Visual LANSA v13.0

Adopting DirectX

February 2013

Adopting DirectX

Introduction

Visual LANSA version 13.0 introduces the DirectX rendering engine. DirectX is a collection of APIs embedded in to Windows that provide superior graphics capabilities.

Traditionally, the appearance of Windows applications was defined by relatively simple prefabricated controls such as TreeView, Push button and Grid. Whilst these controls provided the required functionality, and with use of Visual LANSA Themes and Visual Styles could produce aesthetically appealing applications, user interface styling in terms of both the appearance and operation, was fundamentally limited by their prefabricated nature.

However, the emergence of the internet and the typically more marketing-like appearance of websites, and latterly the rush towards touch devices and mobile computing, has helped fuel a change in developer behavior. The aesthetic of applications has become far more important, with the result that for some application requirements, predefined control appearance is no longer sufficient to allow developers to obtain the required results.

With DirectX rendering, these capabilities are now available. Visual LANSA v13.0 introduces a selection of new features including user designed controls, dynamic styles, mouse events, , transparency, gradient colors, animations, popup panels, and a few more besides.

However, when Visual LANSA was first released, these features weren't even dreamed of, and their inclusion in version 13 has caused some minor issues that all developers need to be aware of before they decide to adopt DirectX for their existing applications.

First and foremost, an existing application running in version 13 will continue to behave exactly as it did in version 12. Only by actively choosing to use the DirectX rendering option will the application change. This can be done at application, form or even panel level. For many users, this transition may be almost seamless. The application may look slightly different, but in terms of functionality, the behavior may well be identical. For some however, DirectX rendering may subtly impact the behavior of the existing application.

LANSA has gone to great lengths to ensure that "flicking the DirectX switch" is as simple and uneventful as possible, and that the user interface remains close to that of version 12. However, with such an array of new functionality and the restrictions imposed by the adoption of new underlying technologies, some change is inevitable. This document outlines some of the changes made to Visual LANSA. It explains how these may impact your existing applications and endeavors to provide simple changes that can be made to resolve any issues found.

Adopting DirectX Rendering

Whilst every effort has been made to ensure that as much of the Win32 appearance has been honored as possible, circumstances dictate that it is simply impossible to provide the new features and a DirectX runtime that precisely reflects that of Win32.

Before enabling for DirectX, consider the following. Your applications will continue to run in version 13 as they did in version 12. While DirectX offers a far greater flexibility of UI and many new features, it also comes at a cost. Some of the changes will affect the behavior of your application and may well require code review and modification, testing and all of the other typical tasks associated with software development. In short, moving an existing application to DirectX is not something that should be undertaken lightly.

While LANSA strongly recommends DirectX rendering for all new applications, as all future UI enhancements will be targeted at DirectX, moving an existing Win32 application to DirectX should be considered on case by case basis.

Enabling for DirectX

Visual LANSA allows DirectX to be applied at panel, form or application level. Once DirectX is enabled, all child panels will also use DirectX, specific Win32 controls notwithstanding e.g. ActiveX & graphs.

If you set the runtime to DirectX, all forms, panels and controls within the application will use DirectX rendering.

If you set a form to use DirectX, child panels and controls will use DirectX rendering.

If you set a panel to use DirectX, child controls will use DirectX rendering.

This simple act of setting a property or two belies the underlying complexity. The reality is that Win32 and DirectX don't really work that well together and you should seriously consider using one or the other.

Adoption Strategies

In practical terms, there are two strategies for the adoption of DirectX – Wholesale or piecemeal.

Piecemeal

This is the seemingly simpler strategy and offers a gentler introduction to DirectX. It allows individual forms or panels to start using DirectX related features e.g. new controls or styles, without affecting the remainder of the application.

However, Win32 and DirectX are very different technologies and while LANSA has greatly simplified their merging, the reality is that this solution is far from perfect. Developers may well find that integrating the old and the new does not come without some significant complications and concessions when it comes to the appearance of the application. For example, DirectX mandates the use of TrueType fonts and it may therefore be necessary to change the font for the remainder of the application to ensure consistency.

In practical terms, while simple additions can be made, there are numerous factors, mostly appearance based, that may mean that a better approach is simply to take the hit and use a wholesale approach.

Wholesale

The wholesale approach means that the runtime is switched to use DirectX and that as a result the entire application will render using DirectX.

Whilst this is certainly the more drastic of the two approaches, it may well turn out this it is also the cheaper of the two in the longer term. The initial impact is certainly larger and there will be issues to resolve, but once done all further work is in DirectX. Whereas, by adopting the piecemeal approach, all further development will need to manage the two different underlying technologies and that must lead to increased development costs.

