Chapter 21 – Graphics and Multimedia

# Objectives

* Understanding the VB Graphics Commands
* Using the PictureBox Control
* Using the Windows Media Player Control
* Using the System.Media Namespace
* Making the Computer Talk – Speech Synthesis

Microsoft has provided a plethora of drawing commands available inside of the Visual Basic language. This chapter will discuss the purpose of graphics device contexts and coordinate systems. In addition, the chapter will look at how to use pens and brushes to draw both graphic primitives and fonts.

The other half of this chapter will examine how to use multimedia for both audio and video. Sample programs include creating a Windows media player project to play back video files and learning how to make your computer talk. Sound interesting? Let’s get started!

# Understanding the VB Graphics Commands

VB provides a rich set of commands if you want to draw computer graphics yourself. Since Windows is a GUI, this shouldn’t be surprising. The most fundamental graphics operation is to illuminate one pixel in some color.

You can draw on both forms and inside of PictureBoxes. Fortunately, many graphics primitives are built in, including Points, Lines, Circles and Rectangles. You can therefore construct simple shapes with little effort. All 216 web safe palette colors are available and have constant-type names to allow you to use them. In addition, there are many different operations that you can perform when drawing with the built-in graphics commands such as setting line type, line thickness and fill patterns.

It’s important to note that the graphic commands are part of what is known as GDI+ (or Graphical Device Interface+). GDI+ serves as a standard way to access the drawing features that are embedded as part of the Windows operating system. GDI+ is not the only graphics tool that you have access to in VB. It is also possible to directly program more graphically rich environments such as OpenGL or DirectX. The problem with using OpenGL or DirectX is the fact that it is complex to program in these environments compared to GDI+. However, if you all that you need to do is basic 2-D drawings and you don’t care about rendering or high-levels of animation, the built-in VB tools should suffice for most of your needs.

Where are things going? Well since Windows 7, Microsoft has been using a GDI+ replacement which is called Direct2D (it’s part of DirectX). There is also a Direct3D component available as well. Unfortunately, it is a cumbersome process to convert a GDI+ application over to a Direct2D version. Besides the learning curve of mastering the new graphics library, there are very few decent examples of Direct2D programming from the Visual Basic perspective. Since many of you have never encountered any graphics programming concepts before, I am sticking with the simple to learn and use GDI+. I am hoping at some point, Microsoft leaves the existing .NET framework calls alone and translates GDI+ calls automatically to Direct2D behind the scenes.

## Understanding the Basics of Graphics in VB.NET

The System.Drawing and System.Drawing.Drawing2D namespaces have most of the methods that you need for drawing strings, lines, rectangles and so forth. The drawing methods typically require a Pen or Brush object which is then used to create a specific shape. The Pen object is typically used for outlines, while the Brush object is used for solid objects.

To do any sort of drawing, a graphics context is required. This is an overly complicated sounding name for the representation of a virtual drawing surface. If you decide to draw on the screen, you are actually drawing on a graphics context that is displayed on the screen. If you want to print some graphics out on a printer, you place the graphics on the printer's graphics context and then the graphics context is sent to the printer and printed on paper.

Every Windows forms based application inherits a Paint method, which is where most of the graphics drawing operations will occur. The arguments to the Paint method include a PaintEventArgs object from which a graphics context object can be obtained for a particular form or control.

When you want to draw some graphical items on a Form, you will override the Paint method to retrieve a Graphics context object or you can create a new Graphics context object and then associate it with a particular context. The overridden Paint method will look like this:

Private Sub Form1\_Paint(sender As Object, e As PaintEventArgs)

Handles Me.Paint

To gain access to the incoming Graphics context object, the next line would be:

Dim myGfxObj As Graphics = e.Graphics

The variable myGfxObj, which is a Graphics context object, is now available to have items drawn upon it. Calling the Paint method raises the Paint event for a form/control. As an alternative to the Paint method, you could also just add an event handler for the overridable OnPaint event, which forms and most controls have.

Here is an example of the overridable OnPaint method on a form:

'We can override the OnPaint in a form. When we build our own

'controls, we will also override the OnPaint in them so that we

'can render the control just the way that we want.

Protected Overrides Sub OnPaint(ByVal e As PaintEventArgs)

'Call the parent’s OnPaint method

MyBase.OnPaint(e)

'If there is an image and it has a location,

'paint it when the Form is repainted.

