

Graphic Command Design

by Mark Stefanchuk

Here's a topic worthy of a detailed look, **graphic command design**. In most instances this is a complex command made up of multiple elements and multiple types such as lines, circles, boxes, and text.

The elements are generally graphically grouped or added to a cell header. And, the command often has some type of non-graphic data attached to it like a tag, data included in a cell name, a database linkage, data appended to the end of the graphic definition, or a combination of these technologies.

Command design is covered at some level in MDL and VBA training, but usually it takes on a simplified format demonstrating line placement or placement of predefined cells from a cell library and in advanced training sessions attaches a database linkage to the element.

These examples are an important part of the learning process but these examples rarely discuss best practice or use parameters to define the shape, size, or symbology of graphics defined by the command.

So, the result, most of us have used trial and error to discover coding practices. In this article I will demonstrate how you might approach building a graphic command from requirements to placement to editing and in doing so, hopefully help you avoid some trial and error.

Pie Symbol Example

The example is a pie symbol used in right of way maps. This symbol is made up of a circle divided into quadrants by lines. And, each quadrant contains text describing the portion of a property that the symbol represents. The new symbol will look like this.



Figure 1 Right of Way Pie Symbol

To create the example I've chosen to use VBA, but the command can, and has been developed in MDL. While much of the design is similar, the coding techniques are, of course much different. Deciding to use MDL or VBA depends on several factors. Here are a couple.

1. MDL applications look more like MicroStation, and many interface tools, like color pickers, weight pickers, and toolboxes are built in.
2. VBA is very easy to code and building interface controls is fast. Excellent integration with Windows and Windows API tools.

If you want a professional looking application, one that looks and integrates well with MicroStation then use MDL. On the other hand if you have limited time or need a Windows style interface then consider VBA. There are other choices but they are just variations on these.

Requirements

Let's start with some requirements, also known as features.

1. Control options for type, tract number, pie number and total pies.
2. Text Controls - Numbers Only.
3. Pie placement independent of view rotation.
4. Pie placement respects active angle.
5. Pie placement respects active scale.
6. Elements in symbol must be grouped together.
7. Edit tool for changing type, tract number, pie number and total pies.

TIPS IN HERE

Something that I find amusing in reading articles like this one is that the requirements magically appear and are well defined at the beginning of the project. A necessity in writing an article, but in reality you can't expect all requirements to be defined so early.

Your end user won't have the experience to know that their new command will need to respect active scale and view rotation. It won't occur to them for instance that you don't know they apply active scale to their symbols. Arguably it's probably your job to ask them about these things, so some domain experience may be required. But the reality is, you won't catch everything.

What if you build a form and validate text fields for numbers, as per request but, the client, upon using the command discovers that they need to be able to use letters. Are you going to tell them the validation method can't be changed because you have fifty more features to build? Probably not. Expect some do-over.

This type of feedback only comes from an interactive development schedule, one where your end user is involved in the testing and requirements analysis throughout the development cycle.

The Pie Symbol is A Primitive Placement Command

I created a new VBA application called *Pie*. And then added a user form with a button labeled *Place*. This button will start a primitive placement command. Add a second button labeled *Edit*. More on that later.

Change the *ShowModal* property of the form to *False*. With *ShowModal* set to *True* your form won't allow you to work in *MicroStation* until the form is dismissed. This means pie placement is disabled too.



Figure 2 Pie Command User Form

Tips - v8

Level Display: Save settings

Whenever we use the *Level Display* dialog box to adjust levels, many times a lot of the levels we have turned *Off* will turn *On*. I've tried using saved views, but when I recall the view, these wrongly turned *On* levels stay on.

You must choose *File>Save Settings* to save changes made via the *Level Display* dialog box.

CAD

Next double click on *Place*. This opens the code window for the form and it adds some code for the *Place* button. In the *Place* button's *Sub* we will add code for starting a primitive placement command. It will look like this.

```
Private Sub cmdPlace_Click()  
    CommandState.StartPrimitive New classPie  
End Sub
```

I renamed *Module1* to *modPie* and I added the following code. This will launch the command form.

```
Public Sub LoadPieForm()  
    Load frmPie  
    frmPie.Show  
End Sub
```

Now we are ready to create the command class for the *Pie*. Insert a class into the project and call it *classPie*. This class implements *IPrimitiveCommandEvents* telling *MicroStation* that the class is a placement command.

