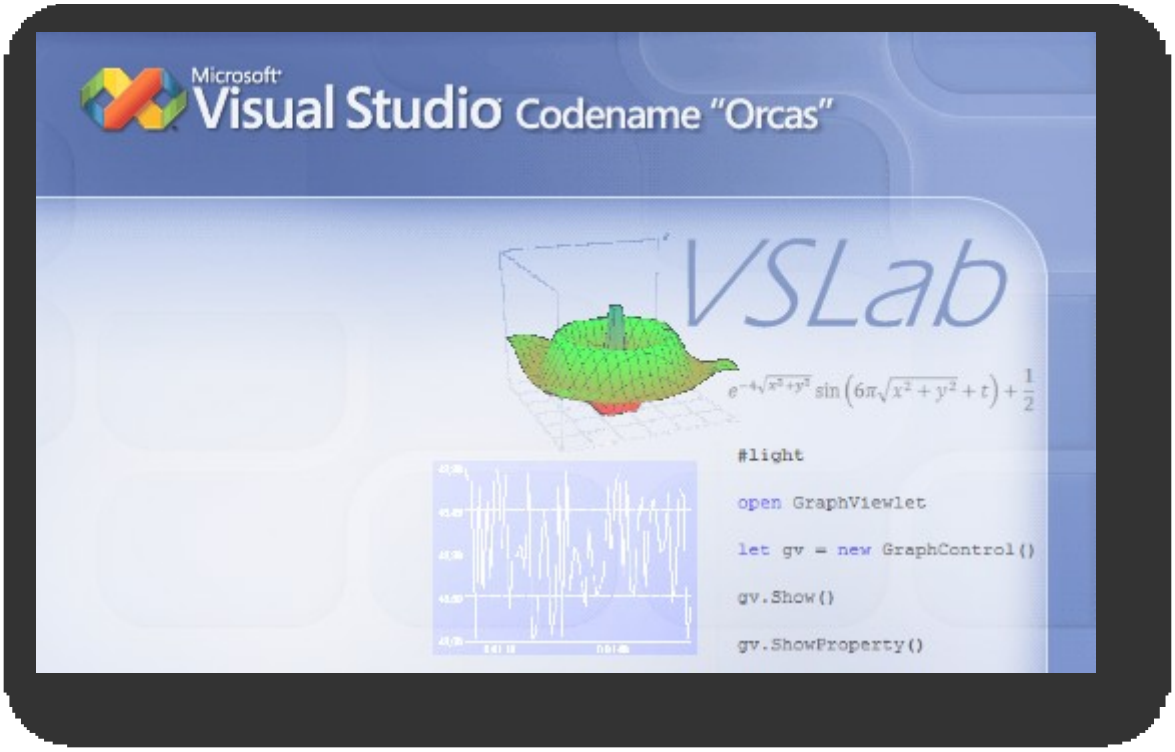
VSLab implementation

Whitepaper



In this paper we discuss the basic structure of VSLab and the core mechanisms used to allow *fsi.exe* to draw inside Visual Studio toolwindows. The content of the paper assumes general knowledge about Win32, the windows GUI messages and COM interop.

Author: Antonio Cisternino (cisterni@di.unipi.it), University of Pisa
Version: 1.0
Last update: 6/27/2008 2:14:00 AM