If Not (Me.picture Is Nothing) And \_

Not (Me.pictureLocation.Equals(Point.Empty)) Then

e.Graphics.DrawImage(Me.picture, Me.pictureLocation)

End If

End Sub

Finally, most controls, including Labels, Command buttons and PictureBoxes, have their own graphics contexts. To draw graphics on a control, you will first need to get its Graphics context object by calling the CreateGraphics method. An example of this process is shown below:

'An example of getting the Graphics context of a Label

Dim myGfxObj As Graphics = Label1.CreateGraphics()

Once you have access to the control's Graphics object, you can then use the methods in the Graphics class to draw items on the control itself. As you can see, you have several choices through which you can access Graphics contexts. The point is that once you have the context, you can start displaying the graphics you choose on it.

## Coordinate Systems

Now that you have the idea of a graphics context, which we said is essentially a virtual piece of paper that you can draw graphics on, you need to understand how the graphics coordinate system is implemented in VB. Both PictureBox controls and Windows forms have their origin point (0,0) located in the upper left hand corner of the control/form. This is different from many other graphics systems.

## Drawing Primitives

Alright, we've spent enough time on the conceptual material behind drawing graphics. It's time to start looking at the drawing primitives that are built into VB. We'll take a drawing method or two at a time and then write a basic application that illustrates the use of that method. About the only thing that you need to remember at this point is that all of these methods belong to the Graphics class.

***DrawString*** – This command is used to print text on a form or control. You can specify the string's contents, the font to draw it in, the brush to use and the location of where you want the string drawn. The syntax is: DrawString (*StringToPrint*, *Font*, *Brush*, *XOriginCoordinate*, *YOriginCoordinate*). Here's a simple program that introduces fonts, brushes and the DrawString method:

'Chapter 21 - Program 1

Public Class Form1

'In this example, we will be chaining into the Form1 Paint

'event handler. We'll grab the form's graphics context and

'use DrawString to print some text to the form.

Private Sub Form1\_Paint(sender As Object, e As PaintEventArgs)

Handles Me.Paint

'Get the graphics context of the form

Dim myGfxObj As Graphics = e.Graphics

'Create a solid brush in Black

Dim brush As SolidBrush = New SolidBrush(Color.Black)

'Create a font of type Arial, 16 point and no special effects

Dim arial As Font = New Font("Arial", 16, FontStyle.Regular)

'For our screen coordinates

Dim intX As Integer

Dim intY As Integer

'Print letter "A"s all over the form

For intX = 0 To 800 Step 40

For intY = 0 To 600 Step 60

'Draw an A using the arial font in a solid black brush

'at coordinates (intX,intY)

myGfxObj.DrawString("A", arial, brush, intX, intY)

Next

Next

End Sub

End Class

Here's a screenshot of the DrawString application in action:



One of the interesting features of a graphics context is the fact that it really is a virtual canvas. This fact becomes evident when you run the previous application and resize your form. You'll notice that the form is essentially a viewport that looks at just a part of the context. As you shrink or enlarge the form, you will see less or more of the context. In the above example, if you make the form quite large, you will see sections of the context where we did not draw the “A” string. We dimensionally sent strings to the context from location (0,0) to (800,600).

To get a good idea of the Font and SolidBrush objects and the DrawString methods, you should play around with the example program provided. Try using different coordinate sets, font styles and sizes and different colors for the SolidBrush object.

DrawString is one of the simplest graphics methods that we have available.

The next series of methods that we want to examine, DrawLine, DrawRectangle and FillRectangle, are similar in style. All require a pen or brush as the first argument and then two sets of coordinate-type data:

***DrawLine*** – This method can be used to draw a line between two sets of coordinates using a pen. The syntax is: DrawLine (*pen*, *x1*, *y1*, *x2*, *y2*).

***DrawRectangle*** – This method can be used to draw a rectangle starting at coordinate (x,y) and then extending a given width and height from that starting coordinate using a pen. This method's syntax is: DrawRectangle (*pen*, *x1*, *y1*, *RectangleWidth*, *RectangleHeight*).

***FillRectangle*** – This method is similar to DrawRectangle except that the rectangle is filled in using a brush instead of just outlined with a pen. The syntax is: FillRectangle (*brush*, *x1*, *y1*, *RectangleWidth*, *RectangleHeight*).

Let's examine a program this will draw these graphic objects:

'Chapter 21 - Program 2

Public Class Form1

'Here's a different technique for getting the Form's

'graphic context. We will Override the OnPaint method

'so that whenever the form is told to paint, this

'method will be triggered.

Protected Overrides Sub OnPaint(e As PaintEventArgs)

'Get the context to the form

Dim myGfxObj As Graphics = e.Graphics

Dim myPen As Pen

Dim myBrush As SolidBrush

'We should call the parent's OnPaint method

MyBase.OnPaint(e)

'Now, let's perform our custom work.

