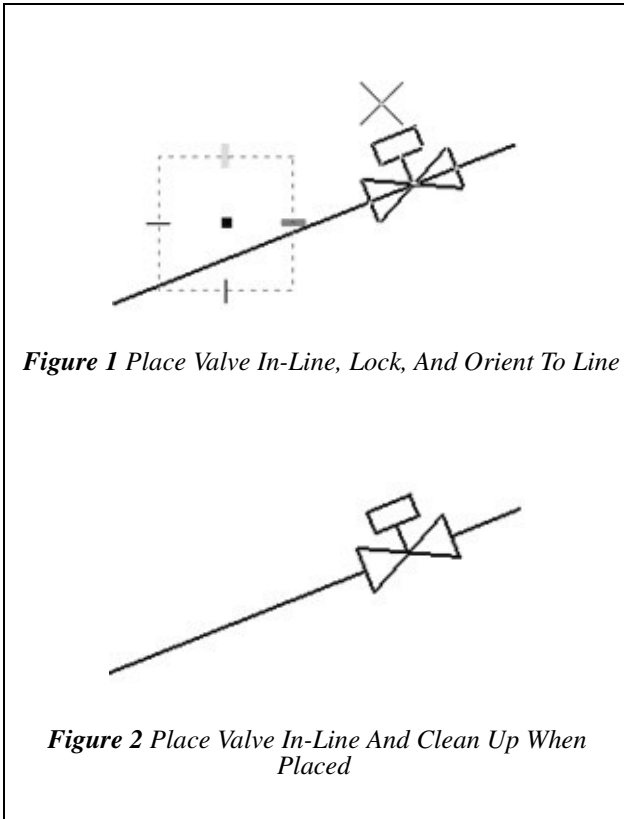


Place In Line and Clean Up

by Mark Stefanchuk

Here's an interesting MicroStation development problem. Create a command that has the ability to place a component, such as a pipe valve or a door cell in-line and then have the line break and clean up automatically. Further, make the component orient to the line and lock onto the line.



How many times have we discovered a user who copies an existing component from a drawing or worse, watched a designer draw the lines and arcs of a door symbol that you know has to exist in a cell library? Really, it happens.

The same user then cuts the wall line – thank goodness he knows how to do that, and then drags the door into the opening eyeballing its location. Arguably this is a training problem and not simply an automation work flow problem.

Maybe, if you can demonstrate a better way the user will be motivated to work a little smarter. If only you could place components in-line and clean up the line automatically you may be able to build smart tools even for your better users. Here's one way to solve this problem, using VBA of course.

Command Design

You may recall that in articles published last year in C-A-D I introduced you to command design where you learned how to place symbols (a primitive command) and edit those symbols (a locate command). Each type of command was created with a VBA Class Module and the primitive and locate interfaces were included respectively.

With our in-line command we want the component to take on characteristics of the line.

In this case it will be orientation, and the symbology of the line (level, color, weight, and style). So, to obtain these characteristics, our command should first identify the line – a locate command. But wait, we need to draw and place the component and this means we should use a primitive command design, right?

Well, you can and you would write into your data point handler all the logic needed to locate the line. But I find it much easier to let MicroStation (Bentley developers) do the hard work for me. So I combine both starting with a locate command and then calling the primitive command after the user has identified and accepted the line.

Command Details

The user is first prompted to identify the line. Upon accepting the line the program saves this element and the accept point in global variables. Generally I save these in a common module, or if my program is small I will define the global variables in my command module – think file where commands are found, not space ships. The primitive command is then called to draw the component. The locate command is quite simple.

The interesting stuff occurs in the data point handler.

```
Option Explicit
Implements ILocateCommandEvents

' DATAPOINT HANDLER
Private Sub ILocateCommandEvents_Accept _
    (ByVal element As element, point As Point3d, _
    ByVal view As view)
```

Save the element and the accept point into global variables since we can't pass variables directly between class modules.

```
Set gPipeElt = element
gAcceptPt = point
```

Start the primitive command.

```
CommandState.StartPrimitive New classDrawValve
End Sub
```

The remaining parts of the locate command are similar to other examples published. For the complete example please refer to the Place Inline download located on my web site <http://www.markstefanchuk.com/cad.aspx>.

The second part of the command draws the component (you can replace this with a cell selection from a cell library), orients the component, locks it to the line, and matches the symbology. That seems like a considerable amount of code, but most of the code in my example just draws the valve. The interesting bits follow.

Like my other commands, this one programmatically draws graphic elements and saves them into a cell element. The cell is then transformed (rotated, moved, and scaled) based on common user options.

