LECTURE NOTES

PROGRAMMING WITH VISUAL BASIC 6.0

CHAPTER 7

“ADDING ARTWORK AND ANIMATION”

PREPARED BY

CHAPTER 7: ADDING ARTWORK AND ANIMATION

This chapter describes how to enhance your program user interface with artwork and special effects. The skills you learn will teach you more about graphics programming and using a coordinate system and the results will make your applications more interesting to use. In this chapter, you will learn how to:

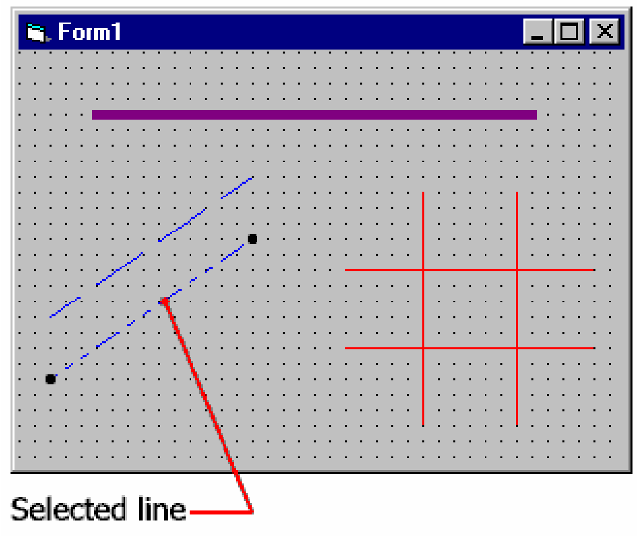
- Create basic artwork with the **Line** and **Shape** controls.
- Add drag-and-drop support to your user interface.
- Create animation effects with the **Timer** control.

7.1: Creating Original Artwork

7.1.1: Line Control

Creating a line on a Visual Basic form is a three-step process. You start by creating the line with the **Line** control. Next, you set a variety of properties that change the appearance of the line you create, just as you do for other objects. Finally, if it's necessary to move or resize the line, Visual Basic surrounds it with selection handles.

The figure below shows several lines created with the **Line** control.



Here's a list of the most important **Line** control properties and the purpose of each:

Property	**Purpose**
BorderWidth	Adjusts the thickness of the line on your form. This is especially useful when you are creating an underline, or a line that separates one object from another.
BorderStyle	Makes the line solid, dotted, or dashed.
BorderColor	Sets the line color to any Visual Basic standard color.
Visible	Hides or displays the line as needed in your program.

7.1.2: Shape Control

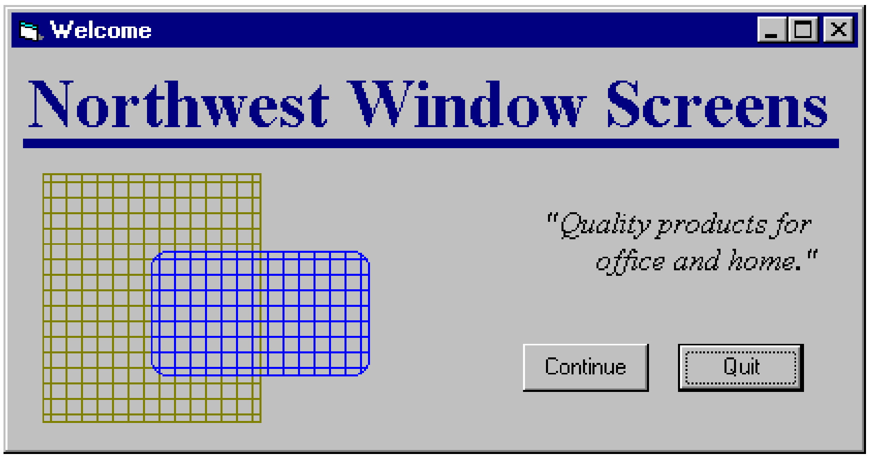
You can use the Shape control to build complex images on your forms by combining lines with rectangles, squares, ovals, and circles. To create shapes with the **Shape** control, you first use the control to draw the image you want. Then, you use the Properties window to adjust the image characteristics.

The **Shape** control has properties similar to those of the **Line** control. Here's a list of the most important **Shape** control properties and the purpose of each:

Property	**Purpose**
Shape	Controls the shape of the image after you create it.
FillColor	Specifies the color of the shape.
FillStyle	Specifies a pattern for the fill color.
BorderColor	Specifies a separate color for the shape's border.
Visible	Hides or displays your artwork in your program as needed.