Introduction

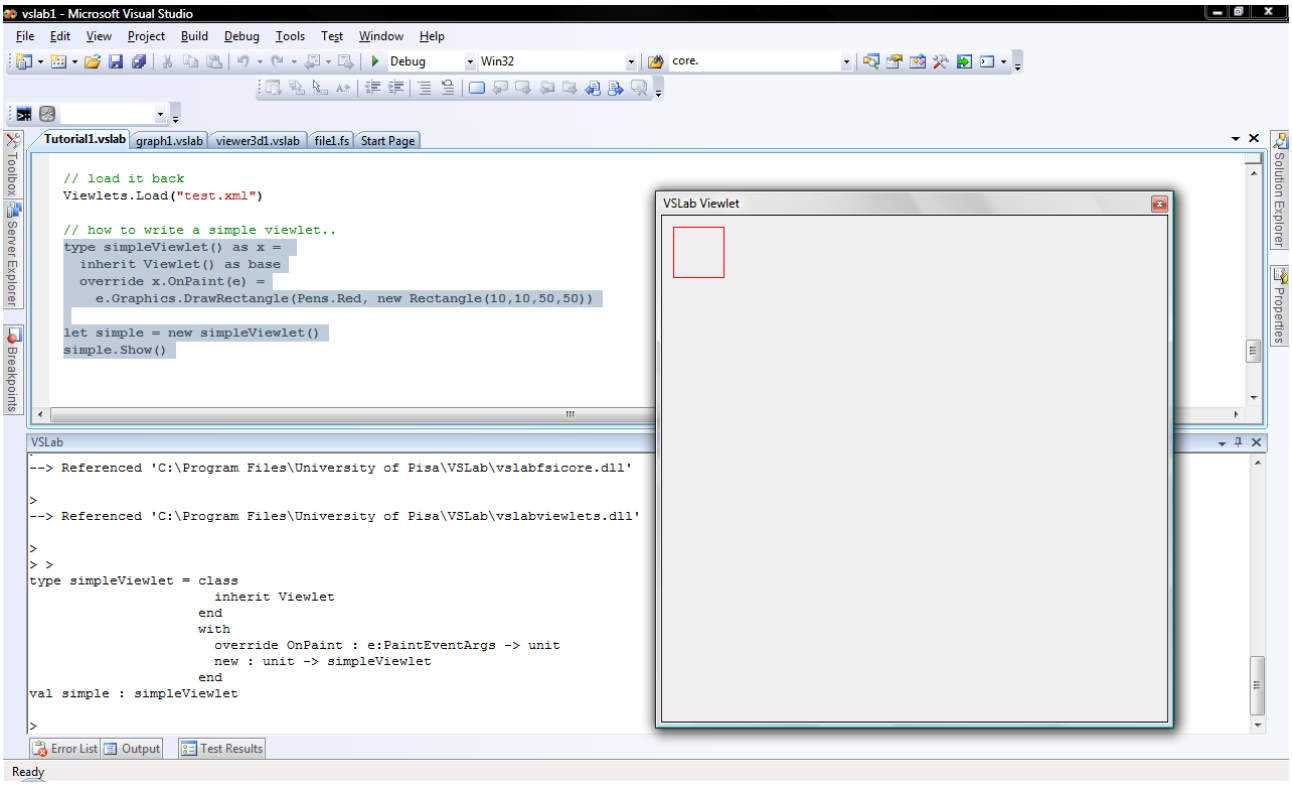
VSLab leverages on F# interactive and Microsoft Visual Studio extensibility to provide an interactive programming environment providing an interface similar to MatLab, but at the same time it targets a compiled programming architecture. There are many different views of this project, though the most challenging aspect was to allow F# interactive, the top level of the F# programming language, to interact with Visual Studio toolwindows while executing in a different process. At first sight it may seem just a technical exercise, but it is more than just that, since developing an interactive environment capable of efficient WinForms and DirectX graphics inside Visual Studio combined with the isolation provided by operating system processes contributes to offer a robust interactive system in which code is safely held by the editor and executed outside of it (no crashes of the execution system may corrupt the source code).

In the future we hope Visual Studio will provide core mechanisms to support this kind of interaction, but at the present we had to resort to many COM and Win32 features to workaround limitations of the Visual Studio extensibility model. This document discusses how the core mechanism of *viewlets* has been implemented and how we provide a WinForms interface that is similar to standard *UserControls* while providing cross-process drawing.

This paper is organized as follows: we first introduce the final interface; after this we briefly describe the COM-based VS extensibility model and the cross-process WinForms drawing; we finally discuss the overall architecture of the system. During the discussion we will try to motivate our design choices and why we have reached the final designed that seems the only possible so far.

