Document: Tutorial and Tips & Tricks

Date: 11-08-98

Version: 1.1

Modifications since version 1.1

None…

Modifications since version 1.0:

Windowed Direct Draw

Overview

The EasyX control is an ActiveX control, which enables the programmer to tap into the realm of DirectX. The control is specifically designed to be used with Visual Basic, but should be compatible with any environment that supports ActiveX technology.
This document is divided into two sections, the first being a tutorial on the use of the control, and the second section is a Tips & tricks section on developing with the control. For other samples on using the EasyX control refer to the 6 undocumented samples and the Pong game supplied with the package.

Tutorials

This section of covers two basic tutorials on using the EasyX control:

Tutorial 1: Using EasyX's DirectDraw functions

Tutorial 2: Using EasyX's DirectSound functions

Tutorial 1

This tutorial demonstrates the use of the following functions:

InitDirectDraw
LoadBitmapFile
FillSurface
FlipSurface
MakeSprite
DrawSprite

Setting up

Start a new Visual Basic project (Standard exe) and ad an EasyX control to it.
(or you could just open the tutorial1 in the tutorials folder)

Before any functions are called, there is one thing that must be set, the *.Window* property of the control. This property is takes a hWnd parameter of a standard window. So the first thing to do, after declaring some variables, is to set this parameter in the Form_Load() procedure:

```
EasyX1.Window = Me.hWnd
```

The next thing to do is to initialize DirectDraw. This step will set the screen properties and display the DirectX primary surface. It is very important to check the return value from this function, since it is the only indication on whether DirectDraw or the screen mode is supported. In this application only a 8 bit color depth mode is needed, so we specify 8 bit which is 256 color mode.

Here we check it against the EX_OK constant, which is defined in the EX_CONSTANTS module. If the return value is not EX_OK, then a different mode is tried and so on. If none of the modes are initialized successfully we notify the user and quit. The reason for this could be either that the display card does not support DirectX or that an older version of DirectX is installed (remember that EasyX requires DirectX 5.0).

```
rt = EasyX1.InitDirectDraw(320, 200, 8)

If rt <> EX_OK Then
    'try a different format
    rt = EasyX1.InitDirectDraw(640, 480, 8)

    If rt <> EX_OK Then 'hmmm wrong again..try another
        rt = EasyX1.InitDirectDraw(800, 600, 8)

        If rt <> EX_OK Then 'that´s it..no more..
            MsgBox "Direct draw could not initializee", vbOKOnly, "Failure"
            Exit Sub
        End If
    End If
End If
```

After initializing the screen mode, it is time to load the surfaces. In this demonstration two simple surfaces are created with the LoadBitmapFile function. We specify the bitmap file name, and the color key (-1). This value ensures that the upper-left pixel in the bitmap are the transparent color, which is not blitted on the back buffer doing a DrawSprite operation. Since this mode is 8 bit, any other value specified as the color key would be an index into the palette associated with the surface.
After loading the bitmap, a check of the return value is done. If the return value is less than 0, then an error has occurred and the user is notified.

```
'load the surfaces
AppPath = App.Path & "\"

SurfaceYellow = EasyX1.LoadBitmapFile(AppPath & "easyx.bmp", -1)

If SurfaceYellow < 0 Then
   MsgBox "Could not load graphics" & vbCrLf & "Make sure that the graphic files
   are there",_  vbOKOnly, "Failure"
    EasyX1.EndDirectX
    Unload Me
    Exit Sub
End If
```

```
SurfaceBlack = EasyX1.LoadBitmapFile(AppPath & "easyx1.bmp", -1)

If SurfaceBlack < 0 Then
    MsgBox "Could not load graphics" & vbCrLf & "Make sure that the graphic
files are there", vbOKOnly, "Failure"
    EasyX1.EndDirectX
    Unload Me
    Exit Sub
End If
```

Now that the surfaces are loaded, it is time to set up the sprites, which are used to draw onto the back buffer. Sprites are created as an area on a given surface. A Sprite created with the MakeSprite function is not physically an image, but instead an area upon the given surface, which will be drawn when the function DrawSprite is called.
Two sprites are defined in this application, one for each surface. Each sprite is defined as the whole surface. But you could easily use any given size for a sprite, as long as they are within the dimensions of the surface.

```
'make the sprites
SpriteYellow = EasyX1.MakeSprite(0, 0, 100, 50, SurfaceYellow)

SpriteBlack = EasyX1.MakeSprite(0, 0, 100, 50, SurfaceBlack)
```

Now everything that is needed is set up, the back buffer is filled with black, using the FillSurface function. 0 is used as the index to the surface palette, since index 0 is almost always black (and index 255 is almost always white). Lastly the back buffer is flipped, so the pure black screen is shown.

```
'color the background black
EasyX1.FillSurface 0, EX_PRIMARYSURFACE
'and flip the color to make it visible
EasyX1.FlipSurface
```

Everything is now loaded and ready for action. All the action takes place in the in Form_KeyDown event.