Embellishing a Splash Screen

The illustration below shows how to use the **Line** and **Shape** controls to embellish a splash screen. The screen advertises a fictitious business named Northwest Window Screens.



In the illustration, note the following property settings:

- The line **BorderColor** property is set to dark blue.
- The line **BorderWidth** property is set to 5.
- Two shapes demonstrate the rectangle and rounded rectangle properties of the **Shape** control.
- In both shapes, the **FillStyle** is set to Cross.

7.1.3: Graphics Methods

In addition to the **Line** and **Shape** controls, Visual Basic also supports several graphics methods, with which you can add artwork to your programs. By using keywords in event procedures, you can create special visual effects on screen or on paper.

Advantages and Disadvantages

Graphics methods give you the advantage of creating visual effects (such as arcs and individually painted pixels) that you can't create by using the **Line** or **Shape** control. Here's a list of the most important graphics methods and the purpose of each:

Method	**Purpose**
Line	Creates a line, rectangle, or solid box.
Circle	Creates a circle, ellipse, or pie slice.
Pset	Sets the color of an individual pixel on the screen.

The disadvantages of using graphics methods include the considerable planning and programming time needed to use them. You need to learn the command syntax, understand the coordinate system used on your form, and refresh the images if they become covered by another window.

For example, the following **Circle** statement draws a circle with a radius of 750 twips at (x, y) coordinates (1500, 1500) on a form:

```
Circle (1500, 1500), 750
```

7.2: Visual Basic Animation

Creating shapes and dragging objects add visual interest to a program. For programmers, though, animation has always been the king of graphical effects. Animation is the simulation of movement produced by rapidly displaying a series of related images on the screen.

In a way, the drag and drop technique is the poor man's animation — it lets you move images from one place to another on a form. But real animation requires you to move objects with program code. Often, animation involves changing the size or shape of the images along the way.

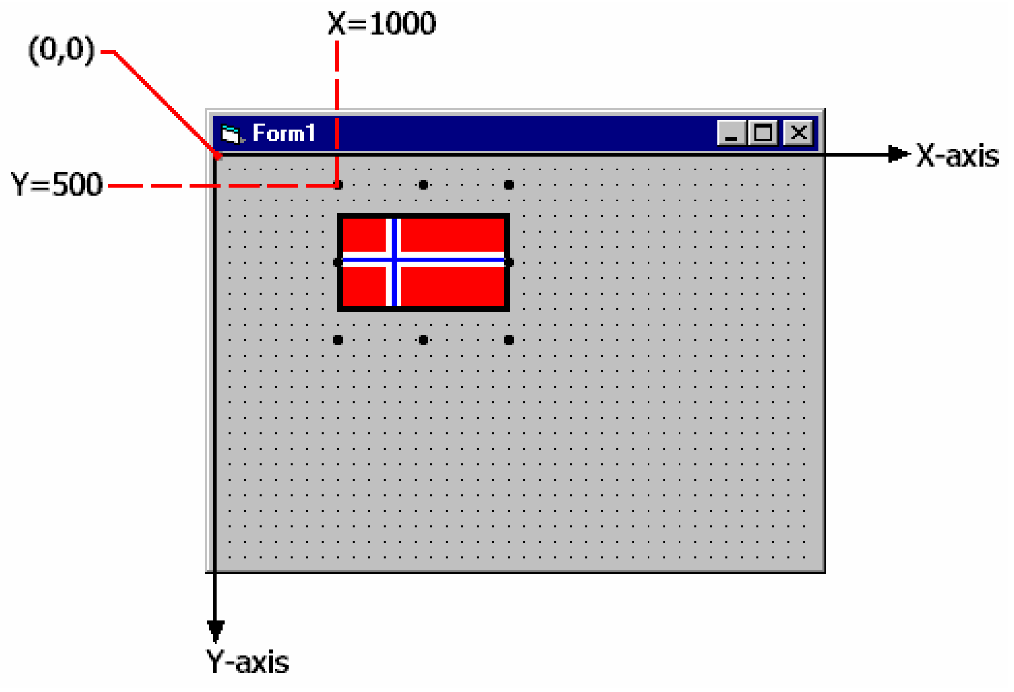
7.2.1: The Form's Coordinate System

Moving images in relation to a predefined coordinate system on the screen is a common trait of animation routines. In Visual Basic, each form has its own coordinate system. The starting point (origin) of this system is the form's upper left corner.

