Autodesk University 2010 LEARN. CONNECT. EXPLORE.

AutoCAD .NET: Practical Examples of Customizing AutoCAD Entity Behavior

Stephen Preston – Autodesk, Inc.

CP230-1V

In this class we will work through some practical examples of customizing AutoCAD entity behavior using the .NET Overrule API introduced in AutoCAD 2010. Coding examples will be presented in VB.NET, but the concepts demonstrated apply to all other .NET programming languages.

About the Speaker:
Stephen has been a member of the Autodesk Developer Technical Services (DevTech) team since 2000, first as a support engineer and now as manager of the EMEA (Europe, Middle East, and Africa) team. In those roles, his responsibilities included supporting the AutoCAD APIs, including ObjectARX and AutoCAD .NET, as well as AutoCAD OEM and RealDWG™ technologies. Currently, he manages the Developer Technical Services Team in the Americas and serves as Workgroup Lead, working closely with the AutoCAD engineering team on future improvements in the AutoCAD APIs.

Stephen started his career as a scientist, and has a Ph.D. in Atomic and Laser Physics from the University of Oxford.

stephen.preston@autodesk.com

Introduction

Question: When is a line not a line?

Answer: When it represents a real world object such as a wire or a pipe.

Overrules are a powerful API first introduced in AutoCAD 2010. They allow us to 'stylize' or 'customize' the behavior of standard AutoCAD entities, so (for example) the lines in a drawing really do look like pipes. Overrules are a simpler to use version of the 'custom entities' that have been available to ObjectARX programmers since AutoCAD R13. Overrules can be implemented in a .NET programming language (e.g. VB or C#) or in ObjectARX (unmanaged C++). We'll work exclusively with the .NET API in this document and in the virtual presentation

This document is an update of the handout provided for Virtual Class *CP9310 - Customizing Entity Behavior in AutoCAD .NET* presented at AU Virtual 2009. The recording of that presentation is still available on AU Online, and I recommend you review it and the supporting samples if you're new to Overrules. There's no need to read the 2009 version of the handout, as all the information from that has been carried forward (and extended) in this version.

All samples in the handout and in the class are in VB.NET. You can easily translate to C# using an online translator. I've found http://www.developerfusion.com/tools/convert/vb-to-csharp/ to be very effective. The samples were created using Visual Studio 2008 Professional, but you can use any version of Visual Studio 2008 or 2010 – including the Express editions.

This handout (and the class) assumes that you are familiar with the .NET Framework and the AutoCAD .NET API; and that you understand how to create, build and run an AutoCAD .NET addin. If you're not familiar with these, then please work through some of the resources listed in the Further Reading section before the class.

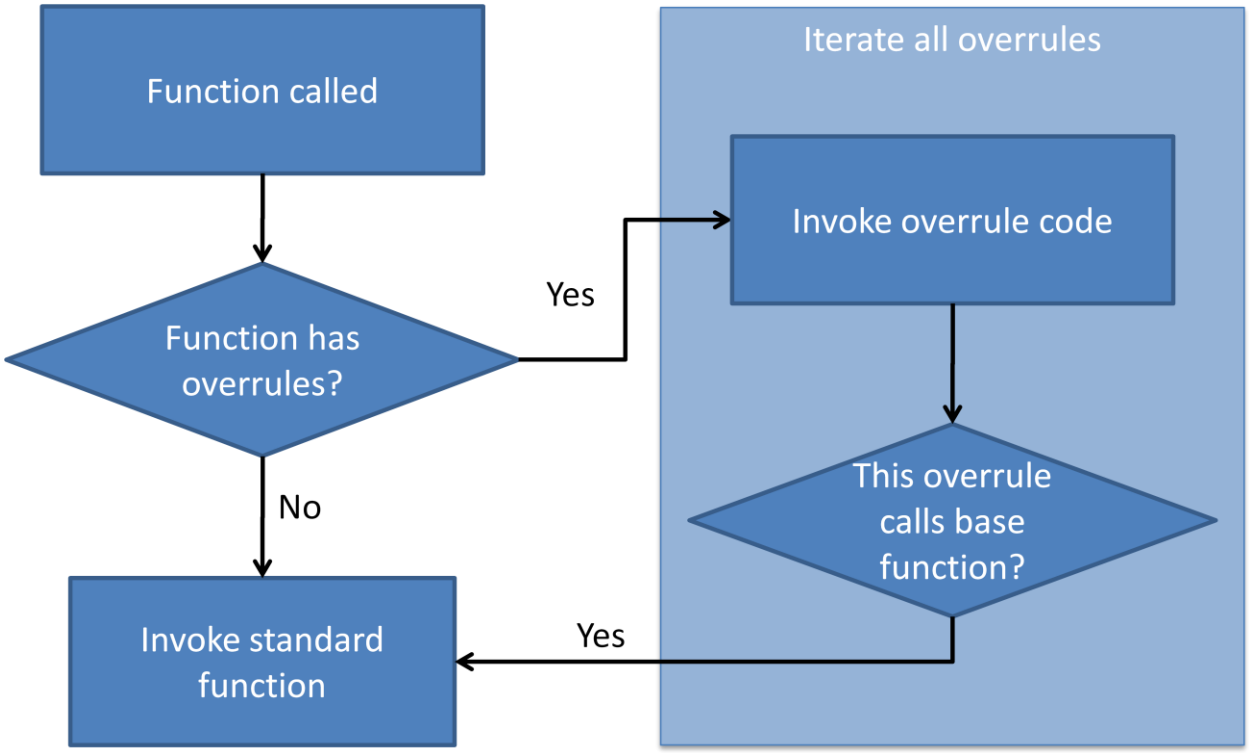
What is an overrule?

So what exactly is an overrule? The implementation is quite simple. Let's take the example of an entity drawing itself:

When AutoCAD wants an entity (let's say a line) to draw itself, it calls the WorldDraw function implemented by the entity – every drawable entity in AutoCAD implements a WorldDraw function. Adding an overrule to the line essentially tells AutoCAD, *"instead of calling this line's WorldDraw function, please call this function I supplied instead"*. The function we supply may choose to call the line's 'native' WorldDraw function as well, but it doesn't have to. And we can even provide more than one overrule, so several of our functions get called (one at a time) when AutoCAD wants the line to draw itself.