The first things to check is whether the user pressed the escape key, if so end the DirectDraw session and program.

```
If KeyCode = vbKeyEscape Then
    EasyX1.EndDirectX
    Unload Me
End If
```

If the user presses the ‘A’ key the sprite defined as *SpriteBlack* will be drawn onto the back buffer. This is done in three steps: first clear the buffer with the FillSurface function (white), then draw the actual function, using the DrawSprite function and then flip the back buffer onto to the primary surface. Pretty simple. One thing to note though, is the DrawSprite function. The first two arguments specifies the upper left corner of the sprite, the next two specify the width and height of the sprite. In this example they are equal to the dimensions of the sprite, but the values could actually be any value greater than 0. The function will stretch the sprite to fit the new dimensions.
The same routine is used if the user presses the ‘B’ key, except that the SpriteYellow is used.

```
If KeyCode = vbKeyA Then
    'fill the surface first
    EasyX1.FillSurface 255, EX_PRIMARYSURFACE
    'draw the sprite
    EasyX1.DrawSprite 100, 100, 100, 50, SpriteBlack
    'flip it
    EasyX1.FlipSurface
End If

If KeyCode = vbKeyB Then
    'fill the surface first
    EasyX1.FillSurface 255, EX_PRIMARYSURFACE
    'draw the sprite
    EasyX1.DrawSprite 100, 100, 100, 50, SpriteYellow
    'flip it
    EasyX1.FlipSurface
End If
```

If the user presses the 'C' key, both the sprite will be drawn using the same routines as above, except that the SpriteYellow sprite will be stretched to double size, in order to demonstrate the stretching capabilities of the DrawSprite function.

```
If KeyCode = vbKeyC Then
    'fill the surface first
    EasyX1.FillSurface 255, EX_PRIMARYSURFACE
    'draw the sprite
    EasyX1.DrawSprite 100, 100, 100, 50, SpriteBlack
    EasyX1.DrawSprite 0, 0, 200, 100, SpriteYellow
    'flip it
    EasyX1.FlipSurface
End If
```

If the user presses the 'D' key, the surface is filled with palette index entry 30 and flipped onto the primary surface.

```
If KeyCode = vbKeyD Then
    'fill the surface first
    EasyX1.FillSurface 30, EX_PRIMARYSURFACE
    'flip it
    EasyX1.FlipSurface
End If
```

Tutorial 2

This tutorial will demonstrate the use of EasyX's DirectSound functions. The following functions will be used:

CreateStaticSound
CreateStreamingSound
PlayStaticSound
PlayStreamingSound
SetStaticPan
SetStreamingPan
SetStaticVolume

SetStreamingVolume

Open up the tutorial2 project. As you can see there are five command buttons on the form. Three of these are dealing with the static sounds, and the other two are dealing with the streaming sound. But before any sounds are played, an initialization of DirectSound is needed. This is done in the Form_Load procedure:

First the Window property of the control is set. (Remember this)

Then we call and check the initialization of DirectSound:

```
'''do not forget this
EasyX1.Window = Me.hWnd
''''''''''''''''''''''''

'initialize sound
rt = EasyX1.InitializeSound()

If rt <> EX_OK Then
    MsgBox "Sound could not initialize", vbOKOnly
    Unload Me
    Exit Sub
End If
```

As usual we end the application if an error occurred.

The next thing to do is to load the sounds. First sound to be loaded is the Static sound and then the streaming sound. As usual the return values (actually the sound indexes) are checked and if any error has occurred, the user is notified.

```
AppPath = App.Path & "\"

'load sounds
StaticSound = EasyX1.CreateStaticSound(AppPath & "boink.wav")

If StaticSound < 0 Then
    MsgBox "Static sound could not be loaded", vbOKOnly
    cmdStatic.Enabled = False
End If

StreamSound = EasyX1.CreateStreamingSound(AppPath & "countdown.wav")

If StreamSound < 0 Then
    MsgBox "Static sound could not be loaded", vbOKOnly
    cmdStream.Enabled = False
End If
```

Everything is now loaded and ready for use. All the action happens in the command buttons.