The Default Coordinate System

The default coordinate system consists of rows and columns of twips. (A twip is 1/20 point, or 1/1440 inch.) In the Visual Basic coordinate system, rows of twips are aligned to the x (horizontal) axis, and columns of twips are aligned to the y (vertical) axis. To define locations in the coordinate system, you identify the intersection of a row and column with the notation (*x*, *y*). You can change units in the coordinate system scale, but the (*x*, *y*) coordinates of the upper left corner of a form are always (0, 0).

The figure below shows how the Visual Basic coordinate system describes an object's location on a form.



7.2.2: Moving Objects

Visual Basic includes a special method, with which you move objects in a form's coordinate system. This method, named the **Move** method, uses the following syntax:

```
object.Move left, top
```

where:

object is the name of the object on the form that you want to move.

left is the x screen coordinate of the new location for the object (measured in twips).

top is the y screen coordinate of the new location for the object (measured in twips).

The *left* measurement is also the distance between the left edge of the form and the object; the *top* measurement is the distance between the top edge of the form and the object.

For example, this Visual Basic statement moves the **Picture1** object to the location (1440, 1440) on the screen, or exactly one inch from the top edge of the form and one inch from the left edge of the form:

```
Picture1.Move 1440, 1440
```

Relative Movement

You can also use the **Move** method to specify relative movement. Relative movement is the distance the object should move from its current location. When you specify relative movements, you use:

- The **Left** and **Top** properties of the object. (These values maintain the object's x- and y-axis location.)
- A plus (**+**) or minus (**-**) operator.

For example, this statement moves the **Picture1** object from its current position on the form to a location 50 twips closer to the left edge and 75 twips closer to the top edge:

```
Picture1.Move Picture1.Left - 50, Picture1.Top – 75
```

Moving a Collection

In Visual Basic terminology, the entire set of objects on a form is called the Controls collection. To animate all the objects on your form, you use the Controls collection with a special **For…Each** loop.

This loop processes eac h object according to a pattern that you set. For example, if you put the following program statements into a command button event procedure, Visual Basic moves each object on your form 200 twips to the right each time you click the command button:

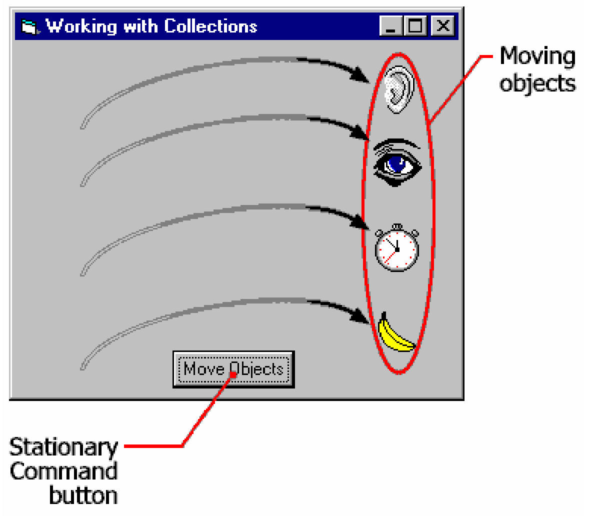
```
Dim Ctrl as Object
For Each Ctrl In Controls
    Ctrl.Left = Ctrl.Left + 200
Next Ctrl
```

Making an Exception

You don't have to move all of the objects in the Controls collection at once. To make just some of them move, simply assign the ones you want to remain stationary with a **Tag** property. With this tag, your **For…Each** loop skips over them. In the following program statements, for example, each object on your form (except the ones containing the “Button” tag) move right:

```
Dim Ctrl as Object
For Each Ctrl in Controls
    If Ctrl.Tag <> "Button" Then
        Ctrl.Left = Ctrl.Left + 200
    End If
Next Ctrl
```

Below is an illustration that shows how you might use a **For...Each** loop with exceptions. Each time the user clicks the command button, the four image objects move to the right—the command button itself, however, remains stationary. (Its **Tag** property has been set to "Button



Collections are useful tools in other programming situations, including controlling forms, switching printers, and managing databases. For more information, search for *collections* in the Visual Basic online Help.

7.3: Other Animation Effects

7.3.1: Expanding and Shrinking Images

In addition to maintaining **Top** and **Left** properties, Visual Basic maintains **Height** and **Width** properties for most objects on a form. You can use the **Height** and **Width** properties in clever ways to expand and shrink objects while a program runs.