Below is a diagram for that process. Putting it simply:

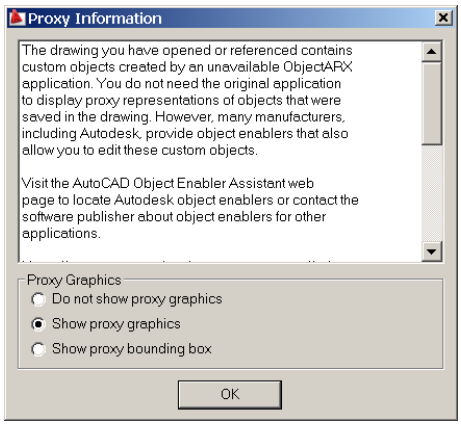
- When AutoCAD calls a function on a standard entity (for example asking the entity to draw itself) the entity checks if it has any applicable Overrules applied to it.
- If it doesn't, the normal function is invoked (the normal graphics are displayed)
- If there are applicable overrules, then each overrule is invoked in turn (we draw our custom graphics).
- An invoked overrule can choose whether to also call the normal function implementation (we can decide whether to also ask the entity to display its normal graphics).



The Overrule API consists of a number of different behavior overrules that we can apply to an object or entity. We'll list them all later.

Why not 'Custom Entity .NET'?

If you've been using AutoCAD for any time at all, you'll almost certainly have heard of the mystical 'custom entity'. If you're an AutoCAD user, you'll have discovered them through the helpful 'proxy information' dialog that appears when opening a drawing with a custom entity in it for which you don't have the Object Enabler.



If you're programming AutoCAD, then you probably know that to implement a custom entity means you have to learn C++ and use the ObjectARX API. If you already know ObjectARX, then it's likely you'll have at least experimented with creating a custom entity. It's even part of the ObjectARX training labs[1].

Custom entities are indeed very powerful. We can use the Custom Entity API to create an entity that behaves as if it were a native AutoCAD entity, but exhibits the behavior you program into it. For example, AutoCAD gives you lines, arcs and circles; an ObjectARX programmer can create pipes, flanges and valves. The pipes can draw themselves as pipes, complete with attachments. They can have grip points, stretch points, and snap points. They can even display themselves differently depending on the view direction (e.g. a plan view or ISO view)[2].

However, the power of the Custom Entity API is also its weakness:

- It's very hard to write a custom entity that behaves as the user would expect when used with every AutoCAD command.
- It's very easy to make a mistake in your implementation that corrupts a DWG file beyond recovery.

Because your custom entity is saved with a drawing, and only works correctly with the 'Object Enabler' you've written for it, you should only consider creating a custom entity as part of your application if you're willing to make these two commitments:

1. To support the users of your custom entity for every future release of AutoCAD.
2. To provide your Object Enabler to anyone who might legitimately be using a drawing with your custom entity in it.

If you're not willing to make those two commitments (especially the first), then you really shouldn't be inflicting a custom entity on your users and customers.

[1] Downloadable from the AutoCAD Developer Center – www.autodesk.com/developautocad.

[2] Some of the best examples of custom entities can be found in Autodesk's AutoCAD vertical applications. For example, AutoCAD Architecture gives you entities representing windows, walls, doors, etc.

The Custom Entity API is a complicated, hard to use, and potentially destructive (if misused) C++ API. Porting a complicated, hard to use, and potentially destructive C++ API directly to .NET wouldn't make it any simpler, easier to use, or less destructive. It would simply open up a complicated, hard to use and potentially destructive API to a new section of the AutoCAD developer community. This was a big reason in our decision to create this new Overrule API. The Overrule API is simpler, easier to use, and less destructive (even if misused) than the Custom Entity API.

The Overrule API simply allows us to tell a standard AutoCAD entity (let's say a line) that we want it to behave differently from a normal AutoCAD line. We can tell it things like:

- When you draw yourself, I want you to look like a 3D pipe, not a line.
- When the user selects you, I want you to display additional grip points to allow the use to alter the pipe diameter.
- When the user moves you, you can only move along the X or Y axis, not diagonally.
- When the user views your properties, they should see additional properties representing pressure rating and bore diameter, and you should tell the user you're called a PIPE not a LINE when they LIST you.

Overrules are inherently safer than Custom Entities because the overruled entity saves itself in a drawing as the native AutoCAD entity and not as a custom entity. Any data associated with the overruled entity is saved as extended entity data. This means we have to try really hard to corrupt a drawing with the Overrule API, whereas automatic drawing corruption is an easy feature to implement with a custom entity.

There are, however, some limitations with using overrules compared to custom entities. The two main differences are:

- Overrules do not store their custom graphical representation in the drawing. This means that, although someone opening a drawing without your application present won't see that nasty proxy dialog (a good thing), they won't see any graphical overrules you implemented either (not a good thing)[3]. The fix for this is relatively straightforward –either tell the user to explode the overruled objects when sending the drawing to someone without our application[4], or implement an automatic explode on save option in your application.
- Overrules are not available for every function exposed by AcDbObject-derived classes you'd normally inherit from to create your ObjectARX custom entity. This means that there are some behaviors that you cannot customize using Overrules.

[3] You can't really trust that a custom entity's proxy graphics are a correct representation either. Someone may have set PROXYGRAPHICS to 0 before saving the drawing, or they may have modified the drawing when the entity was a proxy and couldn't update itself.

[4] Many users of custom entities do this anyway – so their clients don't get proxy warning dialogs.

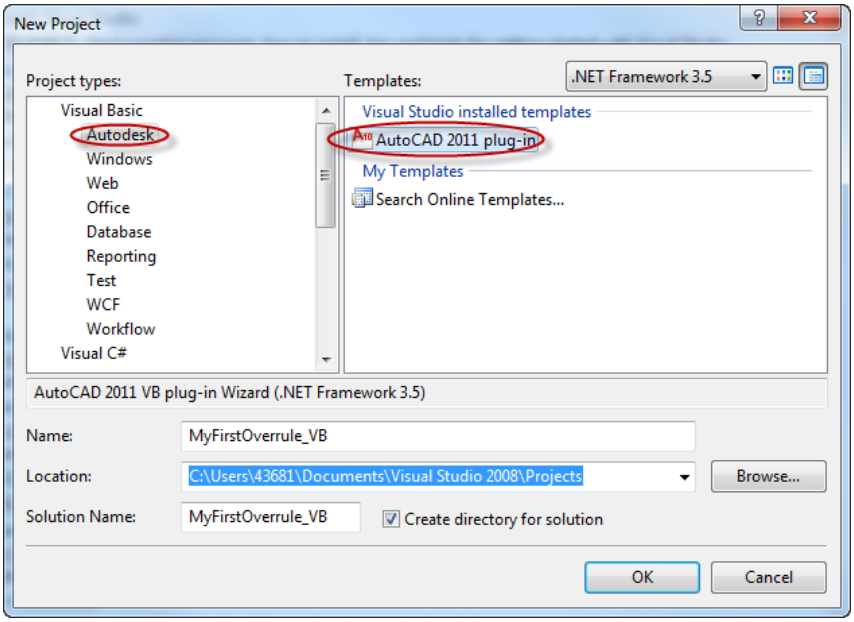
If you can't accept these limitations, then you'll have to use (C++) custom entities instead. And that's ok – Overrules are intended to complement custom entities, not to replace them.