A simple viewlet

Most of the magic provided by VSLab can be observed by looking at the following screenshot:



Although it may seem a deceptively simple interaction there are many things that are going on under the hood to provide this apparently well integrated interface. The following F# code has been highlighted in the editor window:

```
// how to write a simple viewlet..
type simpleViewlet() as x =
  inherit Viewlet() as base
  override x.OnPaint(e) =
    e.Graphics.DrawRectangle(Pens.Red, new Rectangle(10,10,50,50))

let simple = new simpleViewlet()
simple.Show()
```

An F# programmer can easily read the definition of a class inheriting from the *Viewlet* base class and the overriding of the *OnPaint* event handler typical of Windows Forms graphics programming. We can safely assume that the *Viewlet* class inherits from *Control* and that you can use the *Graphics* object to draw a red rectangle in the window. The next statement creates an instance of this new class and invokes the *Show* method which presumably shows the Visual Studio toolwindow that is visible in figure.

If you try this code snippet you will find that this is a true Visual Studio toolwindow which can be docked as any other. It may seem a little advantage from writing a standard Windows Form inside F# interactive but Visual Studio toolwindows use effectively the display by allowing a very flexible model of docking in which it possible to combine multiple windows into graphical control panels.

Now let's go even further and without restarting the F# interactive session let's change our viewlet definition and create another one:

```
// how to write a simple viewlet..
type simpleViewlet() as x =
  inherit Viewlet() as base
  let mutable draw = fun (g:Graphics) -> g.DrawRectangle(Pens.Red, new
Rectangle(10,10,50,50))

  member x.Draw
    with get () = draw
    and set (d) = draw <- d

  override x.OnPaint(e) =
    draw e.Graphics

let simple = new simpleViewlet()
simple.Show()

simple.Draw <- fun g -> g.DrawEllipse(Pens.Red, new Rectangle(10,10,50,50))
```

In this case we can evaluate interactively the code until the invocation of *Show*, and obtain a similar toolwindow. But now we can even redefine the drawing function without having to define a new class, and if we evaluate the assignment to the *Draw* property and refresh the toolwindow a red circle will magically appear. We have redefined the *simpleViewlet* type and two different toolwindows coexist together, and one of them with a parametric paint! The first time I saw it running I was amazed and impressed even if I was expecting it!

Now that we have introduced the example let's think about the interactions that take place under the hood to really appreciate the complexity that is hidden behind this natural way of coding. The state of the application, and therefore the state of the two instances of the two different definitions of *simpleViewlet*

are running inside F# interactive. If the task managed is started, an *fsi.exe* process would be listed among the processes, it is the process hosting the F# interactive session which interacts with the Visual Studio toolwindow using a communication based on inter-process streams like pipes. Streams are enough to transfer text and let output to be shown by the Visual Studio F# interactive toolwindow, but it is difficult to stream graphics primitive in an efficient way! Many F# interactive examples before VSLab were showing graphics based on Windows Forms simply by opening standalone forms, perhaps setting the *AlwaysOnTop* property to ensure its visibility on top of the Visual Studio window. This behavior was restricted to the F# interactive process, thus no communication with Visual Studio was necessary.

Let us assume that it is possible to cross the process barrier and communicate with Visual Studio to create the toolwindow (as VSLab does), who is responsible for drawing in the toolwindow? The traditional programming model would assume that Visual Studio would be in charge for drawing, but in the last example we have shown that it is even possible to redefine the paint function of a *viewlet*. How is it possible to marshal a function to Visual studio in an efficient way? How can we change the state of an object in F# interactive and pretend that the drawing is updated correctly?

The only possible answer is: it must be the F# interactive process that draws in the toolwindow. Since you are running VSLab you know it can be done, and it is an old Win32 magic that makes it possible: if you are able to obtain the handle of a window you can obtain a device context for drawing in it, even if this window does not belong to the process that performs the drawing. Now that I read it while I'm writing it looks so natural and obvious, it wasn't when we started the project and it took us more than a month to spot the actual solution.