Several built in actions such as dynamics, datapoint, and start are added to this class. Since these actions do not automatically get added to the class I either import a template, or I simply cut and paste from an older command.

Rather than include the complete listing here I invite you to download the example code from

<http://www.markstefanchuk.com/>

When the *Place* button is clicked it calls *classPie*.

IPrimitiveCommandEvents_Start() is entered automatically. This is where we initialize variables, prompt for input, and since the action of this command is similar to placing a cell dynamics is started here too.

```
Private Sub IPrimitiveCommandEvents_Start()  
    ShowCommand "Place Pie"  
    ShowPrompt "Enter Pie Origin"  
    CommandState.EnableAccuSnap  
    CommandState.StartDynamics  
End Sub
```

TIPS IN HERE

Command Controls

The pie command needs controls to set symbol type, tract number, pie number, and total pies. These will be added to the main control dialog.



Figure 3 Pie Controls On User Form

Upon initialization of the form we will populate the combo box and set the `ListIndex` to `0` so that the first item in the list is displayed when the form is loaded. To do this the form code needs and initialize sub.

```
Private Sub UserForm_Initialize()  
    Me.cmbType.AddItem "PAR"  
    Me.cmbType.AddItem "TDE"  
    Me.cmbType.AddItem "TCE"  
    Me.cmbType.AddItem "PDE"  
    Me.cmbType.AddItem "BEF"  
    Me.cmbType.AddItem "URP"  
    Me.cmbType.ListIndex = 0  
    Me.cmbType.Style = frmStyleDropDownList  
End Sub
```

The control is populated and initialized, and the final step makes sure that the style of the combobox style is set to `fmStyleDropDownList` disabling new entries. Setting style to select only enforces basic control on option standards, and is appropriate for this command's design.

Validate Text Controls

Adding and initializing controls is pretty standard but the text controls are required to be numbers only. This is an excellent requirement. It forces us to think about validating the user's input.

With VBA this is a simple test using `IsNumeric` and the test is performed in the controls change event. If false, then reset the `Value` of the text box back to `1`.

Code 1

```
Private Sub txtTract_Change()  
    If (IsNumeric(Me.txtTract.text)) Then  
        Me.txtTract.Value = Me.txtTract.Value \ 1  
    Else  
        Me.txtTract.Value = 1  
        MsgBox "Please type a number for Tract Number."  
    End If  
End Sub
```

The number must also be an integer not a double, so we will apply the `\` operator to the textbox value divided by `1` to convert any doubles to the an integer value. See **Code 1** below.

The pie number and total pies are validated in the same way.

As a rule of thumb, validate early. I prefer to validate when control data changes. This is the earliest possible place to test the data. This allows me to prompt the user to correct the mistake immediately.

Otherwise if I wait until the user clicks the `Place` button I have to stop the command if an error is found and display a message indicating what happened and why. If there is more than one mistake the user may attempt to place the pie symbol three or four times before getting it right and that's not a good user experience.

Draw The Pie Symbol

The next task is to tell the dynamics function what to draw. I abstract this method making a common but separate method rather than including the draw routines in the dynamics handler directly. The separate method allows me to call it from several places making the code easier to maintain, and evolve.

`DrawPie` handles several requirements for our command.

1. Calls routines for drawing all of the components.
2. Adds these elements to a cell header so that the elements will be grouped together. Using a cell makes transformations and editing features very simple. Graphic group are difficult to manage and users can make changes to the graphic groups too easily jeopardizing data integrity.
3. Transforms the cell from the origin where it was created to the cursor's placement point and then applies active angle, scale, and makes the symbol independent of view rotation.
4. Adds the pie to the design file.

TIPS IN HERE

With almost every graphic command the smartest approach is to draw all of the elements about the design file origin (0,0,0) combine the elements into a cell and then use MicroStation's transform functions to move, scale, rotate the cell to the location of the cursor. We draw at 0,0,0 primarily because it simplifies our mathematics. Trigonometry is eliminated and we can use a standard coordinate system to position elements.

One master unit was used as the default radius of the pie and text sizes have been defined to fit within this radius.

Using 1 MU simplifies scaling. Alternatively you may want to size the symbol and text sizes independently.