'Draw a line with a black pen

myPen = New Pen(Color.Black)

myGfxObj.DrawLine(myPen, 0, 0, 700, 500)

'Draw a line with a green pen

myPen = New Pen(Color.Green)

myGfxObj.DrawLine(myPen, 0, 0, 800, 50)

'Draw a rectangle with a blue pen

myPen = New Pen(Color.Blue)

myGfxObj.DrawRectangle(myPen, 100, 100, 200, 200)

'Draw a filled rectangle with a red brush

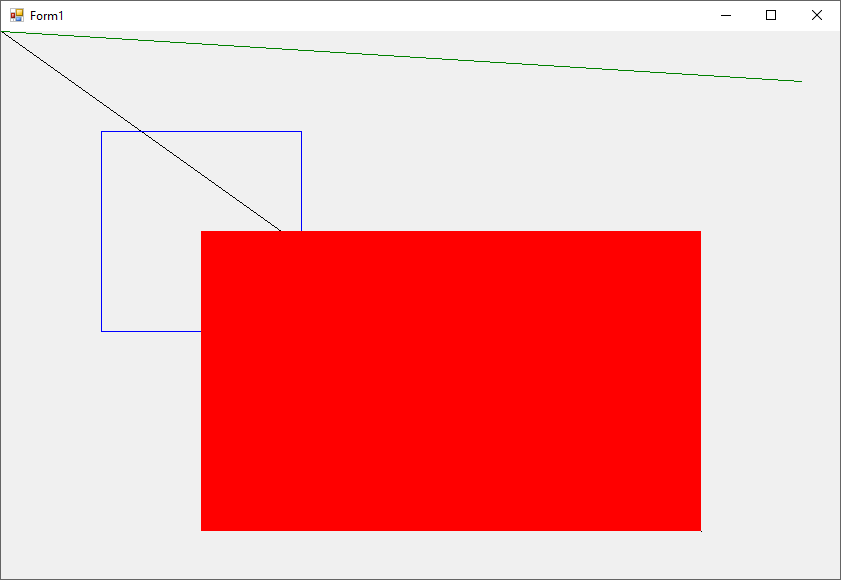
myBrush = New SolidBrush(Color.Red)

myGfxObj.FillRectangle(myBrush, 200, 200, 500, 300)

End Sub

End Class

When the program is run, you will see the following form created:



Following along with the basic drawing methods, we need to learn how to draw circles and ellipses. VB provides two methods for our use: DrawEllipse and FillEllipse. There are no methods for drawing circles, and there is no need for any. All that we do is set the width and the height of the ellipse to the same value: an instant circle.

***DrawEllipse*** – This method will draw an ellipse outline using a pen. The syntax for the method is: DrawEllipse (*pen*, *XOriginCoordinate*, *YOriginCoordinate*, *EllipseWidth*, *EllipseHeight*).

***FillEllipse*** – This method is similar to DrawEllipse, but fills the ellipse using a brush. The syntax is: FillEllipse (*brush*, *XOriginCoordinate*, *YOriginCoordinate*, *Ellipsewidth*, *EllipseHeight*).

Our third example program examines drawing circles and ellipses on a form. Here's the source code for that program:

'Chapter 21 - Program 3

Public Class Form1

Protected Overrides Sub OnPaint(ByVal e As PaintEventArgs)

Dim g As Graphics = e.Graphics

Dim brush As New SolidBrush(Color.Blue)

Dim pen As New Pen(Color.Red)

'We should call the parent's OnPaint method,

'but we don't have to -- notice this is commented out

'MyBase.OnPaint(e)

'Draw an outlined ellipse

g.DrawEllipse(pen, 50, 50, 30, 120)

'Draw an outlined circle

g.DrawEllipse(pen, 100, 100, 80, 80)

'Draw a filled in ellipse

g.FillEllipse(brush, 200, 70, 150, 60)

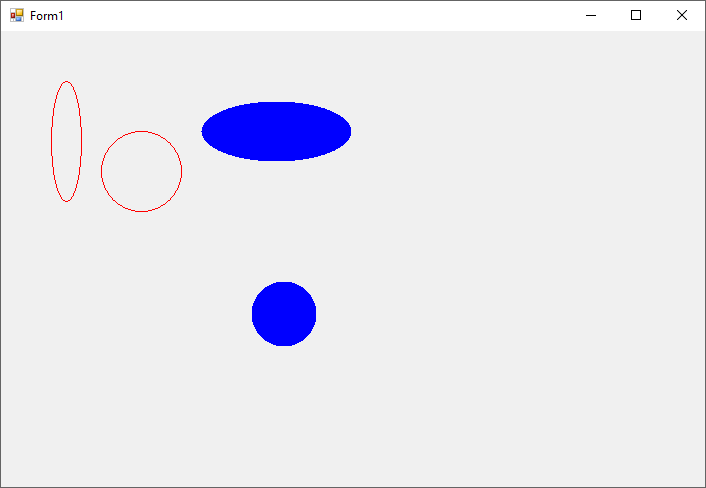
'Draw a filled in circle

g.FillEllipse(brush, 250, 250, 65, 65)