Once we've got the intuition we simply looked for creating an empty toolwindow in Visual Studio and communicating to the F# counterpart the window handle so that it is the F# interactive that actually draws inside the toolwindow. Drawing, however, is not a whole graphical component; we needed events to react to paint messages, keystrokes and mouse interaction to obtain useful controls. We resorted to the message-oriented architecture of Win32 GDI to efficiently proxy events from the Visual Studio toolwindow back into the F# viewlet object.

Now that we have an intuition of what's going on behind the scenes we can briefly discusses the various areas we have used to create this interaction.

Visual Studio DTE

Most of the services of Visual Studio are delivered through the DTE which is a COM object model accessible through traditional Microsoft mechanisms. DTE is also accessible using managed applications, though this approach simply uses the interop abilities of CLR to invoke COM components. The managed API to Visual Studio allows defining addins, which are classes implementing COM interfaces (namely `IDTExtensibility2` and `IDTCommandTarget`) used by VS to notify about relevant events (such as loading/unloading and command invocation) to the addin class. In VSLab the *VSLabAddin* project defines the *Addin* class responsible for implementing these COM interfaces.

Through DTE interfaces (we are using DTE2) we can create named commands and extend the context menu with the *SendTo* option for code consolidation. Named commands can be associated with UI elements or simply treated as macros. We bind few of them to special key bindings in order to support the *Alt+Enter* and *Alt+'* combinations for evaluating a selected text or a single line using F# interactive. We also create the

menu entry for the context menu (to be shown only in the context of a VSLab project) and the toolbar with VSLab-specific commands and toolwindow management.

Another task accomplished by the addin is to create F# interactive toolwindow mimicking the code made by the standard F# toolwindow. We simply create the toolwindow specifying the type of the *UserControl* implementing the F# interactive and the communications with the *fsi.exe* process. The code simply uses the *CreateToolWindow2* method of the *Windows* property of DTE2 object model:

```
let loadVSLabFSI() =
  InitCtxtMenu()
  if not fsiLoaded then
    let mutable programmableObject:obj = null
    toolWnd <- (appObj.Windows :?> Windows2).CreateToolWindow2(
       addInInst, fetchFsBinDir() + @"\FSharp.VisualStudio.Session.dll",
       "Microsoft.FSharp.Compiler.VSFSITools.ToolWindow", "VSLab",
       "{AA8E793E-610F-4f8e-B51A-6B0A97507D3D}", &programmableObject)
    toolWnd.Visible <- true
    ml_send ("#light\n#use @\"" + (fetchVSLabDir()) + "vslabstartup.fsx\"")
    fsiLoaded <- true
```

Notice that we are specifying a GUID during the creation of the toolwindow: this will be the GUID identifying the toolwindow in the Visual Studio object model. We can use the *ml_send* function to the F# interactive session to send text as if it is coming from standard input by invoking the method on the toolwindow object returned by the *CreateToolWindow2* invocation. Using this function we send to *fsi.exe* the initial set of commands responsible for initializing the VSLab environment during startup.

It is handy to be able to send strings to be evaluated to *fsi.exe* since we can invoke core services of VSLab that are running in the remote process responsible for retaining most of the VSLab state. The *CreateToolWindow2* method plays also a fundamental role in the creation of viewlets: the control loaded inside the toolwindow allocated upon creation is responsible for communicating its own window handle to a peer control running in the *fsi.exe* process space. In practice this toolwindow is filled by primitives issued by the peer running in the F# interactive process, and it is responsible for forwarding all the relevant windowing events to it.

Note. In the first release *Kit* of VSLab it is possible to have a single instance running VSLab because the F# interactive peer of a toolwindow uses DTE and COM interop to communicate and COM selects the first running instance of Visual Studio. In the *Hal* release this limitation will be removed since we explicitly go through the COM Running Object Table (ROT) and get the DTE COM server associated with the PID of devenv hosting the running VSLab add-in. In *Kit* the exchange of window handles is performed using a socket connection that is temporarily established by the toolwindow and its peer, this will be likely converted in windows messages exchange in the future.