Expanding an Object

If you want to expand (zoom in on) an object, increase its **Height** and **Width** properties in an event procedure. For example, this code expands an image object by 150 twips in height and 200 in width:

```
Image1.Height = Image1.Height + 150
Image1.Width = Image1.Width + 200
```

You'll see this technique demonstrated in Review Lab 8.

Shrinking an Object

If you want to shrink (zoom out on) an object, decrease its **Height** and **Width** properties in an event procedure. For example, this code shrinks an image object by 150 twips in height and 200 in width:

```
Image1.Height = Image1.Height - 150
Image1.Width = Image1.Width - 200
```

7.3.2: Creating Animation with the Timer

The trick to creating animation in a program is placing one or more **Move** methods in a timer event procedure. That way, the timer will cause one or more objects to drift across the screen at set intervals.

In Creating a Digital Clock, you learned to use a timer object that updated a simple clock utility every second, so that the object displayed the correct time. When you create animation, you set the timer Interval property at a much faster rate — 1/5 second (200 milliseconds), 1/10 second (100 milliseconds), or less. The exact rate you choose depends on how fast you want the animation to run.

Stopping the Animation

Using the **Top** and **Left** object properties to sense the top and left edges of the form is another important animation technique. By using these properties in an event procedure, you can stop the animation (disable the timer) when an object reaches the edge of the form. You can also make an object appear to bounce off one or more edges of the form. To achieve this feat, you would use any combination of the **Top** and **Left** properties in an **If...Then** or a **Select Case** decision structure.

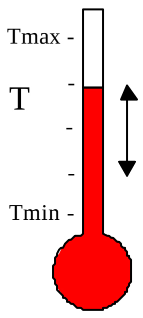
Using the Top Property

For example, the following program statements use the **Top** property to check if an object has reached the top edge of the form during an animation. If the object reaches the edge, the object becomes invisible, and the timer object (which runs the animation) is disabled.

```
If Picture1.Top > 0 Then
    Picture1.Move Picture1.Left - 50, Picture1.Top - 75
Else
    Picture1.Visible = False
    Timer1.Enabled = False
End If
```

The decision structure uses a **Move** method to move the cloud 50 twips closer to the left edge of the form and 75 twips closer to the top edge of the form. When the timer is set to 65 milliseconds per clock tick, the clouds drift gently by.

EXAMPLE: Mercury in the Thermometer



THERMOMETER

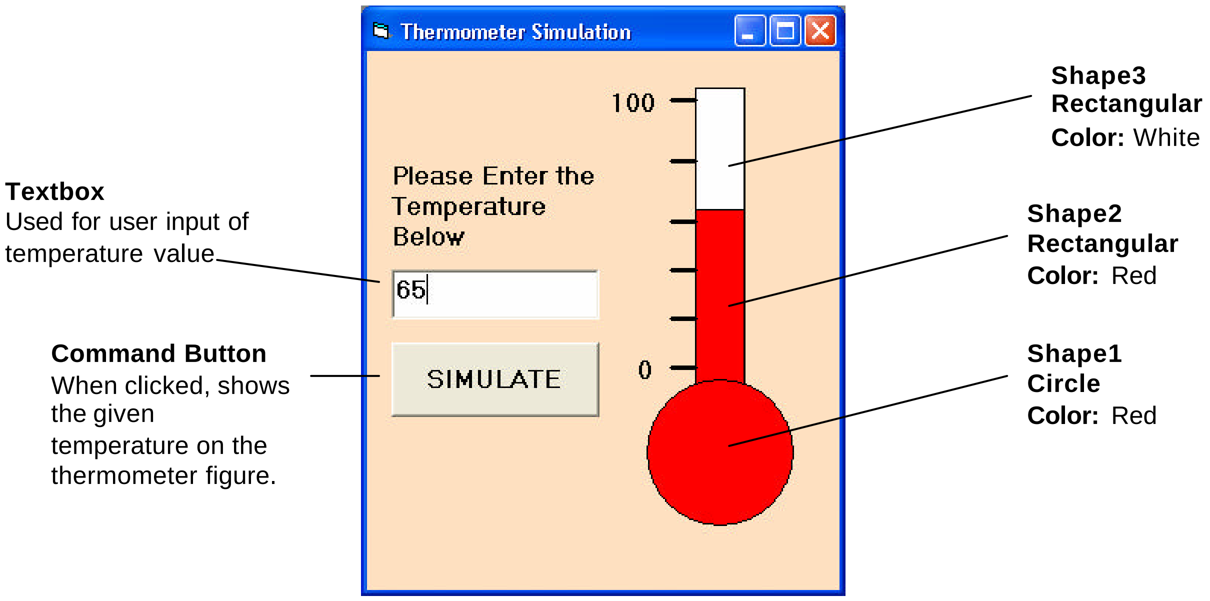
A typical mercury thermometer is used for measuring the temperature of an environment (for example a room) by reading the temperature value shown by the mercury level within a glass tube. The mercury level within the tube changes by the environment temperature.