End Sub

End Class

Here's a screenshot of our program in action:



If we continue on with another set of primitives that are related to ellipses, we find the DrawArc, DrawPie and FillPie methods. These methods are used to draw portions of ellipses or circles.

***DrawArc*** – This method draws an arc (which is the outside edge of a circle) using a pen. The syntax for the method is: DrawArc (*pen*, *XOriginCoordinate*, *YOriginCoordinate*, *CircleWidth*, *CircleHeight*, *StartAtAngle*, *SweepToAngle*).

***DrawPie*** – This method draws a pie shaped section of an arc using a pen outline. The syntax is: DrawPie (*pen*, *XOriginCoordinate*, *YOriginCoordinate*, *CircleWidth*, *CircleHeight*, *StartAtAngle*, *SweepToAngle*).

***FillPie*** – This method is used to fill in a pie section of an arc using a brush. Syntactically the method is: FillPie (*brush*, *XOriginCoordinate*, *YOriginCoordinate*, *width*, *height*, *StartAtAngle*, *SweepToAngle*).

Let's look at a program that creates these different entities:

'Chapter 21 - Program 4

Public Class Form1

Protected Overrides Sub OnPaint(ByVal e As PaintEventArgs)

Dim g As Graphics = e.Graphics

Dim brush As New SolidBrush(Color.Blue)

Dim pen As New Pen(Color.Red)

'MyBase.OnPaint(e)

'Draw a circle in red

g.DrawArc(pen, 40, 40, 90, 90, 0, 360)

'Draw an arc in green

pen.Color = Color.Green

g.DrawPie(pen, 100, 100, 80, 80, 0, 110)

'Draw an arc in black

pen.Color = Color.Black

g.DrawArc(pen, 200, 200, 75, 75, 0, 270)

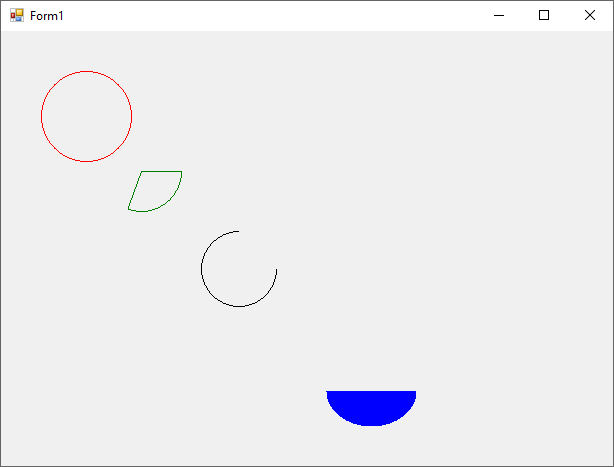
'Draw an elliptic pie piece in Blue

g.FillPie(brush, 325, 325, 90, 70, 0, 180)

End Sub

End Class

Here's the output from the program:



The last set of primitive methods that we are going to examine in this section all deal with drawing multiple, connected lines. These methods include DrawLines, DrawPolygon and FillPolygon.

***DrawLines*** – This method is used to draw a series of connected lines using a pen. Each point in the line set is provided through an array of Points.

***DrawPolygon*** – This method is used to draw a polygon using a pen. Again, each point is provided by an array of Points. The primary difference between this method and the last one is that a Polygon is always a closed shape, whereas DrawLines does not have produce a closed shape.

***FillPolygon*** – This method is used to fill in a polygon using a brush. Each point is provided in an array of points.