Test, Test, Test

Regardless of the strategy employed to start using DirectX, the majority of applications may well behave slightly differently. For some it will be as subtle as a change in font and text not fitting as it did before; for others, parts of the application will not behave quite the same, e.g. mouse over events on a list causing changes to field values, and this may cause runtime errors.

As stated previously, LANSA has gone to great lengths to ensure that “flicking the DirectX switch” is as simple and uneventful as possible. However, LANSA cannot guarantee that applications will continue to work as before and it is strongly recommended that you perform a full test of your applications before enabling any productions systems.

DirectX Changes

The following sections detail many of the individual changes and explain the reasons for the change and how they may impact existing applications. Where possible, workarounds and simple ways in which these issues can be overcome are specified.

Remember, these situations can only occur if you specifically choose to run with DirectX rendering.

ComponentVersion

Software bugs are inevitable, but for the most part they can be worked around and fixed in the longer term. However, some bugs are subtle and aren't immediately recognised as being bugs. The behaviour seems reasonable enough and the user simply uses the product as they find it.

However, this user reliance on a what is really a flawed product creates a problem when looking to fix bugs and introduce new features that may not work well with the existing software. We cannot simply fix a control or start making things work as they should for the simple reason that existing applications could well look and behave differently after an upgrade.

To allow such changes to be affected, the ComponentVersion property is used. When it is no longer feasible to fit the old in with the new, a new ComponentVersion is made. Existing application code will still use the default version 0, while any new controls dropped on a form will automatically be given the latest version number.

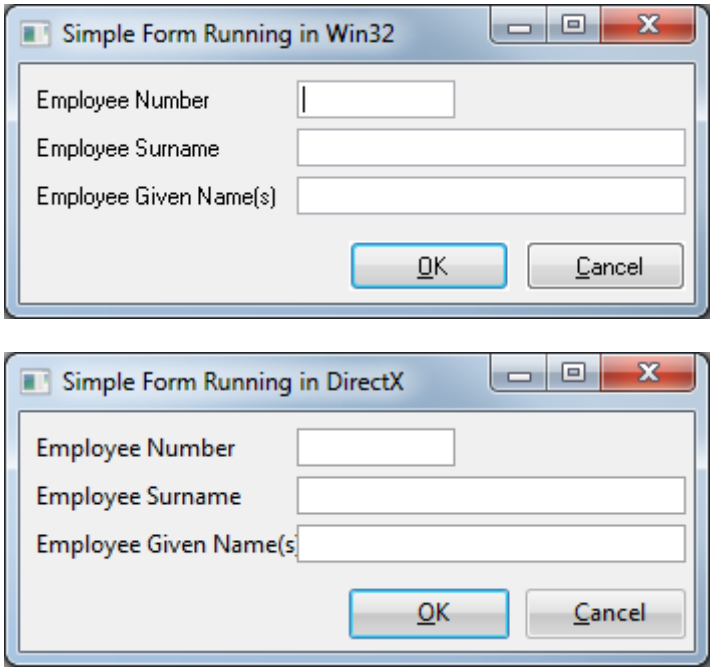
```
Define_Com Class(#prim_trvw) Name(#Tree) ComponentVersion(2)
```

Most controls are still using ComponentVersion(0). However, the controls listed below have later versions. To get the most out of DirectX it is recommended that you use the latest ComponentVersion for these controls.

Control	Primitive	Version
Combo Box	Prim_CMBX	1
Field	Prim_EVEF	1
Grid	Prim_GRID	1
Form	Prim_FORM	1
List View	Prim_LTVW	2
List Box	Prim_LTBX	1
Tab	Prim_TAB	1
Toolbar Button	Prim_SPBN	1

Default Appearance

Below are images of the same form running firstly as it would appear in version 13 using Win32 and secondly as DirectX.



The code for this form is available in the Sample Source section of this document.

Functionally, the two forms are identical. However, the font used for the text is different. Win32 defaults to MS Sans Serif 9 while Direct X defaults to Segoe UI 9.

Ms Sans Serif is an old font and was created a long time ago when screens were much smaller and had much lower resolutions. The result is that on a modern screen, running a modern resolution, it looks rather "blocky" compared to the smooth edges and nicely rounded corners of a modern True Type font using antialiasing.

For most users the change of font may well be of no consequence. However, Segoe UI is slightly wider, and as can be seen in the two images, and this may cause some text to wrap, show ellipses or be truncated.

Blending Win32 & DirectX

There are compatibility issues when trying to work with both Win32 and DirectX in the same UI space. As with fonts, this is a reflection of the underlying technology. The simple explanation is that when using the two technologies, there are essentially two different UI streams that are both unaware of each other. A more detailed explanation is available at http://msdn.microsoft.com/en-us/library/aa970688.aspx.