If we want to make an example Visual Basic program, which simulates the changes in the mercury level in a thermometer for a given temperature, we will make use of some of the animation techniques of Visual Basic, described above.

The necessary work for developing such a thermometer simulation program is explained in the following.

Step 1: Graphical User Interface

Although there are many graphical animation methods in VB, in this example, the thermometer animation performed using the Visual Basic Shape controls, which is the easiest way of creating graphics. For creating the thermometer figure, shown below, 3 independent **shape** controls are used. The definitions and layout of some important controls of the program are shown in the figure below.



Textbox
Used for user input of temperature value

Command Button
When clicked, shows the given temperature on the thermometer figure.

Shape3
Rectangular
Color: White

Shape2
Rectangular
Color: Red

Shape1
Circle
Color: Red

Step 2: Operation Theory and Coding

Basically, the simple thermometer program shown, takes the user input through the textbox provided and with the value of this input, it simulates the change in the mercury level by re-sizing the shapes. The rectangular **Shape3**, which is located at the top of thermometer figure, represents the empty space of the tube. When this shape is resized vertically, it gives the user the sense of changing the level of the mercury in a thermometer tube.

Therefore, to create this sense, the height of the **Shape3** is being indirectly related to the user-entered temperature for animating the mercury motion.

The height of the Shape3 is defined with a **linear inverse interpolation** such as:

$$\frac{T - T\max}{T\min - T\max} = \frac{H - H\max}{H\min - H\max}$$

Where: **T** is the Temperature entered by the user, **H** is the final height of the **Shape3** we want, **Tmin** and **Tmax** are the minimum and maximum scales on the thermometer respectively (for example 0 and 100). **Hmin** is the initial height of the **Shape3** corresponding to the minimum temperature value (**Tmin**) and finally **Hmax** is the height of the **Shape3** in **twips** corresponding to the maximum temperature value (**Tmax**).

Then if we derive an equation for any temperature value between Tmin andTmax, the height of Shape3 in twips can be determined as (from the above relationship),

$$\text{Shape3.height} = H = \frac{(T - T\max)(H\min - H\max)}{(T\min - T\max)} + H\max$$

The following code is written for the implementation of the above relationship to the simulation program. Note that the code also checks whether the temperature entry of the user is within the acceptable limits or not! This is done by an **if-then-end if** block statement.

```
Private Sub Command1_Click()
'variable declarations
Dim temperature, maxtemp, mintemp
Dim maxheight, minheight
'initialization of program variables
'maximum and minimum scales of the thermometer
maxtemp = 100
mintemp = 0
'height of the Shape3 corresponding to the MAXIMUM temperature
maxheight = 135
'height of the Shape3 corresponding to the MINIMUM temperature
minheight = 2775

'initializing the value of given temperature

temperature = Val(Text1.Text)

'CHECK IF THE TEMPERATURE ENTERED IS OUT OF THE LIMITS!
If temperature > maxtemp Or temperature < mintemp Then
    x = MsgBox("THE TEMPERATURE ENTRY IS WRONG!", vbCritical, "WARNING!")
    'exit the procedure
    Exit Sub
```

```
End If
'calculating the necessary shape height by interpolating the minimum and
maximum values of _ shape3.height and the corresponding temperature Values

Shape3.Height = ((temperature - maxtemp) * (minheight - maxheight)) /
(mintemp - maxtemp) + maxheight

End Sub
```

Step 3: Running the Program

As mentioned above, when the program runs, it expects the user to enter a number for temperature to the textbox and click the command button “SIMULATE” after that. The program simply calculates the necessary height of the Shape3 according to the linear relationships and shows the change in the thermometer mercury level for any given temperature between the scale limits.

-END OF CHAPTER 7-