Creating the array of Points is about the most difficult aspect of working with these methods. Program 5 shows an example of how to do this:

'Chapter 21 - Program 5

Public Class Form1

Protected Overrides Sub OnPaint(ByVal e As PaintEventArgs)

Dim g As Graphics = e.Graphics

Dim pen As New Pen(Color.Blue)

Dim brush As New SolidBrush(Color.SaddleBrown)

Dim point1 As ArrayList = New ArrayList()

Dim point2 As ArrayList = New ArrayList()

Dim point3 As ArrayList = New ArrayList()

Dim pointArray() As Point

'MyBase.OnPaint(e)

'Create endpoints of lines

point1.Add(New Point(10, 10))

point1.Add(New Point(90, 120))

point1.Add(New Point(40, 180))

point1.Add(New Point(225, 75))

'Convert to a point array

pointArray = point1.ToArray(point1(0).GetType())

'Draw lines from array points

'Also notice we are just creating the pen dynamically this time

g.DrawLines(New Pen(Color.Red), pointArray)

'Create endpoints of lines

point2.Add(New Point(190, 190))

point2.Add(New Point(305, 280))

point2.Add(New Point(560, 110))

pointArray = point2.ToArray(point2(0).GetType())

'Draw polygon from array points

g.DrawPolygon(pen, pointArray)

point3.Add(New Point(10, 500))

point3.Add(New Point(200, 300))

point3.Add(New Point(450, 470))

pointArray = point3.ToArray(point3(0).GetType())

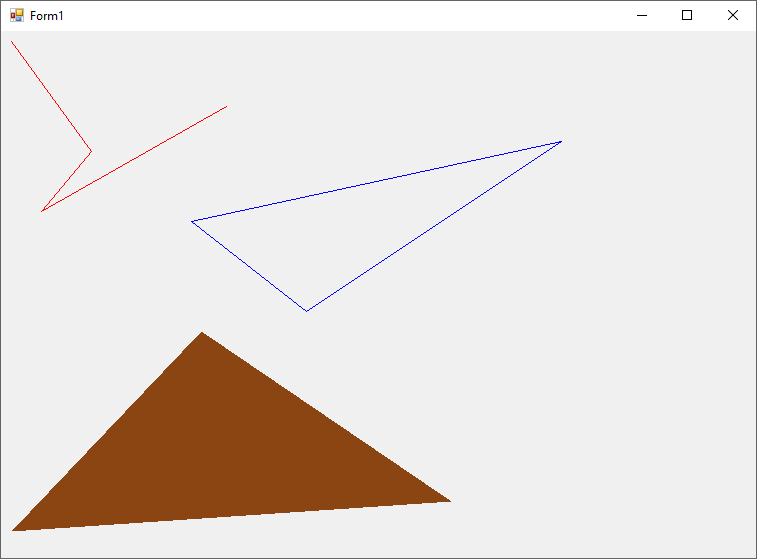
'Fill polygon from array points

g.FillPolygon(brush, pointArray)

End Sub

End Class

Here is the output from our application:

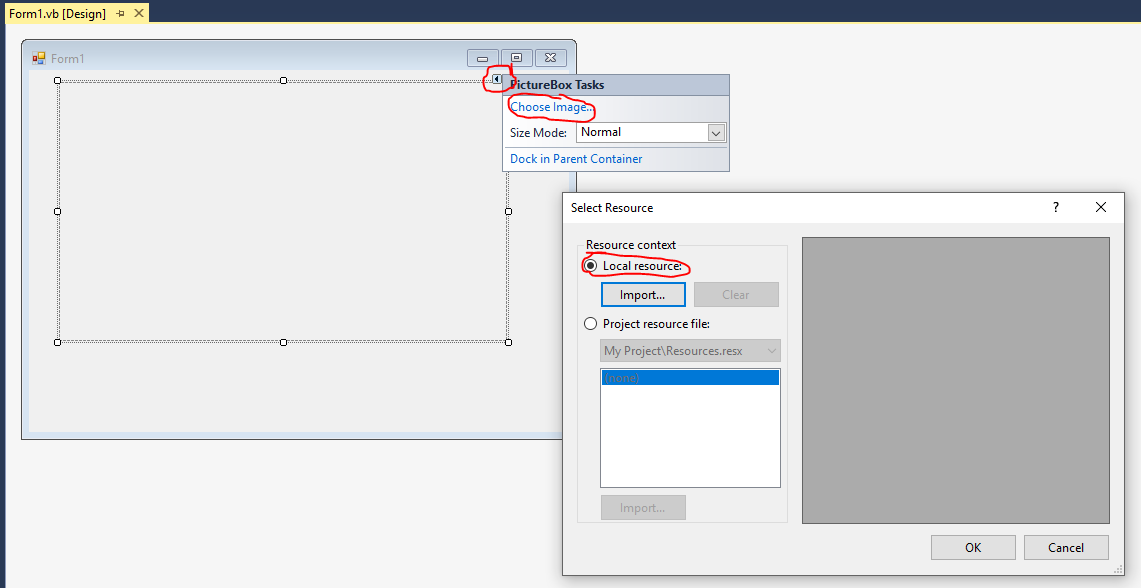


# Using the PictureBox Control

The PictureBox control is an extremely useful control for displaying images. This control will allow you to load and display Bitmap (.bmp), Icon (.ico), GIF (.gif), Metafiles (.wmf), JPEG and PNG images.