Those customers who take a piecemeal approach to the adoption of DirectX and perhaps use it for additional functionality such as new dialogs or new panels in existing forms etc., will have

to deal with two different technologies working together. At a low level, blending DirectX and Win32 is far from simple, and while DirectX can exist inside Win32, and Win32 can exist inside DirectX, but both have issues that need to be overcome, and this can result in behaviour that is neither expected nor desirable.

Clipping

Clipping is the term used to describe how the edges of child controls that extend beyond their parent control are hidden.

Win32 controls cannot occupy part of the same render level as DirectX and are therefore placed on a different level. This results in a situation where child Win32 controls that are bigger than their parent will cause scrolling issues. In the image below, the browser is parented to a panel which has been scrolled. See how the top of the browser coincides with the top of the panel scroll bar on the right. As this is a completely Win32 application, the browser is correctly clipped as there is only one UI stream.

The code for this form is available in the Sample Source section of this document.

However, in the same form running as DirectX (below), scrolling the panel causes the ActiveX to move, but not to clip, because there are now 2 UI streams. See how the browser is now above the panel scrollbar on the right.

The only practical solution to this issue is to ensure that the Win32 control is sized appropriately, probably by use of a layout manager, and does not exceed the size of its parent.

ActiveX

ActiveX deserves a special mention. ActiveX controls by definition are Win32. They must therefore exist in a different UI stream and they will never be able to follow the styles and appearance used in a DirectX application.

Historically, it has been a fairly common practice to embed controls such as a browser or a PDF viewer within applications to show web pages or documents. Whilst this is still possible, it is recommended that an alternative approach be taken. Rather than using the control within the application, make a system call using the System_Command BIF and let the operating system work out what best to do with the specified object.

The alternative to this is to use an equivalent .Net Component for the same task. However, it's worthy of note that .Net objects are often just the same ActiveX controls with a new wrapper, and are in practical terms, little or no different to the original.

DisplayPosition

In a scenario where a Win32 control is on top of another, the disconnection of the two UI streams mean that it is no longer possible to use DisplayPosition to bring one to the front. As far as VL is concerned, the DisplayPosition is correct, but like clipping, the Win32 control is unaware of it surroundings and simply displays as defined.

The simplest is not to rely on DisplayPosition, but instead set all but the uppermost control to Visible(False). This has an additional functional benefit in that it stops a Tab going to the "hidden" panels.

Screen Design

The layering of Win32 controls also affects the IDE. This is a DirectX application as well, so the rendering of Win32 controls in the designer can be problematic.

The Designer will position the control correctly. However, because it is floating above its parent and is not part of the UI in the same way that DirectX controls are, a Win32 control can obscure the grabbers making it impossible to resize the control with the mouse.

A simple work around for this is to briefly turn the designer to Win32 mode.

As at the time of release of this document, Memo (Prim_memo), Graph (Prim_grph) and Property Sheet are still Win32 controls.

Themes and Visual Styles

When running as Win32, Visual LANSA plays lots of games under the covers to ensure that the styles and appearance of controls conform to the specified themes. A good example of this is the appearance of panels. A panel by default is gray, but when running in a themed application it can appear blue or as a toolbar with a gradient color. Any panels parented to that panel automatically adopt the toolbar appearance.

However, should the attached panel be flagged as DirectX, it is no longer subject to the same strict implementation of themes. The runtime does its best to pick up a color from the theme, but it is at best a guess. The reason for this is that the application has crossed over from Win32 to DirectX and the two parts of the application, while appearing as though they are one, are actually separate streams.

This presents an issue when adopting DirectX piecemeal as panels on top of toolbar themed panels simply cannot display the toolbar effect.