Win32 messaging

There are moments in life when being a little bit older comes handy, and my background of Win16 programmer helped me in designing the communication process between a toolwindow and its peer running the F# interactive process space.

As we have quickly discussed in the previous section the managed interface of the DTE allows creating toolwindows by specifying the type of a WinForms user control contained within a managed assembly. It seems to be natural to define *Viewlets* as special Windows Forms user controls capable of drawing on a

remote window instead than the window automatically created by GDI+. This choice allows to piggy back on the message dispatching code implemented in Windows Forms providing the illusion that the viewlet is simply an in-process control.

The static class *Viewlets* is initialized during startup and creates an hidden form that will contain all the peer controls associated with Visual Studio toolwindows, this is the *Init* method of this class:

```
static member Init() =
  _viewletsProxy <- new Form(Text="Viewlet Proxy")
  Application.DoEvents()
  _viewletsProxy.ShowInTaskbar <- false
  _viewletsProxy.FormBorderStyle <- FormBorderStyle.FixedToolWindow
  _viewletsProxy.Show()
  Application.DoEvents()
  _viewletsProxy.Width <- 1
  _viewletsProxy.Height <- 1
  _viewletsProxy.Top <- -100
  Application.DoEvents()
```

The form is conveniently hidden since it is just an implementation artifact. The base class *Viewlet* for viewlets automatically adds the control to the proxy form in order to force Windows Forms to create a window handle associated with the control. When the *Show* method is invoked the toolwindow is created and the handle of the hosted control (implemented in *VSLabCore* project) is exchanged with that of the peer.

Now the magic is performed by an old windowing technique commonly known as *subclassing*, which consist in changing the callback pointer of a GUI window (in the broader sense, not just a top level window of the window manager) and filter incoming messages before dispatching them to the original callback function. Subclassing can be done in Windows Forms using the *NativeWindow* class, though most of the time is enough to override the *WndProc* method inherited from *Control*.

The *ViewletToolWindow* class is the user control created in the Visual Studio toolwindow and upon creation it sends the handle to the viewlet peer running within the *fsi.exe* process. The responsibility of this control is to inform the peer whenever a repaint is needed and send all the relevant events to the peer.

The *SendMessage* function is defined using PInvoke declarations along with the *GetParent* function and is used to forward windowing messages:

```
[<DllImport("user32.dll")>]
extern int SendMessage(IntPtr hWnd, int Msg, IntPtr wParam, IntPtr lParam)

[<DllImport("user32.dll", ExactSpelling=true, CharSet=CharSet.Auto)>]
extern IntPtr GetParent(IntPtr hWnd)
```

The *WndProc* method overriding does the same trick for all the forwarded messages:

```
override x.WndProc(m) =
  base.WndProc(&m)

  if target <> IntPtr.Zero then
    if Enum.IsDefined(typeof<Messages>, m.Msg) then
      match enum(int(m.Msg)) with
      | Messages.WM_PAINT ->
        let encode a b = (a <<< 16) ||| (b &&& 0xFFFF)
        let lt = encode clip.Left clip.Top
        let wh = encode clip.Width clip.Height
```

```
          SendMessage(target, int(Messages.WM_APP_PAINT), new IntPtr(lt), new
IntPtr(wh)) |> ignore
        | Messages.WM_ERASEBKGND ->
          let encode a b = (a <<< 16) ||| (b &&& 0xFFFF)
          let lt = encode clip.Left clip.Top
          let wh = encode clip.Width clip.Height
          SendMessage(target, int(Messages.WM_APP_ERASEBKGND), new IntPtr(lt),
new IntPtr(wh)) |> ignore
        | _ ->
          SendMessage(target, m.Msg, m.WParam, m.LParam) |> ignore
```

Note that we simply forward all the messages defined by the *Messages* enumeration. Most of the messages are simply forwarded verbatim, with the noticeable exception of messages involving a device context. For *WM_PAINT* and *WM_ERASEBKGND* we need to send into application-defined messages (*WM_APP_PAINT* and *WM_APP_ERASEBKGND* respectively) since the device context cannot be sent safely to another process, it will responsibility of the peer to acquire a valid device context for drawing in the toolwindow given its window handle. The clipping rectangle is sent with these messages because it can't be obtained by the peer otherwise.