You may want to research the DrawImage method on the PictureBox – it can be used to apply some basic special effects to the images that you bring in (although many of the effects only work on bitmap images).

You can select an image to place in the PictureBox directly from the Properties window of the control or through the Tasks arrow. The next screenshot shows the Select Resource dialog box that is automatically displayed allowing you to browse around on your computer to locate images. As soon as you pick an image, it is automatically displayed in the PictureBox.



The following list provides some of the more handy properties from the PictureBox control:

|  |  |
| --- | --- |
| *Property* | *Purpose* |
| Height | Sets the height of the PictureBox control (can be used for resizing) |
| Image | Gets or sets the filename of the image to be displayed in the control |
| ImageLocation | Will allow you specify where the image is located and then bring the image in via a Load or LoadAsync method call |
| Size | Contains the height and width properties of the control |
| SizeMode | Indicates whether the image should be AutoSized, Centered, Stretched or placed normally |
| Width | Sets the width of the PictureBox control (can be used for resizing) |

Let’s look at a program that will manipulate the graphics context of a PictureBox as well as illustrate using some of the PictureBox properties mentioned above:

'Chapter 21 - Program 6

Public Class Form1

Private Sub Form1\_Load(sender As Object, e As EventArgs) Handles Me.Load

'Let's load the picture up in the form's load and set its size

PictureBox1.SizeMode = PictureBoxSizeMode.StretchImage

'In the Debug directory...

PictureBox1.ImageLocation = "Changing of the Guard.jpg"

PictureBox1.Load()

End Sub

Private Sub PictureBox1\_Paint(sender As Object, e As PaintEventArgs)

Handles PictureBox1.Paint

'Now inside of the picturebox's paint method, draw the text

Dim myPictureBoxGFX As Graphics

Dim myBrush As SolidBrush = New SolidBrush(Color.AliceBlue)

Dim myFont As Font = New Font("Calibri", 20, FontStyle.Bold)

myPictureBoxGFX = e.Graphics

myPictureBoxGFX.DrawString("Changing of the Guard", myFont, myBrush, 20,

10)

End Sub

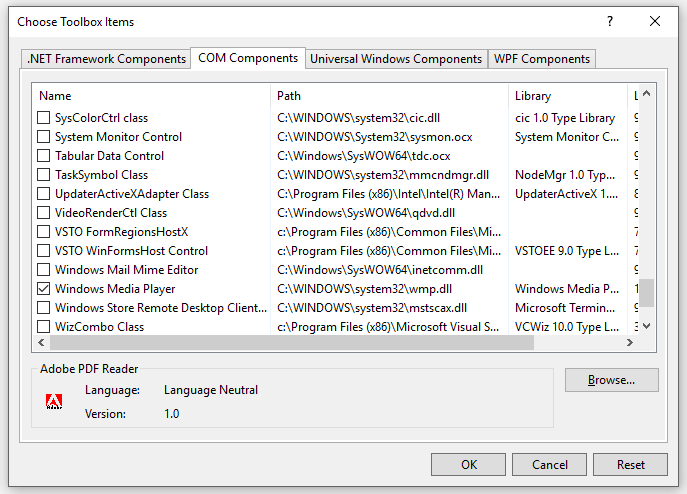
End Class

Here’s the image with the text added to it:



# The Windows Media Player Control

This control is important if you want to play a variety of audio video formats. The control can currently handle MPEG video and audio, AVI video, WAV audio and MIDI. The control is also programmatically pretty easy to use and it is a .NET certified COM control. Modify your toolbox (Tools🡪Choose Toolbox Items) so that you have access to it. Click on the COM Components tab, scroll down and check Windows Media Player and then press the OK button:



Let’s begin by looking at some of the properties that are available in the Windows Media Player control:

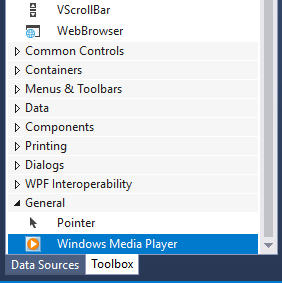
|  |  |
| --- | --- |
| *Property* | *Purpose* |
| Ctlenabled | Gets or sets a value indicating whether the Windows Media Player control panel is enabled |
| CurrentMedia | Gets or sets the IWMPMedia interface that corresponds to the current media item |
| CurrentPlaylist | Gets or sets the IWMPMedia interface |
| EnableContextMenu | Gets or sets whether to enable the context menu |
| FullScreen | Gets or sets if the player should be full screen |
| IsOnline | Gets a value indicating whether the user is connected to a network |
| MediaCollection | Gets an IWMPMediaCollection interface |
| Network | Gets an IWMPNetwork interface |
| OpenState | Gets a value indicating the state of the content source |
| PlaylistCollection | Gets an IWMPPlaylistCollection interface |
| PlayState | Gets a value indicating the state of the player operation |
| Settings | Gets an IWMPSettings interface |
| Status | Gets a value indicating the current status of the player |
| StretchToFit | Gets or sets a value indicating whether a video will stretch to fit the size of the player |
| UIMode | Gets or sets a value indicating which controls are shown in the user interface of the player |
| URL | Gets or sets the name of the media item to play |
| VersionInfo | Returns the version of the player |
| WindowlessVideo | Gets or sets a value indicating whether the player control renders video in a windowless state |

Here are the more important Windows Media Player control methods:

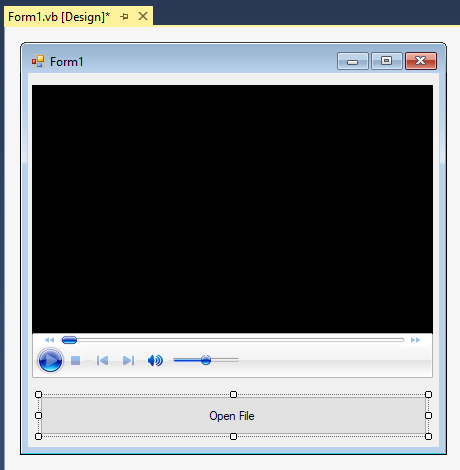
|  |  |
| --- | --- |
| *Method* | *Purpose* |
| Close | Releases the Media Player’s resources |
| LaunchURL | Sends a URL to the user’s default browser to be rendered |
| NewMedia | Returns an IWMPMedia interface for a new media item |
| NewPlaylist | Returns an IWMPPlaylist interface for a new playlist |
| OpenPlayer | Opens the player using the specified URL |

Let's create an application that allows a user to browse for media files using the Open File Dialog and then playback what the user selects. You will create a form with a copy of the Media Player control (default name of axWindowsMediaPlayer1), a copy of the OpenFileDialog box (default name) and a Command button (cmdOpenFile) which is used to display the dialog box.

Remember that you added the Media Player control to your toolbox, so you should find it toward the bottom under the General section. Just drag and drop a copy of the control on your form:



Here's a screenshot of the completed form:



Here’s the source code for the application:

'Chapter 21 - Program 7

Public Class Form1

Private Sub cmdOpenFile\_Click(sender As Object, e As EventArgs)

Handles cmdOpenFile.Click

'Display the OpenFileDialog -- point to debug folder for a video

OpenFileDialog1.InitialDirectory = Environment.CurrentDirectory

OpenFileDialog1.ShowDialog()

'Set the media player's filename to whatever the user chose,

'make sure the media player is visible, and then resize the

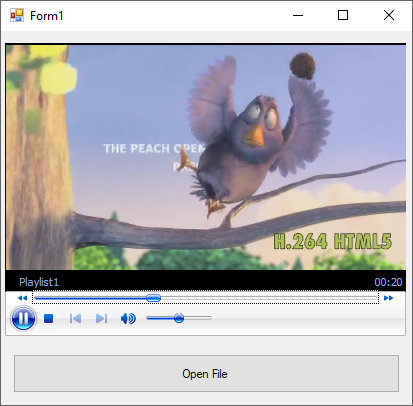
'player and the form to accommodate the file's image size

AxWindowsMediaPlayer1.URL = OpenFileDialog1.FileName

End Sub

End Class

The next screenshot shows our Media Player application in action:



# Using the System.Media Namespace

Sometimes we want to play a WAV file without the overhead and expensive of loading up the Windows Media Player. While it’s a nice control, it’s also very resource heavy. The System.Media namespace exposes several classes that let us get the job done natively with the .NET framework. The SoundPlayer class, for example, will let us load and play audio files with very little effort.

There’s also super quick access to the system sounds that Windows makes. You know like the “ding” when something goes wrong. Well, we can also access those through the SystemSounds class.