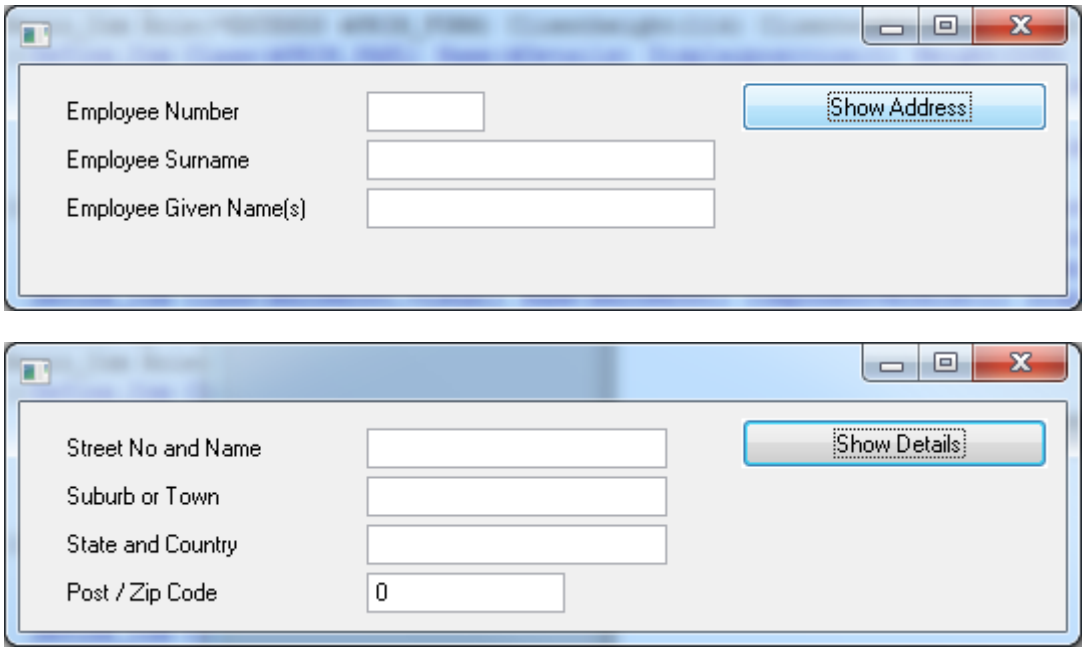
Transparency and Opacity

Transparency and opacity work seamlessly in DirectX, but cannot be employed to see through a DirectX panel to an underlying Win32 environment. As with styles and themes, the DirectX panel is wholly unaware of the Win32 controls that might be beneath it and will appear black.

Transparency and Opacity

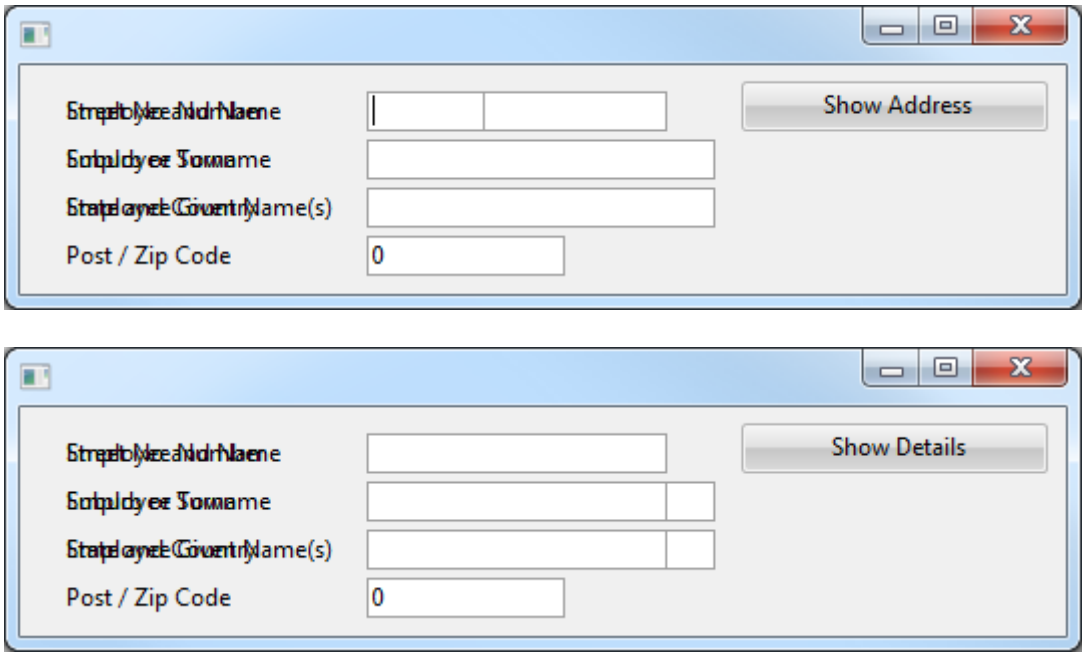
DirectX rendering introduces transparency and opacity. By default in DirectX all panels and labels are considered transparent unless a specific Style has been applied. This new appearance can lead to issues.

Below, a simple form toggles between address and employee details. When the button is clicked, the address details are enabled and brought to the front.



The code for this form is available in the Sample Source section of this document.

However, the results with DirectX rendering are somewhat different.



Regardless of the DisplayPosition of the Address and Details panels, both are plainly visible.

The need for this default stems from the desire to build complex layered forms and to still be able to see watermark images or backgrounds applied to it. If they were opaque, it would be necessary to visit every panel and label and specifically apply a transparent style.

A simple work around for this situation is to set the inactive panel to Visible(False) rather than Enabled(False).

Routed or Bubbled Events

To simplify the coding of complex reusable parts and in particular the design of panels for the new User Designed Controls (UDC), an event detected on a control is now passed up the parent chain.