My first overrule

Let's write a simple overrule together. Changing how an AutoCAD entity displays itself is the most common type of overrule, so let's do that.

I'm assuming you know how to implement a basic AutoCAD .NET add-in, so we'll start off by using the AutoCAD 2011 .NET Wizards[5] to create a new project. We use the Wizards just because it sets all our project settings for us. I'm going to use Visual Studio 2008 Professional, but you can also do this in Visual Studio 2010 or one of the Visual Studio Express editions – the Wizards work there as well. I'm also going to write all my code in VB.NET. C# code will follow the same structure and make the same calls to the AutoCAD .NET API[6]. A ReadMe is included with each samples to explain how to use it.

Start Visual Studio 2008, select File->New Project…, and select the 'AutoCAD 2011 plug-in' project template. This is the Wizard template.

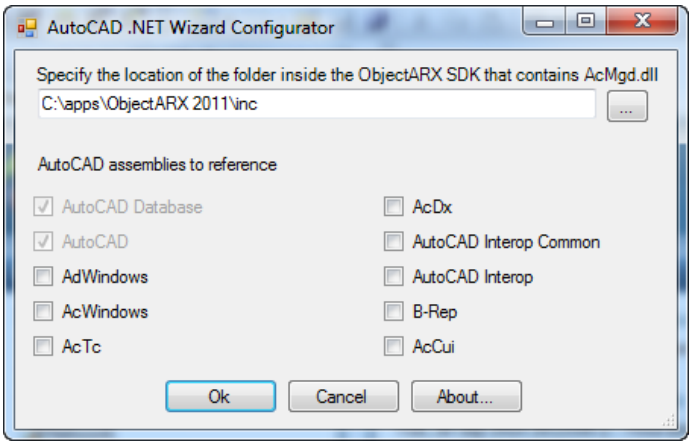


Enter a filename and location and hit OK. The Wizard will then show a dialog asking which .NET reference assemblies we want to use. Keep the defaults and hit OK. (If this is the first time

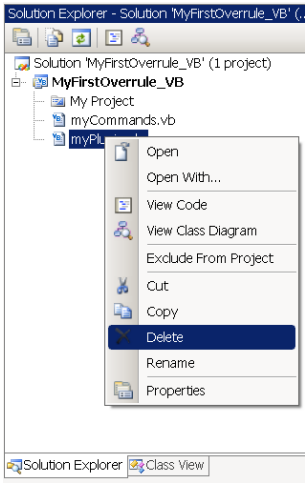
[5] Available from the AutoCAD Developer Center – www.autodesk.com/developautocad.

[6] If you want to translate to C#, you can use a tool like Reflector (http://www.red-gate.com/products/reflector/) to decompile the compiled VB.NET DLL. There are also a number of online translators available, such as http://www.developerfusion.com/tools/convert/vb-to-csharp/.

you've used the Wizards, then you'll also have to enter the location of the 'inc' folder in your ObjectARX 2011 SDK[7] installation).



Before writing any code, go to the Solution Explorer and delete the *MyPlugin.vb* file. We don't need this for this simple demo.



Now we're ready to implement our overrule. We want to customize how an entity draws its graphics, so we're going to derive a new class from DrawableOverrule. Delete all the contents of *myCommands.vb*, and add the following code to the file:

```
Imports System
Imports Autodesk.AutoCAD.Runtime
Imports Autodesk.AutoCAD.ApplicationServices
Imports Autodesk.AutoCAD.DatabaseServices
Imports Autodesk.AutoCAD.Geometry
Imports Autodesk.AutoCAD.EditorInput

Namespace MyFirstOverrule_VB
```

[7] Download the ObjectARX SDK from www.objectarx.com.

```
    Public Class MyDrawOverrule
        Inherits Autodesk.AutoCAD.GraphicsInterface.DrawableOverrule
    End Class

End Namespace
```

You'll notice as you type that the Visual Studio Intellisense fills a lot of your code in for you. The code here is very simple – we've defined a new class that inherits from the DrawableOverrule class. But nothing is going to happen until we overrule a specific function. We want to change how an entity draws itself, so we'll overrule its WorldDraw function. (AutoCAD calls the WorldDraw function of an entity when it wants that entity to draw itself). Inside the class definition (after the line that starts with Inherits), type the following:

```
        Public Overrides Function WorldDraw(ByVal drawable As
Autodesk.AutoCAD.GraphicsInterface.Drawable, ByVal wd As
Autodesk.AutoCAD.GraphicsInterface.WorldDraw) As Boolean
            Return MyBase.WorldDraw(drawable, wd)
        End Function
```

Actually, all you need to do is type `Public Overrides` and select WorldDraw from the drop down list that appears, and the rest will be completed for you.

Let's take a look at the parameters of this function, because this will be new to you if you're not familiar with custom entities.

The first thing to note is that the parent class - `DrawableOverrule` - is part of the `Autodesk.AutoCAD.GraphicsInterface` namespace. As the name suggests, this namespace contains the classes used by AutoCAD's Graphics Interface (or GI) system[8]. A lot of the other classes you'll be using in the `DrawableOverrule` are from this namespace as well.

Now look at the two variables passed into the WorldDraw function:

```
drawable As Autodesk.AutoCAD.GraphicsInterface.Drawable
```

Every entity[9] in AutoCAD is a Drawable. So this variable is just one of the instances of the entity class this overrule has been applied to (the entity that is currently drawing itself). Later in our example, we'll apply this overrule to lines, so we'll be able to cast this Drawable to a Line and query the Line's parameters (i.e. this function would be called by AutoCAD for every Line in the drawing when the drawing graphics are regenerated).

[8] It is common practice to write an interface layer to separate your application from the underlying hardware (or from another software component). That is what the Graphics Interface is doing. The entity can call the same interface methods to draw itself, regardless of the graphics driver being used (e.g. DirectX, or OpenGL, or even a plotter driver).

[9] An entity is what we call an object in AutoCAD that can draw itself (e.g. a line, circle, polyline, block insert, etc.). This is because they are derived from the Entity class. We refer to all objects stored in a DWG database as 'objects' because they are derived from the DBObject class. 'Entities' are also 'objects' because the entity class is derived from the DBObject class.

```
wd As Autodesk.AutoCAD.GraphicsInterface.WorldDraw
```

Don't confuse this WorldDraw class with the WorldDraw function we're overruling. Think of this `GraphicsInterface.WorldDraw` instance (`wd`) as the canvas onto which AutoCAD asks the entity to draw itself. By calling methods exposed by the WorldDraw class, the Line (or whatever) can draw itself in the correct color, linetype, thickness etc (and of course in the correct position in the drawing). We'll see how we use `wd` inside the function in a moment. But first let's add the code to register and activate our Overrule.