We used subclassing to detect when toolwindows are hidden and resumed. The Windows Forms library is supposed to inform with an event that the control visibility state is changing; unfortunately because of a well-known bug in the implementation there is no way to know when the control gets hidden with the toolwindow. We found two hacks to detect when the control is shown and hidden, we know it is fragile, but there is no official API for this. The *OnPaint* method is triggered whenever paint occurs, thus we can send an application-defined message to inform that the window is now visible:

```
  override x.OnPaint(g) =
    if not visible then
      visible <- true
      SendMessage(target, int(Messages.WM_APP_VISIBLECHANGED), new IntPtr(1),
IntPtr.Zero) |> ignore
    clip <- g.ClipRectangle
```

The opposite change is trickier to catch and we use subclassing as follows:

```
type internal ToolWindowSubclass() as x =
  inherit NativeWindow() as base

  let evt_fire, evt_listen = IEvent.create()

  member x.Activated = evt_listen

  override x.WndProc(m) =
    if (enum(m.Msg) = Messages.WM_SHOWWINDOW) then
      evt_fire(false)

    base.WndProc(&m)
```

This subclassing is meant to intercept the *WM_SHOWWINDOW* message of the toolwindow that contains our proxy control. We perform the subclassing during handle creation:

```
    parent <- GetParent(x.Handle)
    subclass.AssignHandle(parent)
    subclass.Activated.Add(fun v ->
      visible <- false
      if target <> IntPtr.Zero then
```

```
        SendMessage(target, (int Messages.WM_APP_VISIBLECHANGED), IntPtr.Zero,
IntPtr.Zero) |> ignore
    )
```

The *Viewlet* class performs the complementary operations by overriding the *WndProc* method. This is how the *WM_APP_PAINT* is handled:

```
    | Messages.WM_APP_PAINT ->
      use g = Graphics.FromHwnd(target)
      let decode a = ((a >>> 16) &&& 0xFFFF, a &&& 0xFFFF)
      let left, top = decode (m.WParam.ToInt32())
      let width, height = decode (m.LParam.ToInt32())
      let rect = new Rectangle(left, top, width, height)
      g.SetClip(new RectangleF(float32 left, float32 top, float32 width, float32
height))
      let pe = new PaintEventArgs(g, rect)
      x.OnPaint(pe)
```

The magic is performed by the *FromHwnd* method of the *Graphics* class, and then we simply dispatch the method by calling the appropriate event handler.

It is important to understand that with this approach there are two limitations:

- Only a subset of Win32 messages are routed to the peer
- The viewlet cannot contain children controls

The first issue is not real; if a message is required it can be easily sent to the proxy. As for the second we are working to a library of lightweight controls to be able to have children controls even in the toolwindows.

Visual Studio integration

VSLab integrates with Visual Studio by defining custom project and custom items. These extensions are fairly simple to do; we simply modify the .vsz files that contain project and item definitions as discussed in the VSIP documentation.

The C# custom actions project used by the installer contains all the information about the actions and it can be easily extended and modified. There is a generic class that gets specialized for the VSLab installation process.

Conclusions

VSLab core mechanisms are well hidden under the hood but are sophisticated and rely on many features of Windows and Visual Studio. The approach taken has shown to be very efficient and it also supports Direct3D graphics as witnessed by one of the example viewlets.

Although we are happy of being able to cross the barrier and fond of our solution we are not stickled to it. We are hoping that upcoming versions of Visual Studio will provide native support for out-of-process addins and toolwindows. The ultimate goal of VSLab was to show that one of the strength of F# is to lose the separation between top-level, interpretation and compilation which make F# ideal for those applications that benefit of interactive development and final consolidation possibly requiring execution performances not guaranteed by dynamic languages.