Typically for UDC, the panels displayed are constructed of little more than labels and images. However, with the existing event processing, each of the labels would take the click event and not pass it on. The result would be that the user would need to code every click event for every child control. By sending the event up the parent chain, coding is greatly simplified.

Of course, it may be necessary to know which of the controls actually fired the event initially. The EVTROUTINE command already has the Com_Sender selector, but this only ever reports the control firing the event in the context of the controls referenced on the EVTRoutine command. As a result, the Origin selector has been added. Regardless of how many layers of parent are used, Origin will contain a reference to the instance on which the event was actually started.

```
Evtroutine Handling(#Com_owner.Click) Origin(#Origin)
…
Endroutine
```

Clearly though, this change of event behavior may have some side effects. In a simple example where there is a click event for both child and parent components, in Win32 the two events would remain separate. However, with DirectX processing and event routing a click on the child would result in both the child click and parent click firing. This is an unusual situation and it is unlikely that many customers will encounter it, but nevertheless it is conceivable and should be catered for.

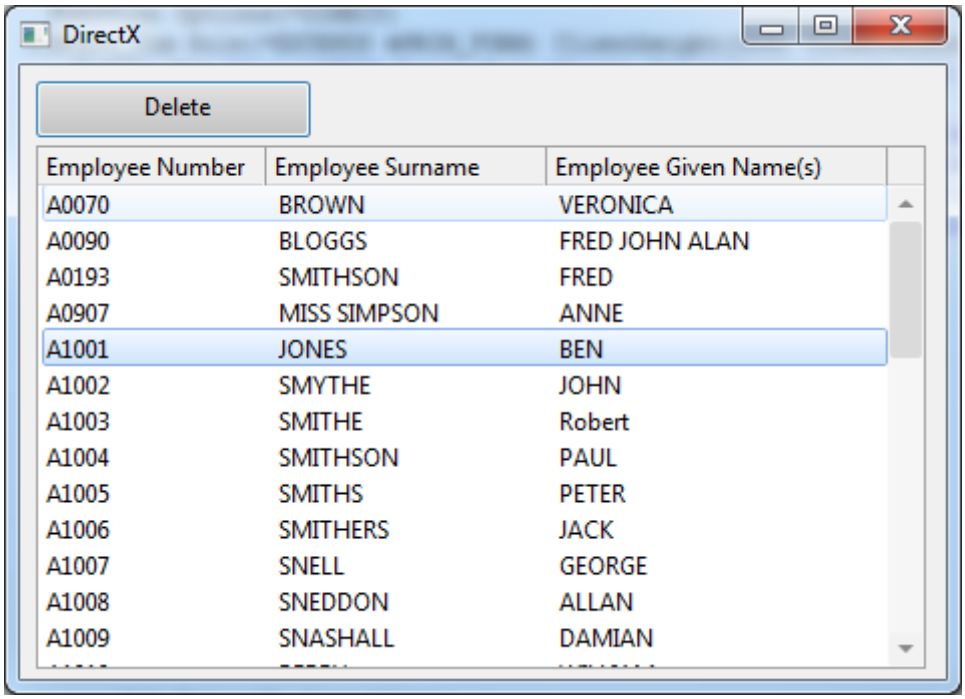
To counter unwanted event propagation, the Handled selector has been added to EVTROUTINE. By setting Handled to true the event is no longer passed beyond the routine being processed.

```
Evtroutine Handling(#Button.Click) Handled(#Handled)

* Stop the event going any further up the parent chain.
#Handled := True

Endroutine
```

Mouse Events, LANSA Lists and CurrentItem

When an application is using DirectX, there are changes to the way in which the list controls (Tree, Grid etc.) appear. Most notably, when the mouse is over an item in the list, the item will be highlighted. In the image below, the 5th item in the list is the FocusItem, i.e. the last item clicked. The 1st item has the mouse over it and is therefore the CurrentItem.

Adopting DirectX



The code for this form is available in the Sample Source section of this document.

All LANSA lists use a CurrentItem concept that maps the equivalent field values from the list and into the equivalent variables. CurrentItem effectively represents the item last processed in the list. This might be the last item clicked, which will also be the FocusItem, or perhaps the last item processed in a Selectlist loop.

Historically, it was common to see processing similar to the image above where the Delete button would cause the deletion of the currently selected item with code similar to the following.

```
Evtroutine Handling(#Delete.Click)

If (#List.CurrentItem *IsNot *null)
Dlt_Entry Number(#List.CurrentItem.Entry) From_List(#List)
Endif

Endroutine
```

This code effectively assumes the CurrentItem and FocusItem are going to be one and the same, and for many scenarios prior to DirectX that would be the case. However, while this may have worked, it wasn't a failsafe mechanism and the reliance on CurrentItem, whilst commonplace, was not best practice.