In this case a preferences manager might be appropriate. If implementing a preferences form I recommend saving the settings to an independent control file such as an ini file and when drawing the pie read the settings from this file rather than from the preference control. The control file makes the command operate independently so you don't have to ensure that the form is loaded before starting the Place Pie command.

```
Private Sub DrawPie(point As Point3d, View As View, _
    drawMode As MsdDrawingMode)
    Dim oCell As CellElement
    Dim oElts(7) As Element
    Dim center As Point3d

    On Error GoTo DrawPieError

    ' draw pie at design origin
    center.X = 0: center.Y = 0: center.Z = 0

    ' add the pie circle
    Set oElts(0) = CreateArcElement2(Nothing, center, 1, 1,
    Matrix3dIdentity, 0, 2 * Pi)

    ' draw pie quad lines
    Set oElts(1) = PieAddLine(center, 1, 0)
    Set oElts(2) = PieAddLine(center, 0, 1)

    ' add pie text
    Set oElts(3) = PieAddText(center, 0 - 0.5, 0.125, _
    frmPie.txtTract.text, msdTextJustificationCenterBottom, 0.5, 0.5)

    Set oElts(4) = PieAddText(center, 0.0625, 0.125, _
    frmPie.cmbType.text, msdTextJustificationLeftBottom, 0.375, 0.25)

    Set oElts(5) = PieAddText(center, 0 - 0.5, 0 - 0.125, _
    frmPie.txtPieNumber.text, msdTextJustificationCenterTop, 0.5, 0.5)

    Set oElts(6) = PieAddText(center, 0.5, 0 - 0.125, _
    frmPie.txtTotalPies.text, msdTextJustificationCenterTop, 0.5, 0.5)

    ' add elements to cell
    Set oCell = CreateCellElement1("PIE", oElts, center, False)
    ' move pie to point and apply active settings
    PieTransform oCell, point, View

    ' add pie to design file
    If drawMode = msdDrawingModeNormal Then
        ActiveModelReference.AddElement oCell
        oCell.Redraw drawMode
    Else
        oCell.Redraw drawMode
    End If
    ' clean up objects
    Set oCell = Nothing
    Dim i As Integer
    For i = 0 To 6
        Set oElts(i) = Nothing
    Next

    Exit Sub
DrawPieError:
Debug.Print "Error: Draw Pie Dynamics"
End Sub
```

Both `PieAddLine` and `PieAddText` are similar in design. They are both private functions and return the new element. `PieAddLine` is shorter so lets look at it.

```
Private Function PieAddLine(center As Point3d, _
    xOffset As Double, yOffset As Double) As LineElement
    Dim startpoint As Point3d, endpoint As Point3d
    On Error GoTo PieAddLineError

    startpoint = center
    startpoint.X = startpoint.X - xOffset
    startpoint.Y = startpoint.Y - yOffset
    endpoint = center
    endpoint.X = endpoint.X + xOffset
    endpoint.Y = endpoint.Y + yOffset
    Set PieAddLine = CreateLineElement2(Nothing, _
        startpoint, endpoint)

    Exit Function
PieAddLineError:
Debug.Print "Error: Add Line to Pie"
End Function
```

All the function does is create a lines with start and end points offset about the design file origin. All of the elements, the arc, lines, and text are added to an element collection `oElts`. This collection of elements is then used to create the cell, "PIE" at the design file origin.

Once we have a cell we can translate it to the cursor location so that it shows up in dynamics. We can also apply other transformations like active scale, angle, and apply a correction for view rotation. That's what `PieTransform` does.

```
Private Sub PieTransform(oCell As CellElement, _
    point As Point3d, View As View)
    On Error GoTo PieTransformError

    ' move to point
    oCell.Transform Transform3dFromXYZ(point.X, point.Y, point.Z)

    ' active scale
    Dim ScaleMatrix As Matrix3d
    ScaleMatrix = Matrix3dFromScale(ActiveSettings.Scale.X)
    oCell.Transform _
        Transform3dFromMatrix3dAndFixedPoint3d(ScaleMatrix, point)

    ' rotate by active angle
    Dim rMatrix As Matrix3d
    Dim axis As Point3d
    axis.X = 0: axis.Y = 0: axis.Z = 1
    rMatrix = _
        Matrix3dFromVectorAndRotationAngle(axis, ActiveSettings.Angle)
    oCell.Transform _
        Transform3dFromMatrix3dAndFixedPoint3d(rMatrix, point)

    ' view rotation
    Dim rAngle As Double
    If (Matrix3dIsXYRotation(View.Rotation, rAngle)) Then
        rMatrix = Matrix3dFromVectorAndRotationAngle(axis, 0 - rAngle)
        oCell.Transform _
            Transform3dFromMatrix3dAndFixedPoint3d(rMatrix, point)
    End If

    Exit Sub
PieTransformError:
Debug.Print "Error: Pie Transform"
End Sub
```

Despite the length of some of these function names they aren't that difficult to understand. And it's actually fantastic that all of these functions are available to us.