The first (cmdStatic) plays the static sound in a non looping manner, by using the PlayStaticSound function and supplying the sound index (StaticSound).

```
Private Sub cmdStatic_Click()

EasyX1.PlayStaticSound StaticSound, 0
```

```
End Sub
```

If you want to play the *boink* sound twice at the same time, you have two options, either load it anew or create a duplicate. You should always choose the duplicate option, since this makes use of the existing sound resource, instead of loading the sound again which takes up new resources. To duplicate a sound use the DuplicateStaticSound function.(only static sounds can be duplicated)

```
Private Sub cmdStaticDuplicate_Click()

StaticDuplicate = EasyX1.DuplicateStaticSound(StaticSound)

cmdStaticPlayDup.Enabled = True

End Sub
```

To play the newly duplicated sound, simply use the PlayStaticSound function, supplying the index of the duplicated sound.

```
Private Sub cmdStaticPlayDup_Click()

EasyX1.PlayStaticSound StaticDuplicate, 0

End Sub
```

To play the streaming sound, use the function PlayStreamingSound, as is done in the cmdStream click procedure.

```
Private Sub cmdStream_Click()

EasyX1.PlayStreamingSound StreamSound, 0

cmdStopStream.Enabled = True

End Sub
```

Since streaming sounds are usually longer than static sounds an option to stop it from playing is supplied in the cmdStopStream procedure.

```
Private Sub cmdStopStream_Click()

cmdStopStream.Enabled = False

EasyX1.StopStreamingSound StreamSound

End Sub
```

That is actually it, very easy and quick to set up.

Tips and Tricks

This section is divided into three sections, a DirectDraw section, a DirectSound Section and a DirectInput section. Each section briefly describes some features of the way that EasyX implements each technology and then gives some pointers on how to avoid errors.

DirectDraw and EasyX

EasyX implements DirectDraw in a very straightforward and easy manner. When DirectDraw is initialized with the *InitDirectDraw* function, a primary display surface and a back buffer is created. These two surfaces are flipped when calling the *FlipSurface* function, meaning that everything drawn onto the back buffer, are now shown on the screen. Always remember this, since you can not draw on the primary surface through EasyX, only onto the back buffer.
To use graphics with EasyX two steps must be performed. The first step is to load the graphics using the *LoadBitmapFile* function, this function, if successful, will return an index to the loaded bitmap surface. Then a sprite must be defined using the *MakeSprite* function. The sprite is defined as an area on the bitmap surface supplied to the function. This will enable you to define several sprites on a single surface, and thereby using bitmaps consisting of several sprites (this is actually the recommended way to do it, check out the undocumented sample in this package). When the sprite should be drawn, use the *DrawSprite* function. This function should be supplied with the details on where to draw the sprite and what dimensions the sprite should have. The *DrawSprite* function will stretch the sprite to fit the dimensions supplied to the function. To make the sprite appear on the screen, use the *FlipSurface* function.

To summon up on this information:

To initialize DirectDraw:

1) Set the Window property of the EasyX control.
2) Initialize DirectDraw with the *InitDirectDraw* function.

To use graphics:

1) Load a bitmap with *LoadBitmapFile* function.
2) Make a sprite on the surface created in step 1, with the *MakeSprite* function.
3) Draw the defined sprite with the *DrawSprite* function.
4) Flip the back buffer with the *FlipSurface* function.

A big warning is place at this point. Since speed is essential in Drawing sprites, filling and slipping surfaces, error checking in the EasyX control has been kept to a minimum in some functions. So if you try to fill a surface with *FillSurface* function and DirectDraw has not been initialized correctly, you will get an runtime **'Automation Error'**. So take great care when initializing DirectDraw, and check the return values. Likewise when loading bitmaps and defining sprites.

Tips:

- Always check return values when initializing and setting up the application.
- Do not use the form where the control is placed for anything else than keeping the control
- Use the *FillSurface* function to fill the back buffer, before any drawings are made.
- Always moderate the bitmaps loaded into surfaces, not too small and not too big. This saves resources used on both the video cards and system memory.
- Remember that you can free all surfaces with the *ReleaseSurfaces* function.