Mouseover processing in DirectX follows the same rules as all other list based mouse events e.g. ItemGotFocus, ItemGotSelection etc. As soon as the mouse interacts with an item it becomes the CurrentItem. This in turn updates the field values associated with the list.

The result is that it is no longer acceptable to rely on CurrentItem and the existing field values as these can easily be changed by mouse movements. In the previous image, to move from the FocusItem to the Delete button requires moving the mouse over the four items above it.

The last one touched will be the first item, so deleting CurrentItem will therefore delete the first item.

Clicking a popup menu item is similarly affected. As soon as the popup menu closes a mouse event is detected by the list beneath it. Somewhat counter intuitively, this event will precede the click event for the popup menu item.

When running in DirectX it is best practice to strictly adhere to the use of FocusItem. Further, if the code relies on field values being correct at the moment of processing it is prudent to ensure that the code updates the fields from the FocusItem by use of Get_Entry immediately prior to any processing.

```
Evtroutine Handling(#Delete.Click)

If (#List.FocusItem *IsNot *null)

Dlt_Entry Number(#List.FocusItem.Entry) From_List(#List)

Endif

Endroutine
```

True Type Fonts

DirectX rendering only supports True Type fonts. This is simply a reflection of the underlying Microsoft technologies. True Type and Open Type, an extension of True Type, are industry standards and designed to render smoothly regardless of the font size used.

Where a font cannot be rendered, Visual LANSA uses Segoe UI.

Fonts such as MS Sans Serif, which is not True Type, typically have modern True Type alternatives. The MS Sans Serif equivalent is Microsoft Sans Serif.

If you intend to adopt DirectX, it is strongly recommended that you change your application to use a True Type font. This may cause issues with text no longer fitting in the available space and it is recommended that you review any changes you have made.

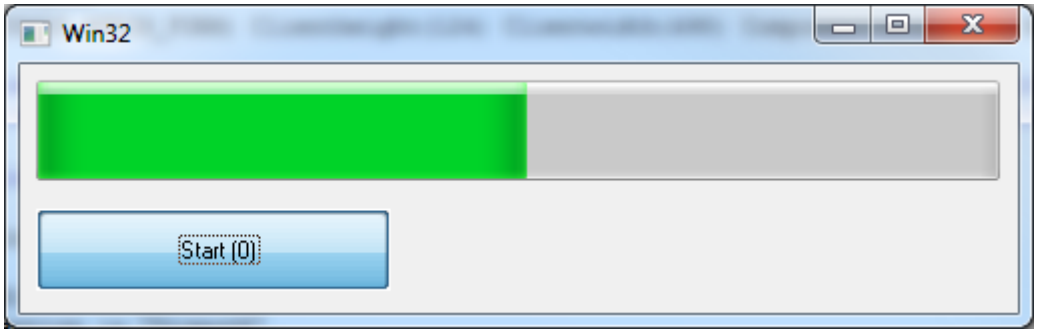
UpdateDisplay

In Win32 the UpdateDisplay method could be called on a control to force the screen to refresh during a long running process. The Win32 runtime was able to address individual controls specifically and in effect could update a small portion of the UI.

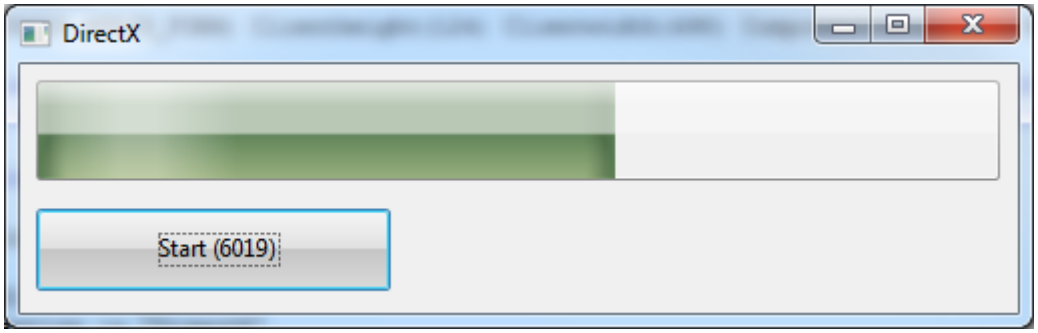
However, the DirectX runtime works in a different manner and this is no longer possible. UpdateDisplay will cause the whole of the form to update.

In most circumstances this will be of little consequence. However, in situations where UpdateDisplay is called repeatedly, this will cause noticeable performance degradation.

A typical situation where that occurs is when a Progress Bar is used. Progress Bars automatically use UpdateDisplay to ensure that they reflect their latest value. In the example below a simple loop is executed and the progress bar and start button caption are updated every iteration.