Add the following code after the end of your Overrule class definition (i.e. after the End Class statement:

```
    Public Class myCommands

        'Shared member variable to store our Overrule instance
        Private Shared mDrawOverrule As MyDrawOverrule

        ' Define Command "TOGGLEOVERRULE"
        <Autodesk.AutoCAD.Runtime.CommandMethod("TOGGLEOVERRULE")> _
        Public Shared Sub MyCommand()
            ' Initialize Overrule if this is the first time this function has run
            If mDrawOverrule Is Nothing Then
                mDrawOverrule = New MyDrawOverrule
                Overrule.AddOverrule(RXObject.GetClass(GetType(Line)), mDrawOverrule,
False)
            End If

            'Toggle Overruling on/off
            Overrule.Overruling = Not Overrule.Overruling
            'Regen is required to update changes on screen.
            Application.DocumentManager.MdiActiveDocument.Editor.Regen()
        End Sub

    End Class
```

Let's go through this a few lines at a time, although the code is pretty much self-documenting:

```
Private Shared mDrawOverrule As MyDrawOverrule
```

This is just a class member variable we'll be using to store our single instance of the draw overrule. Note that this is a Shared (or static in C#) variable. This means the same global variable is associated with every instance of the class. (I.e. if you change it in one instance of myCommands, then it changes in every instance). Overrules are applied globally across all open drawings, so we only need one instance of our overrule.

Our command is contained in the `Public Shared Sub MyCommand()` subroutine. We tag the subroutine with the attribute

```
<Autodesk.AutoCAD.Runtime.CommandMethod("TOGGLEOVERRULE")>
```

to mark it as our custom command TOGGLEOVERRULE. Note that MyCommand() is Shared, which means AutoCAD doesn't have to instantiate our command class (myCommands) in every

new document we invoke the TOGGLEOVERRULE command. AutoCAD doesn't have to instantiate our class at all because Shared methods and variables are available at the class level, and don't require an instance of the class to call them.

In the next lines of code

```
            If mDrawOverrule Is Nothing Then
                mDrawOverrule = New MyDrawOverrule
                Overrule.AddOverrule(RXObject.GetClass(GetType(Line)), mDrawOverrule,
False)
            End If
```

we first check we've not already registered our one and only overrule instance. If we haven't (`mDrawOverrule Is Nothing`) then we create a new instance of `MyDrawOverrule` and register it by calling the `AddOverrule` function of the global `Overrule` instance. The first parameter is the class we're registering the overrule for (in this case a Line), the second parameter is our custom overrule instance, the third parameter specifies whether we want this overrule to be the last one called (if we have more than one overrule registered for this class)[10].

Now we've registered our overrule (if it wasn't already registered), we toggle the value of the global `Overrule.Overruling` property – so calling this command is like flicking a switch on and off.

```
            Overrule.Overruling = Not Overrule.Overruling
```

Finally, we have to regenerate our drawing to force AutoCAD to call WorldDraw on each Line in the drawing (so our custom graphics are drawn):

```
Application.DocumentManager.MdiActiveDocument.Editor.Regen()
```

Now put a breakpoint in the override of the WorldDraw function (select the line of code and hit F9); hit F5 to launch AutoCAD from your debugger; NETLOAD the compiled DLL; draw a line, and type 'TOGGLEOVERRULE' at the command line. We're not drawing any graphics yet, but your breakpoint should be called.

Now for the final hurdle – actually drawing some custom graphics from our overrule. I always think that lines are a bit too long and pointy, so we're going to make every line in our drawing draw itself as something much less pointy - a circle. Close AutoCAD and end your debugging session. Then modify the overridden WorldDraw function as follows:

```
        Public Overrides Function WorldDraw(ByVal drawable As
Autodesk.AutoCAD.GraphicsInterface.Drawable, ByVal wd As
Autodesk.AutoCAD.GraphicsInterface.WorldDraw) As Boolean
            'Cast Drawable to Line so we can access its methods and properties
            Dim thisLine As Line = drawable
            'Draw some graphics primitives
```

[10] If multiple applications try to register their overrule as the last to be called, then the last one registered will be the last one called.

```
            wd.Geometry.Circle(thisLine.StartPoint + 0.5 * thisLine.Delta,
thisLine.Length / 2, thisLine.Normal)
            'In this case we don't want the line to draw itself, nor do we want
ViewportDraw called
            Return True
            'Return MyBase.WorldDraw(drawable, wd)
        End Function
```

Let's walk through that code.

First we cast the Drawable passed to this function to a Line. The only way it wouldn't be a Line is if we changed our call to AddOverrule to register this Overrule for some class other than a Line, so we can safely assume it is a Line here.

```
Dim thisLine As Line = drawable
```

Then we use some parameters from the Line to draw a circle. To do this, we call the Circle method on the GraphicsInterface.Geometry class returned by the Geometry property of the GraphicsInterface.WorldDraw instance (`wd`):

```
wd.Geometry.Circle(thisLine.StartPoint + 0.5 * thisLine.Delta, thisLine.Length / 2,
thisLine.Normal)
```

The GraphicsInterface.Geometry class exposes all the graphics primitives that AutoCAD entities use to draw themselves (for example, circles, arcs, lines, meshes, shells, text). The parameters we're passing into the Circle method are its center, radius and normal, respectively.

Finally, we return True to tell AutoCAD that this entity doesn't implement a ViewportDraw function (return False if it does). If we'd wanted the line to draw its own graphics, then we'd have called the native line WorldDraw function and returned its return value instead of returning True. We've commented out the line we'd use to do that (`'Return MyBase.WorldDraw(drawable, wd)`).

Now it's time to run your project again. Remove your breakpoints, hit F5, NETLOAD your project into AutoCAD, draw a few lines and type TOGGLEOVERRULE at the command line. If you've implemented the above steps correctly, you should see all your lines turn into circles. (You can find the finished code for this in the MyFirstOverrule sample included with this document).

Congratulations on implementing your first Overrule! You now understand the basics of adding an overrule to a standard AutoCAD entity.

Available overrules

The Overrule API is broken up into several overrule classes according to functionality. Here is the full list of overrule classes and the entity properties and methods they overrule. The best reference for understanding the purpose of these different overrulable functions is the ObjectARX Developers and Reference Guides[11].

[11] A good starting point is the ObjectARX Developers Guide 'Deriving from AcDbObject' and 'Deriving from AcDbEntity' sections.