- Do not use the Do – Loop scenario in the undocumented samples as the Main function, since its execution speed varies.
- Always use the constants defined in the EX_CONSTANTS module

Windowed DirectDraw

As of version 1.1 EasyX will enable the developer to use DirectDraw® in windowed applications. This provides the developer with bundle of opportunities, primarily the fact that the application will operate in the normal Windows environment. One very important thing to note is that both modes can not be used at the same time. Either the 'normal' exclusive mode is used or the windowed mode is used. Using Windowed DirectDraw might seem to be the best choice, because all normal functions that applies to the Window environment can be used. But, and there is a big but, some very profound restrictions apply. The most important thing is the lack of page flipping, which is a primary feature of full screen DirectDraw. And other very important 'feature' is the fact that the application resides on the normal desktop, and is therefore subject to normal windows behavior. This is an especially important fact to take into account, when running in a palettized mode.

To create a windowed DirectDraw session with EasyX, start by setting the *Window* property of the control. After this, very important step, a call to the *IWInitDirectDraw* will initialize and create the primary surface of the window. This function takes one parameter, a string to a bitmap file, from which a palette will be created and attached to the primary surface. This parameter is only useful in palettized mode. If the mode is palettized and an empty string is passed, a standard palette will be created and attached to the surface. When operating in palettized mode it is always a good idea to pass a string to a bitmap when initializing DirectDraw, this way the graphics that the DirectDraw application uses will always look like they should. But setting a specific palette when initializing is not always enough, since the palette may be changed by the system, if the application loses focus. So when the application gets activated again, the palette will be the one the prior application used, and the graphics may look a little funny. To prevent the funny looking graphics, use the function *IWResetSurfaces,* which will reset the palette and reload all the surfaces, so that they once more look normal. All this is of course only a problem in palettized mode.

The other major concern with windowed DirectDraw is the fact that flipping surfaces are unavailable, and the fact that the developer is allowed to blit directly to the primary surface. The result will usually be flickering and tearing. To counter this a fake back buffer can be created, using the *IWCreateFakeBackBuffer* function. The basic principle is that instead of using the normal *IWDrawSprite* function to the sprite, the *IWDrawToBackBuffer* is used. Thereby drawing all sprites into the fake back buffer. When it is time to update the primary surface, the *IWDrawFakeToPrimary* function is used to draw everything in the back buffer to the primary surface. This technique is also known as double buffering. By using this method, the whole surface is updated all at once, and thus avoids flickering and tearing. This of course requires two blits, instead of just one blit to the primary surface, which means it is slower, but nicer. Loading graphics to a windowed DirectDraw session is done in exactly the same way as with full screen modes.

To initialize DirectDraw in a Windowed mode:

1) Set the Window property of the EasyX control.
2) Initialize DirectDraw with the *IWInitDirectDraw* function.

To use the fake back buffer:

1) Create the fake back buffer with *IWCreateFakeBackBuffer* function
2) Draw all the sprites into the fake back buffer with the *IWDrawToBackBuffer* function
3) Draw the back buffer onto the primary surface using the *IWDrawFakeToPrimary* function

Tips with Windowed DirectDraw

- Only use the *IW* specific functions for drawing in a windowed mode
- All functions regarding DirectInput and DirectSound still works normally in windowed mode
- Use the *IWResetSurfaces* function to reload the surfaces and palette when the application has lost focus and gets it again.
- Avoid tearing by using double buffering with the *IWCreateFakeBackBuffer* and related functions.

DirectSound and EasyX

EasyX implements DirectSound in two ways, a Static way and a Streaming way. The static way is used with short, often played sounds, which are not taking up many resources. The streaming way are used with longer sounds, which are being streamed internally by EasyX into a sound buffer, and from the sound buffer mixed into the primary buffer, and then played. Sounds, which are longer than 4 seconds of play, should always be created as a streaming sound, while sounds shorter than 2 seconds should always be static sounds.
Before any sounds are created, the *Window* property should be set and then DirectSound should be initialized with the *InitializeSound* function. After this initialization, sounds are created with either the *CreateStaticSound* or the *CreateStreamingSound* functions. The sound file used to create sounds must a regular PCM WAV format. Whether it is stereo or mono does not matter, but it must be a PCM - format.
After creating the sounds, they can be played with either *PlayStaticSound* or *PlayStreamingSound.* These functions take two arguments, the sound index and a flag, which determines whether the sound should be looped.
One thing to remember is that all the sounds playing are mixed together into the primary buffer and played from there.
EasyX implements some special effects for use with DirectSound, these are the volume and pan setting functions. Especially the pan functions are useful in game programming as a special effect to use when sounds can come from the left or right side (see the Pong game).