In Win32 above, the start button is not updated. The UpdateDisplay is specific to the Progress Bar. However, in DirectX, the whole form gets updated.



The code for this form is available in the Sample Source section of this document.

To counteract this situation, rather than updating the progress bar or specifically executing UpdateDisplay every iteration, a simple test can be added so that the update only occurs every 10th time.

Samples Source

Default Appearance

```
Function Options(*DIRECT)
Begin_Com Role(*EXTENDS #PRIM_FORM) Clientheight(119) Clientwidth(336)
Componentversion(1) Height(157) Left(175) Top(215) Width(352)
Define_Com Class(#EMPNO.Visual) Name(#EMPNO) Componentversion(1) Displayposition(1)
Left(8) Marginleft(130) Parent(#COM_OWNER) Tabposition(1) Top(8)
Define_Com Class(#SURNAME.Visual) Name(#SURNAME) Componentversion(1) Displayposition(2)
Left(8) Marginleft(130) Parent(#COM_OWNER) Tabposition(2) Top(32) Width(321)
Define_Com Class(#GIVENAME.Visual) Name(#GIVENAME) Componentversion(1)
Displayposition(3) Left(8) Marginleft(130) Parent(#COM_OWNER) Tabposition(3) Top(56)
Width(321)
Define_Com Class(#PRIM_PHBN) Name(#OK) Buttondefault(True) Caption('&OK')
Displayposition(4) Left(164) Parent(#COM_OWNER) Tabposition(4) Top(88)
Define_Com Class(#PRIM_PHBN) Name(#Cancel) Buttoncancel(True) Caption('&Cancel')
Displayposition(5) Left(252) Parent(#COM_OWNER) Tabposition(5) Top(88)
Evtroutine Handling(#Com_owner.CreateInstance)
Case (#sys_appln.RenderStyle)

When (= DirectX)
#Com_owner.Caption := "DirectX"

When (= Win32)
#Com_owner.Caption := "Win32"

Endcase
Endroutine

End_Com
```

Win32 & DirectX (ActiveX and Clipping)

```
Function Options(*DIRECT)

Begin_Com Role(*EXTENDS #PRIM_FORM) Clientheight(701) Clientwidth(965)
Componentversion(1) Height(739) Left(189) Top(211) Width(981)

Define_Com Class(#PRIM_PANL) Name(#Panel) Displayposition(1) Height(481)
Horizontalscroll(True) Layoutmanager(#Layout) Left(80) Parent(#COM_OWNER) Tabposition(1)
Tabstop(False) Top(56) Verticalscroll(True) Width(793)

Define_Com Class(#VA_WEBCTL.WebBrowser) Name(#Browser) Displayposition(1) Height(650)
Left(0) Parent(#Panel) Tabposition(1) Top(0) Width(775)

Define_Com Class(#PRIM_ATLM) Name(#Layout)

Define_Com Class(#PRIM_ATLI) Name(#LayoutItem) Attachment(Top) Manage(#Browser)
Parent(#Layout)

Evtroutine Handling(#Com_owner.CreateInstance)

Case (#sys_appln.RenderStyle)

When (= DirectX)
```

```
#Com_owner.Caption := "DirectX"

When (= Win32)

#Com_owner.Caption := "Win32"

Endcase

Endroutine

Evtroutine Handling(#Com_owner.initialize)

#Browser.Navigate( www.lansa.com )

Endroutine

End_Com
```