PropertiesOverrule

- GetClassID
- List

ObjectOverrule

- DeepClone
- WblockClone
- Open
- Close
- Cancel
- Erase

DrawableOverrule

- SetAttributes
- WorldDraw
- ViewportDraw
- ViewportDrawLogicalFlags

OsnapOverrule

- GetObjectSnapPoints

TransformOverrule

- TransformBy
- GetTransformedCopy
- Explode
- CloneMeForDragging
- HideMeForDragging

GripOverrule

- GetGripPoints
- MoveGripPointsAt
- GetStretchPoints
- MoveStretchPointsAt

OnGripStatusChanged

SubentityOverrule

- AddSubentPaths
- DeleteSubentPaths
- GetSubentPathsAtGsMarker
- GetGsMarkersAtSubentPaths
- GetGripPointsAtSubentPath
- MoveGripPointsAtSubentPaths
- SubentGripStatus
- SubentPtr
- TransformSubentPathsBy
- GetCompoundObjectTransform
- GetSubentPathGeomExtents
- GetSubentClassId

HighlightOverrule

- Highlight
- Unhighlight

GeometryOverrule

- GetGeomExtents
- IntersectWith

Selective overruling

The 'my first overrule' example we walked through above changed the behavior of every line in the drawing. It's more likely that we will want some lines in the drawing to represent something special, while the rest should be … well … just lines. There are five methods of the Overrule class that allow us to selectively overrule entities in a drawing:

- SetIdFilter – Passes an array of ObjectIds of those entities the overrule should be applied to.
- SetXdataFilter – The overrule is only applied to entities that have Xdata attached that belongs to the specified registered application id[12].
- SetExtensionDictionaryEntryFilter – The Overrule is only applied to entities that have an entry in their extension dictionary with the specified name[13].
- SetCustomFilter – Allows us to override your custom overrule's IsApplicable function to set our own custom filter criteria. For example, we might apply our overrule to all red circles with a radius of less than 5 units that are not on layer 0.
- SetNoFilter – Reverts the Overrule to apply to all entities of the type we registered it for in the drawing. As we've already seen, this is the default behavior.

The *ObjectIdFilterOverrule* sample demonstrates filtering using ObjectIds. This technique is not ideal if we want wish to persist our ObjectId list between sessions. We would have to store the list in the drawing (in the Named Objects Dictionary, for example), so we might as well flag our entities to be overruled using Xdata or extension dictionary entries instead.

The *EEDOverrules* sample demonstrates SetXdataFilter and SetExtensionDictionaryEntryFilter. These are the filter methods to use if we wish your filtered list to persist between sessions. And, as our overrules will often act on some instance specific data, we'll be storing data as Xdata or in an extension dictionary anyway.

The *CustomFilterOverrule* sample demonstrates using SetCustomFilter. In the example, we filter on a block insert's name.

Xrecords vs Xdata

The easy way to persist data used by your overrules is to add it to the overruled objects using either xdata or xrecords. There are advantages and disadvantages to both. Here are some of the key differences broken down by behaviors:

[12] You should prefix this name with your Registered Developer Symbol (http://www.autodesk.com/symbolreg).
[13] Ditto.

Jigging

A significant benefit of xdata over xrecords is that xdata is copied when an object is shallow-cloned, whereas the extension dictionary and its contents are not. AutoCAD makes a shallow clone of an entity when it's being dragged in a jig (e.g. in the MOVE or COPY command). This means that any custom graphics that depend on your xdata can be displayed easily when jigging your overruled entity, but the data to construct your graphics isn't available during jigging if you've put it in the extension dictionary – in which case you can't draw your graphics and the dragged entity will look like a normal non-overruled entity.

This means you should always try to store data used to construct your graphics and your grips as xdata. The *CustomLeader* sample demonstrates this; and also shows how to use the xdata world coordinate data type to let AutoCAD automatically transform your grip points when the entity you're overruling is being transformed.

Data storage capacity

A major limitation with using Xdata is that it is limited to 16KB per object. And that 16KB has to be shared with AutoCAD and any other addin applications that your user may have installed. Consequently, to be a good 'AutoCAD citizen', your application shouldn't store more than about 2KB of Xdata per object. Even then, you're stuck if another addin is being a naughty AutoCAD citizen.

Xrecords can store up to 2GB of data. It's unlikely that you'll need all that space, but it is possible you'll want more than 2KB. And, unlike xdata, you get the xrecord all to yourself so other applications can't be rude and steal all the storage space.

Another minor irritation is that when you retrieve xdata from an object you then have to iterate the ResultBuffer chain to find the data associated with your registered application ID. Xrecords help here because you can store your data in its own Dictionary, and even break up your data between multiple Xrecords. The *Network* example demonstrates storing three types of data in three separate Xrecords.

Inter-object pointers

Data is stored in both xrecords and xdata as DXF codes. Xrecords use DXF codes 1-369. These are the DXF codes used to define data within AutoCAD native objects. This means an xrecord can store certain data types that cannot be stored as xdata – in particular, soft pointer Ids, hard pointer Ids, soft ownership Ids and hard ownership Ids. The advantage of these types is that they are automatically translated when an object is deepcloned.

The best xdata can offer is storing handles using DXF code 1005. Depending on the context, handles stored in xdata are sometimes translated during cloning and sometimes not - which means you can't ever assume that they will be. Therefore, you have to take care of the translations yourself.

This means that inter-object relationships should be stored in xrecords wherever possible. The *Network* example demonstrates the use of inter-object pointers.

Drawing stuff

Changing how an entity displays itself is the most common use for Overrules. The 'my first overrule' sample showed how to draw custom graphics using calls to the GraphicsInterface.Geometry class. This provides very low level access to the graphics system, and is the most flexible way to create your custom graphics. However, we can also create temporary (non-database-resident) AutoCAD entities to draw our geometry for us. There are two ways to do this.

Let's take a look at all three techniques.

Raw Geometry

Here is a simple example of a WorldDraw overrule that uses the GraphicsInterface graphics primitives to draw a circle, arc and some text as additional graphics for a Line.

```
    'Draw using graphics primitives
    Public Overrides Function WorldDraw(ByVal drawable As
Autodesk.AutoCAD.GraphicsInterface.Drawable, ByVal wd As
Autodesk.AutoCAD.GraphicsInterface.WorldDraw) As Boolean
      Dim myLine As Line = drawable
      'Draw a circle
      wd.Geometry.Circle(myLine.StartPoint + 0.5 * myLine.Delta, myLine.Length / 4,
myLine.Normal)
      'Draw some text using the default textstyle
      Dim txtStyle As New TextStyle
      wd.Geometry.Text(myLine.StartPoint, myLine.Normal, myLine.Delta, "Graphics
primitives", True, txtStyle)
      'Draw an arc
      wd.Geometry.CircularArc(myLine.EndPoint, myLine.Length / 4, myLine.Normal,
myLine.Normal.CrossProduct(myLine.Delta.GetNormal), Math.PI, ArcType.ArcSimple)
      Return MyBase.WorldDraw(drawable, wd)
    End Function
```

A full discussion of the GraphicsInterface namespace is beyond the scope of this document. The best reference for how to use the GraphicsInterface classes are ObjectARX custom entity samples and the ObjectARX helpfiles[14], as this is generally how we would draw our graphics if we were developing a custom entity. In ObjectARX, the GraphicsInterface classes are prefixed by AcGi.