`Transform3dFromXYZ` takes the cursor location point components and moves the cell to that location. We didn't have to figure out how to do this ourselves MicroStation VBA takes care of the math for us.

Similarly, scale and rotation are handled by the function `Transform3dFromMatrix3dAndFixedPoint3d` where scale is changed by applying a scale matrix, and rotation is handled by converting the angle to a rotation matrix. There are excellent examples in the MicroStation VBA help, so check it out.

So what would happen if I decided to draw the elements without using the cell? Can't I apply transformations to individual elements? Well yes you can. But then you must call the transform element for every element in `oElts` independently making the code messy.

And what about these transformations? What would be different if I draw the elements at the cursor point and didn't use the transform functions?

Well these are good questions too. Let's just consider the circle and the lines? If I draw the elements without scale or rotation then there is no difference, I simply replace center with point. But, if you rotate now you must find the start and end point of the dividing lines. Where should they be? Let's see $\tan \alpha = \text{opp/adj}$, and if alpha is 33 degrees then . . . I'd at least study rotation matrix functions. And we haven't started to figure out how much to rotate the text or where the text origins need to be. Draw at the design file origin and transform. It's easier.

Editing The Pie Symbol

So far we have taken care of the first six requirements. But, in my opinion a graphic command with data elements is not complete unless that data can be changed. To do that we need an edit command.

Next, insert a new class and call it `classEdit` and implement `IlocateCommandEvent`s. Double click on the Edit button we created on our user form and add the following subroutine.

```
Private Sub cmdEdit_Click()  
    CommandState.StartLocate New classEdit  
End Sub
```

*Put a finger on the pulse of **MicroStation*** Subscribe to **ControlAltDelete**

Please enter my annual subscription to
ControlAltDelete with 4th Qtr 2003/ 1st Qtr 2004/ 2nd Qtr 2004 (Please circle one).

I have enclosed payment of:

In Australia	\$A38.50 (GST Inclusive)
In NZ, Asia-Pacific, USA & CAN	\$A50.00
In Europe & Others	\$A60.00

Name:
Company:
Address:
City: State Zip
Country:
Tel/ Fax:
Email:

Make your cheque payable to:
Pen and Brush Publishers
PO Box 808, Hawthorn VIC 3122 Australia
Email: penbrush@ozemail.com.au
Tel: +61 3 9818 6226
<http://www.penbrush.com>

We accept VISA and MASTERCARD

Card Number:
Expiry Date:
Card holder Name:
Signature:

Amount \$ _____



This will call the locate function which will implement standard events similar to those found in classPie. The most obvious difference will be that the locate classEdit includes an Accept function instead of a data point function. The accept function tests the clicked element for cell type and name. Checking for cells named PIE is probably a good enough test, but if there may be conflicts with existing library cells then use a more unique cell name.

```
Private Sub ILocateCommandEvents_Accept(ByVal Element As Element, _
                                       Point As Point3d, ByVal View As View)

    Dim oCell As CellElement
    If (Element.Type = msdElementTypeCellHeader) Then
        Set oCell = Element
        If (StrComp(oCell.Name, "PIE", vbTextCompare) = 0) Then
            ChangePieData oCell
        End If
    End If
End Sub
```