Transparency and Opacity

```
Function Options(*DIRECT)
Begin_Com Role(*EXTENDS #PRIM_FORM) Clientheight(114) Clientwidth(522) Height(152)
Left(106) Top(204) Width(538)
Define_Com Class(#PRIM_PANL) Name(#Details) Displayposition(1) Height(108) Left(15)
Parent(#COM_OWNER) Tabposition(2) Tabstop(False) Top(13) Width(338)
Define_Com Class(#PRIM_PANL) Name(#Address) Displayposition(2) Enabled(False)
Height(108) Left(15) Parent(#COM_OWNER) Tabposition(1) Tabstop(False) Top(13) Width(338)

Define_Com Class(#PRIM_PHBN) Name(#MoveToFront) Caption('Show Address')
Displayposition(3) Left(360) Parent(#COM_OWNER) Tabposition(3) Top(8) Width(153)

Define_Com Class(#EMPNO.Visual) Name(#EMPNO) Componentversion(1) Displayposition(1)
Height(20) Left(8) Parent(#Details) Tabposition(1)
Define_Com Class(#SURNAME.Visual) Name(#SURNAME) Componentversion(1) Displayposition(2)
Height(20) Left(8) Parent(#Details) Tabposition(2) Top(24) Width(321)
Define_Com Class(#GIVENAME.Visual) Name(#GIVENAME) Componentversion(1)
Displayposition(3) Height(20) Left(8) Parent(#Details) Tabposition(3) Top(48) Width(321)
Define_Com Class(#ADDRESS1.Visual) Name(#ADDRESS1) Componentversion(1)
Displayposition(1) Height(20) Left(8) Parent(#Address) Tabposition(1) Width(300)
Define_Com Class(#ADDRESS2.Visual) Name(#ADDRESS2) Componentversion(1)
Displayposition(2) Height(20) Left(8) Parent(#Address) Tabposition(2) Top(24)
Usepicklist(False) Width(300)
Define_Com Class(#ADDRESS3.Visual) Name(#ADDRESS3) Componentversion(1)
Displayposition(3) Height(20) Left(8) Parent(#Address) Tabposition(3) Top(48) Width(300)
Define_Com Class(#POSTCODE.Visual) Name(#POSTCODE) Componentversion(1)
Displayposition(4) Height(20) Left(8) Parent(#Address) Tabposition(4) Top(72)
Usepicklist(False) Width(249)

Evtroutine Handling(#MoveToFront.Click)

If (#Details.DisplayPosition <> 1)

#Details.DisplayPosition := 1
#Details.enabled := True
#Address.enabled := False
#MoveToFront.Caption := "Show Address"
```

```
Else

#Address.DisplayPosition := 1
#Details.enabled := False
#Address.enabled := True
#MoveToFront.Caption := "Show Details"

Endif

Endroutine

End_Com
```

Mouse Events

```
Function Options(*DIRECT)
Begin_Com Role(*EXTENDS #PRIM_FORM) Clientheight(306) Clientwidth(462)
Componentversion(1) Height(344) Left(103) Top(200) Width(478)
Define_Com Class(#PRIM_Trvw) Name(#List) Columnbuttonheight(19) Componentversion(2)
Displayposition(1) Fullrowselect(True) Haslines(False) Height(261)
Keyboardpositioning(SortColumn) Left(8) Linesatroot(False) Parent(#COM_OWNER)
Tabposition(1) Top(40) Viewstyle(UnLevelled) Width(444)

Define_Com Class(#PRIM_TVCL) Name(#TVCL_1) Displayposition(1) Level(1) Parent(#List)
Source(#EMPNO) Width(27)
Define_Com Class(#PRIM_TVCL) Name(#TVCL_2) Displayposition(2) Level(2) Parent(#List)
Source(#SURNAME) Width(33)
Define_Com Class(#PRIM_TVCL) Name(#TVCL_3) Displayposition(3) Level(3) Parent(#List)
Source(#GIVENAME) Width(40)
Define_Com Class(#PRIM_SPBN) Name(#Delete) Caption('Delete') Displayposition(2) Left(8)
Parent(#COM_OWNER) Tabposition(2) Top(8) Width(137)

Evtroutine Handling(#Com_owner.CreateInstance)

Case (#sys_appln.RenderStyle)

When (= DirectX)
#Com_owner.Caption := "DirectX"

When (= Win32)
#Com_owner.Caption := "Win32"

Endcase

Select Fields(#List) From_File(pslmst)

Add_Entry To_List(#List)

Endselect

Endroutine

Evtroutine Handling(#Delete.Click)
```

```
If (#List.CurrentItem *IsNot *null)

Dlt_Entry Number(#List.CurrentItem.Entry) From_List(#List)

Endif

Endroutine

End_Com
```

UpdateDisplay

```
Function Options(*DIRECT)
Begin_Com Role(*EXTENDS #PRIM_FORM) Clientheight(124) Clientwidth(498)
Componentversion(1) Height(162) Left(261) Top(195) Width(514)

Define_Com Class(#PRIM_PGBR) Name(#ProgressBar) Displayposition(1) Left(8)
Maximumvalue(10000) Minimumvalue(0) Parent(#COM_OWNER) Tabposition(1) Top(8) Value(1)
Width(481)
Define_Com Class(#PRIM_PHBN) Name(#Start) Caption('Start (0)') Displayposition(2)
Height(41) Left(8) Parent(#COM_OWNER) Tabposition(2) Top(72) Width(177)

Evtroutine Handling(#Com_owner.CreateInstance)

Case (#sys_appln.RenderStyle)

When (= DirectX)
#Com_owner.Caption := "DirectX"

When (= Win32)
#Com_owner.Caption := "Win32"

Endcase

Endroutine

Evtroutine Handling(#Start.Click)

#ProgressBar.value := 0

Begin_Loop To(10000)

#ProgressBar.value += 1

#Start.Caption := ("Start (&1)").Substitute( #ProgressBar.Value.Asstring )

End_Loop

Endroutine

End_Com
```