Let’s build an application to showcase how these two concepts work. I have included two Command buttons (default named). The top button (Button1) will allow you to pick an audio file to play. The lower button (Button2) will play the Exclamation sound from the SystemSounds class. The source code is super straightforward:

'Chapter 21 - Program 8

Imports System.Media

Public Class Form1

Private Sub Button1\_Click(sender As Object, e As EventArgs)

Handles Button1.Click

Dim myOFD As New OpenFileDialog

Dim myPlayer As New SoundPlayer

myOFD.InitialDirectory = Environment.CurrentDirectory

myOFD.Filter = "WAV Files|\*.WAV|All Files|\*.\*"

'The sound player class will let us pick the file to play

If myOFD.ShowDialog = Windows.Forms.DialogResult.OK Then

myPlayer.SoundLocation = myOFD.FileName

myPlayer.Play()

End If

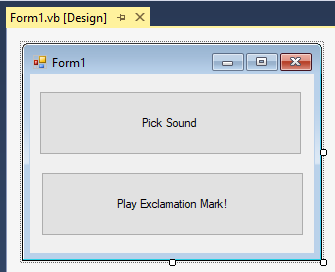
End Sub

Private Sub Button2\_Click(sender As Object, e As EventArgs)

Handles Button2.Click

'The SystemSounds class will let us play the default system sounds

'without any work on our part!

 SystemSounds.Exclamation.Play()

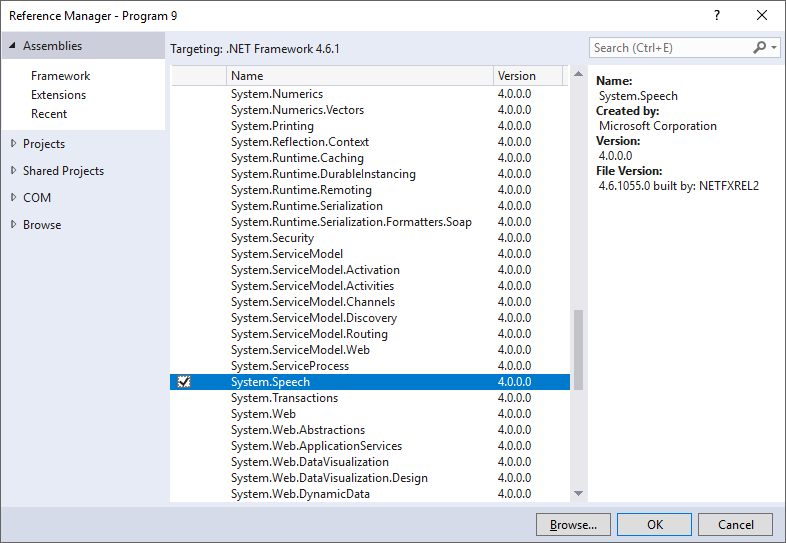
End Sub

End Class

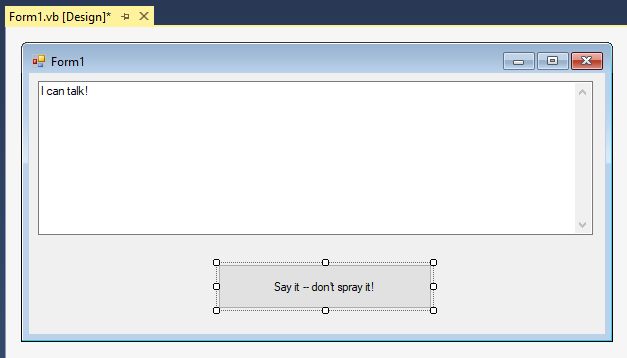
Here’s the form, for what it’s worth:

# Making the Computer Talk – Speech Synthesis

One more item that may be more of a parlor trick, but you may find a need for it, is speech synthesis. .NET provides you with these capabilities, and since we are looking at multimedia, we will cover this too. Begin by including a Reference (Project🡪Add Reference) to the NET Speech Library, which is under the Assemblies tab:



Remember to check the checkmark out in front and then click OK. Next, create a form that has a multiline TextBox (default named) and one Command button (default named):



All the real work (not that there’s much) will happen in the code behind the form:

'Chapter 21 - Program 9

'Let's import the speech synthesizer namespace

Imports System.Speech.Synthesis

Public Class Form1

Private Sub Button1\_Click(sender As Object, e As EventArgs)

Handles Button1.Click

'Create a new speech synthesizer

Dim mySpeech As New SpeechSynthesizer

'This is it -- call the Speak method using

'the default voice

mySpeech.Speak(TextBox1.Text)

'These are the default installed voices

'Name: Microsoft David Desktop

'Name: Microsoft Zira Desktop

'Select another voice and speak the text again

mySpeech.SelectVoice("Microsoft Zira Desktop")

mySpeech.Speak(TextBox1.Text)

End Sub

End Class

And there you have it. You may have to phonetically spell a few words, but for the most part modern speech synthesizers have it down pretty good!