If the test passes then the ChangePieData sub routine is called. This sub gets the sub elements from the cell and adds them to an element enumerator. The enumerator allows us to step through the cell using a do while loop.

```
Private Sub ChangePieData(oCell As CellElement)
    Dim ee As ElementEnumerator
    Dim oTxt As TextElement
    Dim count As Integer

    On Error GoTo ChangePieDataError

    Set ee = oCell.GetSubElements
    count = 0
    Do While ee.MoveNext
        If (ee.Current.Type = msdElementTypeText) Then
            ee.Current.Redraw msdDrawingModeErase
            Select Case count
                Case 0
                    Set oTxt = ee.Current
                    oTxt.text = frmPie.txtTract.text
                Case 1
                    Set oTxt = ee.Current
                    oTxt.text = frmPie.cmbType.text
                Case 2
                    Set oTxt = ee.Current
                    oTxt.text = frmPie.txtPieNumber.text
                Case 3
                    Set oTxt = ee.Current
                    oTxt.text = frmPie.txtTotalPies.text
            End Select
            ee.Current.Redraw msdDrawingModeNormal
            ee.Current.Rewrite
            count = count + 1
        End If
    Loop

    ' clean up
    Set ee = Nothing
    Set oTxt = Nothing

    Exit Sub
ChangePieDataError:
    Debug.Print "Error: Change Pie Data"
End Sub
```

Within the `Do While` each element in the cell is tested for type text. A counter is used to indicate which text element is current and the new data is assigned to it. The data comes from the same form controls used by the place command. Not only is control re-use convenient for development, but it is also good interface design since the users only have to learn to use one set of controls.

The counter works because we know what order the text elements were added to the pie.

You may recall that in the pie draw routine text was added using `PieAddText` and were added as elements 3, 4, 5, and 6. Knowing that there are only 7 elements in the cell we could take out the test for text and select case 3 to 6 instead of 0 to 3. Your choice here as the change is unlikely to have a significant impact on performance.

Improvements

Several additional features would make the pie command more complete.

1. Build a preference manager to handle symbology based on pie type.
2. Add leader lines extending the pie tool to a compound command. Leader lines require a new collection of editing tools - delete leader, drop vertex, add leader - and so I'll save it for a future article.
3. An `ini` file is a simple tool to implement that would allow you to save control data and form position between sessions.
4. Make the pie command more generic by including preferences for text and pie size.
5. Protect administrative preferences.
6. Improve validation by matching total pies to pie number when a pie number is greater than the current setting for total pies.
7. Limit the value for tract number, pie number, and total pies to be > 1 but < 100 .
8. Consider adding a post validation tool to examine the sequence of pies and compare the pie count per tract-type to the total count in the pie symbol.

About The Author

Mark Stefanchuk is a consultant and contract programmer specializing in business and design workflow automation. He was co-founder and principle architect of cadgurus.com a web community for MicroStation professionals.

More recently Mr. Stefanchuk continues to work on projects involving MDL, VBA, and Microsoft Windows technology. Mark may be contacted by sending email to markstefanchuk@yahoo.com.

CAD

Tips - v7 -> v8

Associations with Group Holes

I have some hatched solids in /J. I need to cut some rectangular holes from inside the patterned area. I would like to keep them associative because the shapes of these areas are subject to change. I can use the `Modify Element` tool to stretch them. But I have problems getting 'group holes' to work.

In / J, you can hatch group holes, but will not be able to modify the element and have the hatch adjust accordingly. This functionality is available in MicroStation v8.

CAD

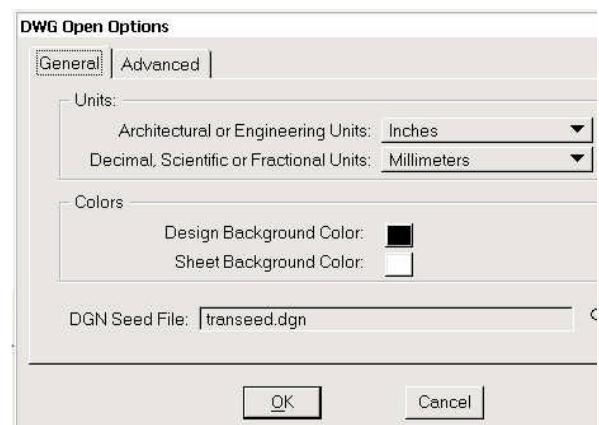
Tips - v8.1

Background Colour in DWG

When I open an AutoCAD drawing using v8, the background colour is always white but when I'm in AutoCAD the background is black. How do I change the background colour from white to black in v8?

The background colour for DWG files is not stored in the file itself, but is instead a program setting.

You can change the background colour of sheet and design models in DWG files before opening them in MicroStation by clicking the `DWG Options` button and then use the color pickers for `Design Background Color` and `Sheet Background Color` respectively.



CAD

TIPS IN HERE