight);
```

The color is assigned to a brush:

```
Brush brush = new SolidBrush(Color.FromArgb(...));
...
brush.Dispose();
```

Remember that brushes we create must also be disposed of!

If we're drawing a character, we select the character from our character array using a random index. We then draw the character using Graphics.DrawString:

```
gfx.DrawString(...);
```

If we're erasing characters, we draw a black rectangle using Graphics.FillRectangle:

```
gfx.FillRectangle(...);
```

The only trick is to keep track of where each stream starts on the screen as well as its relative speed and other details. This information is kept in arrays:

```
protected Int32[] _startX = null;
protected Int32[] _startY = null;
protected Int32[] _streamSpeed = null;
protected Int32[] _streamOrigSpeed = null;
protected bool[] _streamStatus = null;
```

The arrays themselves will be allocated once we know how many streams (trails) we're to display.

Final Thoughts

Writing a screen saver in pure Win32 is easy from an integration standpoint, but it defeats the tremendous value .NET brings to the table as far as programming power and capability. So writing a screen saver using .NET makes a lot of sense, but to truly integrate the screen saver into Windows, you still must do some of that old Win32 magic. Not much! But some.

Screen savers pose some interesting challenges for debugging. If you automatically rename the compiler output from *XYZ.exe* to *XYZ.scr* (as I do with the sample code), then the F5/F11 debugging breaks in Visual Studio. You have to monkey with the post build commands to single-step through your code. Debugging the preview window is especially challenging because it runs under Windows control. Once you roll the mouse to hook it with Visual Studio, the screen saver terminates.

The screen saver, when displayed in the Desktop's Properties dialog, uses its name as its display value. This is the default functionality. If we were to assign an old Win32 style string table to our screen saver, the string with the table ID of 1 would be displayed instead (no, I've not figured out how to attach old resource types to .NET assemblies, so don't ask!).

To install the screen saver, simply copy the .scr file to the System32 directory of your primary Windows subdirectory (typically C:\Windows\System32) and select "The Matrix" from the list of available screen savers or right-click the .scr file and select *Install* from the context menu.

This is a fun screen saver. It's mesmerizing to watch. But hopefully there are some useful nuggets of information here you can use when developing your own Windows Forms applications. With luck, it won't be so long before I again visit the Code Café. Angie, being a very private person, doesn't accept e-mail from anyone but Johan, and even then only for shift schedule changes. But I can sure ask her a question for you. Just drop me a line at kenns@wintellect.com and I'll pass your question along. Until then, happy coding!

(A special *thank you* to John and Scott for helping me with testing the screen saver on their systems. Thanks for keeping me straight, guys!)