To summon up on DirectSound:

To initialize DirectSound with EasyX:

1) Set the Window property of the control
2) Initialize DirectSound with *InitializeSound* function

To play a sound:

1) Created (or rather load) the sound with the *CreateStaticSound* or *CreateStreamingSound* functions, depending on the kind of sound which is being loaded.
2) Play the sound with *PlayStaticSound* or *PlayStreamingSound* functions.

Tips

- Remember that both of the *Play…Sound* functions will return immediately after starting the sound player functions.
- Setting the pan value of specific sounds can create some special directional effects.
- Always duplicate static sounds, instead of reloading the same sound file into a new buffer.

DirectInput and EasyX

DirectInput is a technology that enables the user to directly communicate with the hardware, instead of going through the normal windows message system. This version of EasyX supports only the keyboard and mouse as DirectInput objects, future versions will probably support all ranges of DirectInput objects. The first thing to do when using DirectInput through EasyX is (as usual) to set the Window property of the control. After this a call to the *InitDirectInput* function is made. DirectInput is now set up, and the next thing to is to create the DirectInput objects, mouse or keyboard. This is done through the *CreateMouse* and *CreateKeyboard* functions. The device objects are then created (remember to check the return values), but can not be used before the objects are acquired through the *AcquireKeyboard* or *AcquireMouse* functions.

After this initialization the device object(s) are set up and ready to use. If the keyboard has been chosen as a device, the different state of the keys can be checked with the *GetKeyState* function. The function takes one argument, the keyboard key that should be checked. The return values from the function are EX_KEYDOWN, EX_KEYNOTDOWN or EX_DEVICENOTACQUIRED. In the last case a simple call to *AcquireKeyboard* should fix it. The other two values are flags indicating whether the key that was passed to the function is pressed or not pressed. The values of the different keys are defined in the EX_CONSTANTS module. This kind of 'polling' scenario might seem a bit awkward (at least for the VB programmer), but it is very much faster than the usual event system, plus you can check for multiple keys (with successive calls to the *GetKeyState* function), and react specifically when multiple keys are pressed.

With the mouse as the device object you have two choices, either set the control to event notification or use the *GetMouseState* function. The recommended choice is the *GetMouseState* function, since it is faster and more efficient. The *GetMouseState* function takes three arguments, which are filled with the appropriate values from the function. The X and Y coordinate are relative values, meaning that they represent the movement of the mouse since the last call to the function. The Button argument is filled with values, as represented in the constants EX_LEFTBUTTON, EX_RIGHTBUTTON and EX_BOTHBUTTONS, which are declared in the EX_CONSTANTS module. The event notification is set with the *SetMouseEvents* function. When the event is set, the event *EasyX_MouseEvent* will be fired when the mouse state changes. As with the other mouse functions, the values X and Y are relative to last event, and the Button parameter is represented in the same way as with the *GetMouseState* function.

To summon up on DirectInput through EasyX:

Create a DI object:

1) Set the Window property of the control
2) Initialize DirectInput with *InitDirectInput* function
3) Create a device object with either *CreateKeyboard* or *CreateMouse* functions
4) Acquire the created device with either *AcquireKeyboard* or *AcquireMouse* functions

To use the keyboard object

1) Call the *GetKeyState* function with the key in question as the parameter
2) Check the return value from the *GetKeyState* function.

To use the mouse object

1) Either set the Event notification with the *SetMouseEvent*s function or check the state with the *GetMouseState* function.
2) Remember that all coordinate values are relative values

Tips

- Remember to acquire the device before use
- Always check return values for the EX_DEVICENOTACQUIRED value, reacquire the device if the value was returned
- Make sure that everything is initialized properly
- Remember that the usual Windows mouse cursor disappears when the mouse is acquired. So in order to use a mouse cursor the program has to programmatically create one. See the EX_3 sample in the undocumented samples.