Draw

An alternative approach to using raw geometry is to use temporary entities to draw our graphics. This approach is more intuitive if you're not familiar with the GraphicsInterface namespace. We

[14] See the ObjectARX Developers Guide section on 'The Graphics Interface Library'.

can change the characteristics (color, linetype, etc.) of the temporary objects using their standard properties (e.g. myCircle.ColorIndex).

You do have to be a little careful when using the Geometry.Draw method because the graphics system expects the drawables (your temporary entities) to remain in memory for as long as the graphics are displayed. If you Dispose of your temporary entities too early, then you can crash AutoCAD. The 3D graphics pipeline is particularly sensitive to this.

This example demonstrates the power of combining temporary entities that remain unchanged for the lifetime of the Overrule with the Push/PopTransform method to transform the graphics interface coordinate system:

- We create the temporary entities in fixed locations in a coordinate system we define (let's call that the Entity Coordinate System, the ECS).
- Then we transform between the ECS and the World Coordinate System (WCS) using the PushTransform method.
- Then we call PopTransform to restore the coordinate system to the state in which it was at the start of our WorldDraw function.

Geometry.Draw and Push/PopTransform is used by BlockReferences to display the BlockTableRecord contents at the correct location, scale and orientation. Think about how that would work for blocks or Xrefs nested several layers deep.

```
    'Create our temporary entities once and keep them for the lifetime of the overrule
    Public Sub New()
      mText = New DBText
      mText.TextString = "Temporary entities (Draw)"
      mText.Position = New Point3d(0, 0, 0)
      mCircle = New Circle(New Point3d(0.5, 0, 0), New Vector3d(0, 0, 1), 0.25)
      mArc = New Arc(New Point3d(1, 0, 0), New Vector3d(0, 0, 1), 0.25, Math.PI / 2, -
Math.PI / 2)
    End Sub

    'Draw using temporary entities - calling Geometry.Draw.
    Public Overrides Function WorldDraw(ByVal drawable As
Autodesk.AutoCAD.GraphicsInterface.Drawable, ByVal wd As
Autodesk.AutoCAD.GraphicsInterface.WorldDraw) As Boolean
      ' Calculate matrix to transform from WCS to Line's OCS
      Dim myLine As Line = drawable
      Dim OCS_OriginPt As Point3d = myLine.StartPoint
      Dim OCS_XVec As Vector3d = myLine.Delta.GetNormal
      Dim OCS_ZVec As Vector3d = myLine.Normal
      Dim OCS_YVec As Vector3d = OCS_ZVec.CrossProduct(OCS_XVec)
      ' We don't want text to scale with the line length, so we calculate an
'unscaled' OCS for the text
      Dim unscaledOCS_Mat As Matrix3d = Matrix3d.AlignCoordinateSystem(Point3d.Origin,
Vector3d.XAxis, Vector3d.YAxis, Vector3d.ZAxis, OCS_OriginPt, OCS_XVec, OCS_YVec,
OCS_ZVec)
      ' We do want the circle and arc to scale with the line, so we calculate a 'full'
(corrctly scaled) OCS
      Dim OCS_Mat As Matrix3d = Matrix3d.Scaling(myLine.Length, myLine.StartPoint) *
unscaledOCS_Mat
```

```
      ' Push our scaled transform
      wd.Geometry.PushModelTransform(OCS_Mat)
      ' Draw the arc and circle
      wd.Geometry.Draw(mCircle)
      wd.Geometry.Draw(mArc)
      ' Remove our scaled transform
      wd.Geometry.PopModelTransform()
      ' push our unscaled transform
      wd.Geometry.PushModelTransform(unscaledOCS_Mat)
      ' Draw the text
      wd.Geometry.Draw(mText)
      wd.Geometry.PopModelTransform()
      ' Tell line to draw itself
      Return MyBase.WorldDraw(drawable, wd)
    End Function

    'And remember to Dispose of mText, mArc and mCircle before Disposing the overrule
class.
```

WorldDraw

Finally, there is one other way to use temporary entities. In this case, we're creating the entity, transforming it to the correct coordinate system, and then Disposing it. Calling WorldDraw on one of these temporary entities is just like calling any other helper function from your WorldDraw overrule. This is simpler than using Geometry.Draw because we don't have to worry about the lifetime of the temporary entities.

```
    ' Draw using temporary entities - calling WorldDraw
    ' Not using Push/PopTransform to demonstrate transforming entity instead of
transforming GI coordinate system. I could have used them if I wished.
    Public Overrides Function WorldDraw(ByVal drawable As
Autodesk.AutoCAD.GraphicsInterface.Drawable, ByVal wd As
Autodesk.AutoCAD.GraphicsInterface.WorldDraw) As Boolean
      ' Calculate matrix to transform from WCS to Line's OCS
      Dim myLine As Line = drawable
      Dim OCS_OriginPt As Point3d = myLine.StartPoint
      Dim OCS_XVec As Vector3d = myLine.Delta.GetNormal
      Dim OCS_ZVec As Vector3d = myLine.Normal
      Dim OCS_YVec As Vector3d = OCS_ZVec.CrossProduct(OCS_XVec)
      ' We don't want text to scale with the line length, so we calculate an
'unscaled' OCS for the text
      Dim unscaledOCS_Mat As Matrix3d = Matrix3d.AlignCoordinateSystem(Point3d.Origin,
Vector3d.XAxis, Vector3d.YAxis, Vector3d.ZAxis, OCS_OriginPt, OCS_XVec, OCS_YVec,
OCS_ZVec)
      ' We do want the circle and arc to scale with the line, so we calculate a 'full'
(corrctly scaled) OCS
      Dim OCS_Mat As Matrix3d = Matrix3d.Scaling(myLine.Length, myLine.StartPoint) *
unscaledOCS_Mat
      ' Draw the arc and circle
      Using circ As Circle = New Circle(New Point3d(0.5, 0, 0), New Vector3d(0, 0, 1),
0.25)
        circ.TransformBy(OCS_Mat)
        circ.WorldDraw(wd)
      End Using
      Using arc As Arc = New Arc(New Point3d(1, 0, 0), New Vector3d(0, 0, 1), 0.25,
Math.PI / 2, -Math.PI / 2)
        arc.TransformBy(OCS_Mat)
```

```
        arc.WorldDraw(wd)
      End Using
      ' Draw the text
      Using txt As DBText = New DBText
        txt.TextString = "Temporary entities (WorldDraw)"
        txt.Position = New Point3d(0, 0, 0)
        txt.TransformBy(unscaledOCS_Mat)
        txt.WorldDraw(wd)
      End Using
      ' Tell line to draw itself
      Return MyBase.WorldDraw(drawable, wd)
    End Function
```

The two temporary graphics approaches are particularly useful if you're also overruling Explode and IntersectWith. You can use the same temporary entities to calculate your exploded entity set and intersection points as you use for drawing your graphics.