In my previous articles the cells were moved from the origin where the cell is drawn to the cursor location. In this case however, I want the component to "stick" to the line. This can be accomplished by getting a perpendicular projection from the cursor location to the line element. Like this,

```
prjPoint = gPipeElt.AsLineElement.ProjectPointOnPerpendicular _
    (point, Matrix3dIdentity)
```

Where the point is the cursor location. And it's this new prjPoint that is used to move the cell and lock it to the line. To orient the cell to the line we our common transformation method applies the following.

```
' rotate
rotAngle = getRotationAngle(gPipeElt, gAcceptPt, view, resetFlag)
```

All that getRotationAngle does is to calculate vectors between the accept point and the line endpoints. It then uses the first vector to calculate the angle between it and the global X-axis, which is the second vector.

```
Public Function getRotationAngle(elt As element, aPt As Point3d, _
    view As view, resetFlag As Integer) As Double
    Dim vec1 As Vector3d, vec2 As Vector3d
    Dim rotAngle As Double

    On Error GoTo cError

    'get vectors in plane of view
    If Point3dDistance(aPt, elt.AsLineElement.StartPoint) > _
        Point3dDistance(aPt, elt.AsLineElement.EndPoint) Then
        vec1 = Vector3dFromMatrix3dTimesVector3d(view.Rotation, _
            Vector3dSubtractPoint3dPoint3d _
            (elt.AsLineElement.StartPoint, _
            elt.AsLineElement.EndPoint))
    Else
        vec1 = Vector3dFromMatrix3dTimesVector3d _
            (view.Rotation,
            Vector3dSubtractPoint3dPoint3d _
            (elt.AsLineElement.EndPoint, _
            elt.AsLineElement.StartPoint))
    End If
```

Here we let the user decide the mirror option of the component. Since the example we are using is non-symmetrical we allow the user to click reset to change the direction of the second vector.

```
If resetFlag = 0 Then
    vec2 = Vector3dFromXY(1#, 0#)
Else
    vec2 = Vector3dFromXY(-1#, 0#)
End If
```

Get the angle.

```
rotAngle = Vector3dAngleBetweenVectorsXY(vec2, vec1)
```

Handle angles less than 0.

```
If rotAngle < 0# Then
    rotAngle = rotAngle + (2 * Pi)
End If

getRotationAngle = rotAngle

Exit Function
cError:
    Debug.Print "getRotationAngle" & Err.Description
End Function
```

At this stage the command has drawn the graphics shown in figure 1. The next step is to accept placement and clean up the line. This happens when the user gives a final data point. In MicroStation VBA terms we would say, when draw mode equals normal draw.

And, the following happens during normal draw.

```
ActiveModelReference.AddElement ocell
ocell.Redraw DrawMode
```

Make sure that the global pipe element is defined and not equal to nothing. If gPipeElt is equal to nothing then an error will be raised when it is referenced.

```
If Not gPipeElt Is Nothing Then
    Dim iPts() As Point3d
    Dim Partial1 As element, Partial2 As element
    Dim oTmpArc As ArcElement
```

There is more than one way to accomplish the discovery of cut points. I like this method just because it works. I have and still use cell connect points, which are more appropriate when placing pre-defined cells like those found in a cell library. But using an arc to find the intersections works very well when you control the size of the component graphics.

```
Set oTmpArc = CreateArcElement2(Nothing, origin, ftgSize / 2, _
    ftgSize / 2, Matrix3dIdentity, 0, 2 * Pi)
```

Move the arc (a circle) so that its center is at the component center.

```
FittingTransform oTmpArc, prjPoint, view, resetFlag
```

Find the intersection points.

```
iPts() = gPipeElt.AsLineElement.GetIntersectionPoints _
    (oTmpArc, Matrix3dIdentity)
```

Make sure that we found intersection points.

```
If UBound(iPts) > -1 Then
```

Use the intersection points to perform a partial delete. And then redraw the components based on the elements returned by this method. Don't forget that you must remove the old element. Do this before adding the partial elements. If you don't your user will have to refresh the view before she can see the new elements.

```

gPipeElt.PartialDelete Partial1,
    Partial2, iPts(0), iPts(1), iPts(1), view
gPipeElt.Redraw msdDrawingModeErase
ActiveModelReference.RemoveElement gPipeElt
Set gPipeElt = Nothing
ActiveModelReference.AddElement Partial1
Partial1.Redraw msdDrawingModeNormal
ActiveModelReference.AddElement Partial2
Partial2.Redraw msdDrawingModeNormal
End If
End If

```

The partial delete method applies all data and symbology of the original to the partial elements it creates. So there is no need to set the graphic symbology or any user data that the original element may have had connected to it.

Finally, lets step back a little. The element symbology is set on the individual components before the cell is created.

```

If Not gPipeElt Is Nothing Then
    rc_setFittingSymbology oElts(2)
End If

```

I do this so that I can control the level, color, and weight, and style of the cell elements independently. When you apply graphic symbology attributes to the cell element directly then all of the sub elements will get the same settings. I don't necessarily want that. For completeness here's the code for setting symbology.

```

Private Sub rc_setFittingSymbology(oElt As element)
    On Error GoTo cError
    oElt.color = gPipeElt.color
    oElt.LineWeight = gPipeElt.LineWeight
    oElt.Level = ActiveDesignFile.Levels(gPipeElt.Level.Name)
    oElt.LineStyle = gPipeElt.LineStyle

    Exit Sub
cError:
    Debug.Print "rc_setFittingSymbology " & Err.Description
End Sub

```

Unfortunately, too many MicroStation users place lines, circles boxes and text. And, not enough of us attempt to automate the processes. Presented here were just the first steps in design automation, but consider further that each component might have information connected to them describing things like vendor information, maintenance work orders, size, and price.

With a little ingenuity you can create analytics to measure flows, weight, or automate schedules, and reports. Taking design automation and your programs to this level will yield outstanding results in design management. It will make your organization stand out from everyone else.

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 mark.stefanchuk@gmail.com.

CAD