All three of these functions draw the same graphics (except for the contents of the text). Which you choose is a matter of personal preference.

The *DrawingStuff*[15] sample demonstrates all three approaches.

Deep Cloning

If a DBObject depends on some other DBObjects in the DWG database, then it has to make sure those other objects are correctly copied when it is copied. For example, consider an overrule that uses inter-object pointers to link a Line to a Circle. What behavior do you want when your user copies that line?

- Do you want the new Line to be linked to the same Circle?
- Do you want to also copy the Circle and link the new Line to the new Circle?
- Do you want the new Line to not be linked to anything at all?

The ObjectOverrule DeepClone WblockClone functions allow you to customize how your overruled object is copied. The ObjectARX Developer's Guide 'Deep Cloning' chapter explains the complexities of deep cloning in great detail (and it can get very complex), but the simple rule is that:

- DeepClone is used for copying within the same DWG database.
- WblockClone is for copying between different databases.

For the most common case of linking one entity to another (the Line and the Circle I just described), you will maintain the link between the two entities in your own data (I normally store these link as SoftPointerIds in an Xrecord). You can then use your custom data to add the linked entity to the deep cloning operation if needed.

[15] Note: The DrawingStuff sample has been updated from the version posted with the AU 2009 presentation.

The *Network* sample included with this handout demonstrates adding entities to DeepClone and WblockClone operations. All you have to do is relay the DeepClone/WblockClone call to the entities you want to include in the deep cloning operation.

Deep cloning is something of a black art[16]. You can get some strange behaviors at times, especially when you add in the complexity of using overrules and potentially circular dependencies. It's because of that particular case that I'm including this short section on Deep Cloning …

When studying the Network sample along with the ObjectARX Developer's Guide, it would appear that all that should be required to include the linked entities in the cloning operation is to use HardPointerIds in your Xrecord to hold those links. (HardPointers are supposed to be automatically included in WblockClone operations). But it confuses the cloning operation if you try to clone a circular network. (Try it and see!) This is why I'm using SoftPointerIds and then adding the linked entities to the cloning operation myself. If you know you'll have no circular dependencies in your inter-object links, then you'll probably be ok using HardPointerIds.

And another tip – When using Xrecords in an extension dictionary to hold your Hard/SoftPointerIds, make sure you set DBDictionary.TreatElementsAsHard to True for your whole dictionary tree. This ensures the dictionaries and their contents are included in the WblockClone operation. (The Hard/SoftPointerIds are automatically translated to point to the clones of the originals or set to null if the entity wasn't cloned).

Transactions

You've probably been writing code like this in most of your .NET addins:

```
      Dim db As Database = Application.DocumentManager.MdiActiveDocument.Database
      ' Start the transaction
      Using trans As Transaction = db.TransactionManager.StartTransaction
        ' Open current space for read
        Dim btr As BlockTableRecord = trans.GetObject(db.CurrentSpaceId,
OpenMode.ForRead)
        ' Iterate through all entities in current space
        For Each objId As ObjectId In btr
          ' Open entity for write
          Dim ent As Entity = trans.GetObject(objId, OpenMode.ForWrite)
          ' ...
          ' ...
          ' ...
        Next
        trans.Commit()
      End Using
```

That's not such a good idea when using Overrules. Transactions interact with AutoCAD's Undo filer, and that can cause problems (for example) when you use Transactions inside your

[16] Watch Cyrille Fauvel's AU Virtual 2010 Product Clinic - CP220-3C-Advanced Deepclone API in AutoCAD® -for an in-depth explanation of deepcloning in AutoCAD

WorldDraw overrule. There is an alternative to using Transactions in .NET – using the Open/Close mechanism. There are actually two ways of using Open/Close: directly or via the OpenCloseTransaction type.

Here is equivalent code to the Transaction example shown above using the Open/Close[17] model:

```
      Dim db As Database = Application.DocumentManager.MdiActiveDocument.Database
      ' Open current space for read
      Dim btr As BlockTableRecord = db.CurrentSpaceId.Open(OpenMode.ForRead)
      ' Iterate through all entities in current space
      For Each objId As ObjectId In btr
        ' Open entity for write (will fail if entity is aleady open)
        Dim ent As Entity = objId.Open(OpenMode.ForWrite)
        ' ...
        ' ...
        ' ...
        ' Close the entity
        ent.Close()
      Next
      ' Close the BlockTableRecord
      btr.Close()
```

The biggest problem with using Open/Close directly is that you are responsible for closing every object you open. If you forget to close one then AutoCAD will crash – and probably not straight away, so it's really hard to work out where your bug is. Transactions are great because they take care of all that for you.

Now here is the same code again using an OpenCloseTransaction. See how similar it is to the normal transaction usage:

```
      Dim db As Database = Application.DocumentManager.MdiActiveDocument.Database
      ' Start the transaction
      Using trans As OpenCloseTransaction =
db.TransactionManager.StartOpenCloseTransaction
        ' Open current space for read
        Dim btr As BlockTableRecord = trans.GetObject(db.CurrentSpaceId,
OpenMode.ForRead)
        ' Iterate through all entities in current space
        For Each objId As ObjectId In btr
          ' Open entity for write (will fail if entity is aleady open)
          Dim ent As Entity = trans.GetObject(objId, OpenMode.ForWrite)
          ' ...
          ' ...
          ' ...
        Next
        trans.Commit()
      End Using
```

The OpenCloseTransaction is a convenient wrapper for the Open/Close model and allows you to write your code as you would using normal Transactions. The direct Open/Close close

[17] Don't worry about the compiler warning you get when using Open/Close. They are just politely suggesting that you may prefer to use Transactions (or OpenCloseTransactions) instead.

approach gives you more flexibility than OpenCloseTransaction. For example, it allows you to decide exactly when each object is actually closed (if you're into that kind of thing). However, it makes error handling a lot more troublesome. Look at the differences in the following two code snippets.

Using OpenCloseTransactions:

```
      Try
        Dim db As Database = Application.DocumentManager.MdiActiveDocument.Database
        Using trans As OpenCloseTransaction =
db.TransactionManager.StartOpenCloseTransaction
          ' Open current space for read
          Dim btr As BlockTableRecord = trans.GetObject(db.CurrentSpaceId,
OpenMode.ForWrite)
          ' Add new circle to current space
          Using aCircle As New Circle
            aCircle.Radius = 5
            btr.AppendEntity(aCircle)
            trans.AddNewlyCreatedDBObject(aCircle, True)
          End Using
          ' Add new line to current space
          Using aLine As New Line
            aLine.EndPoint = New Point3d(100, 100, 0)
            btr.AppendEntity(aLine)
            trans.AddNewlyCreatedDBObject(aLine, True)
          End Using
          trans.Commit()
        End Using
      Catch ex As Autodesk.AutoCAD.Runtime.Exception
        ' ...
      Finally
        ' ...
      End Try
```

Using Open/Close – note the extra (ugly) code in the Finally block:

```
      Dim btr As BlockTableRecord
      Dim aCircle As Circle
      Dim aLine As Line
      Try
        Dim db As Database = Application.DocumentManager.MdiActiveDocument.Database
        ' Open current space for read
        btr = db.CurrentSpaceId.Open(OpenMode.ForWrite)
        ' Add new circle to current space
        aCircle = New Circle
        aCircle.Radius = 5
        btr.AppendEntity(aCircle)
        ' Add new line to current space
        aLine = New Line
        aLine.EndPoint = New Point3d(100, 100, 0)
        btr.AppendEntity(aLine)
      Catch ex As Autodesk.AutoCAD.Runtime.Exception
        ' ...
      Finally
        ' We need to close any open objects whether or not an exception was thrown
        ' But if an exception was thrown then we don't know if they were all created,
```

```
        '  so we have to check each one
        If Not btr Is Nothing Then
          btr.Close()
        End If
        If Not aCircle Is Nothing Then
          aCircle.Close()
        End If
        If Not aLine Is Nothing Then
          aLine.Close()
        End If
      End Try
```

The main difference between using a normal Transaction and an OpenCloseTransaction or Open/Close is that, when not using a normal Transaction, opening an object can fail if that object is already open. The table below lists the different object opening scenarios:

Object is …	**Attempt to open it for …**	**Transaction will …**	**Open/Close and OpenCloseTransaction will …**
Not open	Read	Succeed	Succeed
Not open	Write	Succeed	Succeed
Open for read	Read	Succeed	Succeed
Open for read	Write	Succeed	Fail
Open for write	Read	Succeed	Fail
Open for write	Write	Succeed	Fail

I recommend you use Open/Close or OpenCloseTransactions exclusively in your Overrules. I demonstrate both OpenCloseTransaction and Open/Close in the samples.

Samples demonstrated in the presentation

The samples demonstrated in the presentation are available for download from the AU Online site with this handout. The readme accompanying each sample provide full usage instructions. Here is a brief description of each, listing the overrules used in each:

CustomLeader

Adds a simple leader line to a BlockReference. Differs from the AutoCAD multileader because it allows the position of the startpoint and endpoint of the leader line to be specified. Stores start and end points for leader line as Xdata to ensure automatic transformation of points when the

block reference is moved. Use of Xdata also allows display of overruled graphics during jigging (e.g. during copy and paste).

Overrules used:

DrawableOverrule: WorldDraw.

Grip Overrule: GetGripPoints, MoveGripPointsAt.

OsnapOverrule: GetObjectSnapPoints.

GeometryOverrule: GetGeomExtents.

TransformOverrule: Explode.

Network

Simulates information flow through a user-created network of nodes. Demonstrates updating graphics of linked entities when the primary entity is moved or erased, and adding linked entities to cloning operations.

Overrules used:

DrawableOverrule: WorldDraw.

ObjectOverrule: Erase, Close, WblockClone, DeepClone.

DigSigStamp

Overrules a BlockReference to display information from a digital certificate if the DWG that contains it has a valid digital signature. Demonstrates simple use of SetAttributes return value to turn off caching by 3D graphics pipeline.

Overrules used:

DrawableOverrule: WorldDraw, SetAttributes.

DimensionPatrol

Highlights Dimensions in a drawing that have had their dimension text overwritten by custom user text. Demonstrates use of IsApplicable function to define a custom Overrule filter.

Overrules used:

DrawableOverrule: WorldDraw, IsApplicable.

DigSignStamp and DimensionPatrol have both been published on the Autodesk Labs Plugin of the Month website - http://labs.autodesk.com/utilities/ADN_plugins/.

Call to action

Overrules are a powerful API first introduced in AutoCAD 2010. Because adding new overrules will break existing ObjectARX custom entities, we can't add any significant new Overrule APIs until the next break in binary compatibility – which is likely to be with AutoCAD 2013[18].

You now know the basics of the Overrule API, so go ahead and experiment with your own. There are so many things you can do with overrules already, but please do let us know what additional entity behavior you'd like to be able to overrule when we can next add to this API. (And tell us about any bugs you find – we can fix those before we break binary compatibility).

Autodesk Developer Network (ADN) members can submit 'Wishlist' requests and bug reports through DevHelp Online on the members-only ADN website. If you're not an ADN member, then you're welcome to email me – stephen.preston@autodesk.com – with your overrule requests and bug reports.

Good luck with your overrule programming!

[18] Assuming the current release naming convention continues, along with our policy to break application compatibility only every three releases.

Further reading

- A good starting point for all .NET developers is the resources listed on the AutoCAD Developer Center – www.autodesk.com/developautocad. These include:
    - Training material, recorded presentations, and our AutoCAD .NET Wizards.
    - The AutoCAD .NET Developers Guide - http://www.autodesk.com/autocad-net-developers-guide.
    - Information on joining the Autodesk Developer Network – www.autodesk.com/joinadn
    - Information on training classes and webcasts – www.autodesk.com/apitraining
    - Links to the Autodesk discussion groups – www.autodesk.com/discussion. (Click on the AutoCAD link from there to access the AutoCAD API discussion groups).
- Many of the concepts needed to understand overrules are included in the ObjectARX SDK documentation. The general descriptions in the C++ documentation are still extremely useful to .NET developers, including:
    - Using the GraphicsInterface API – See the ObjectARX Developers Guide section on 'The Graphics Interface Library'
    - The purpose of the entity functions you're overruling – See the ObjectARX Developers Guide 'Deriving from AcDbObject' and 'Deriving from AcDbEntity' sections.
    - The ObjectARX and Managed Reference Guides for function by function descriptions.
    - Overrules from a C++ perspective – See the ObjectARX Developers Guide 'Behavior overrules' section.
- Kean Walmsley's .NET focused blog includes several Overrule API usage examples - http://blogs.autodesk.com/through-the-interface.
- If you're an ADN partner, there is a wealth of Autodesk API information on the members-only ADN website – http://adn.autodesk.com.
    - And ADN members can ask unlimited API questions through our DevHelp Online interface
- Watch out for our regular ADN DevLab events. DevLab is a programmers' workshop (free to ADN and non-ADN members) where you can come and discuss your AutoCAD programming problems with the ADN DevTech team. They are advertised on our API training schedule accessible from www.autodesk.com/apitraining.