MA5840

This class discusses the concepts and details you need to know to access the Autodesk Inventor assembly functionality through the Inventor Application Programming Interface (API). Some topics covered are: the assembly occurrences, assembly structure, assembly constraints. Previous experience working with the Inventor API is recommended.

Learning Objectives

At the end of this class, you will be able to:

- Understand Inventor’s assembly structure and file references.
- Create assemblies automatically using Inventor’s API.
- Understand how to use constraints and assemblies through Inventor’s API.

About the Speaker

Brian is the designer of the Autodesk Inventor programming interface. He began working in the CAD industry over 25 years ago in various positions, including CAD administrator, applications engineer, CAD API designer, and consultant. Brian is the original designer of the Inventor and Solid Edge API’s.

brian.ekins@autodesk.com
This is a brief introduction to Inventor's assembly functionality, as accessed through the API. Most of the time is spent focusing on the most commonly used assembly functionality; both querying and creating the assembly structure with a brief overview of some other assembly functionality.

**Assembly Structure**

The basic concept of assembly structure is simple. You have parts at the lowest level. You use assemblies to combine parts and other assemblies. When an assembly is used within another assembly it is referred to as a subassembly. You can create an assembly that contains any number of subassemblies and parts, with each of those subassemblies containing any number of subassemblies and parts, with no limit to the number of levels. In the user-interface, this assembly structure is graphically represented as a tree in the browser.

Below is a simple assembly of a toy car. You can see by looking at the browser that the top-level assembly (Car.iam) contains the body part (Body.ipt) and two subassemblies (WheelAssembly.iam). Each subassembly contains two wheel parts (Wheel.ipt) and one axle part (Axle.ipt).
You’re already familiar with assemblies from a user-interface point of view. Here’s a look at the API point of view, which is shown below, on the right. At the top of the hierarchy is the `AssemblyDocument` object. This object represents any .iam file and primarily provides general document functionality (iProperties, saving, file references, etc). From the `AssemblyDocument` you get the `AssemblyComponentDefinition` object. This object provides access to all of the assembly specific functionality. To access the assembly structure you use the `ComponentOccurrences` collection object. This collection returns all of the parts and subassemblies in the current level of the assembly. It also supports methods to add new parts or assemblies to the assembly. In the example below, the `ComponentOccurrences` collection for `Car.iam` will return `Body:1`, `WheelAssembly:1`, and `WheelAssembly2`. The `ComponentOccurrence` collection for `WheelAssembly.iam` will return `Axle:1`, `Wheel:1`, and `Wheel:2`. The next API object is the `ComponentOccurrence` object. The `ComponentOccurrence` object represents a single instance of a part or subassembly within an assembly. In this example, `Body:1`, `WheelAssembly:1`, and `WheelAssembly2` are all `ComponentOccurrence` objects.

From a `ComponentOccurrence` object, also referred to simply as an occurrence, you can get a lot of interesting information. Here are some of the most important:

- The type of document (part of assembly) the occurrence represents.
- For an occurrence representing an assembly, you can get its sub occurrences.
- For an occurrence representing a part, you can get the geometry of the part.
- You can get the position of the occurrence in the assembly.
- You can get the occurrence properties; visible, suppressed, color, etc.
- You can get the part or assembly document that is being referenced by that occurrence.
Assembly Traversal
A common programming task is to traverse through an entire assembly structure. The following sample code does that. It uses a programming technique referred to as recursion. A recursive function is one that calls itself. Remember that an assembly can contain any number of occurrences and any number of levels. You can’t know ahead of time what to plan for. With a recursive function it doesn’t matter because it will continue to traverse the entire assembly until no more occurrences are found.

The recursive sub here is TraverseAsm. The sub AssemblyTraversal is the starting point that gets everything going. AssemblyTraversal calls TraversAsm and passes in the ComponentOccurrences collection of the top-level assembly. The second argument isn’t really needed but is just used to track what level of the assembly it’s currently at so it can format the output.

TraverseAsm iterates through all of the occurrences in the ComponentOccurrences collection that is passed in. For each occurrence, it prints out the name. The Space function that’s used in the Print statement adds some spaces based on the current level of the assembly so that the result is indented. Next, it checks to see if the current occurrence is a part or an assembly. If it’s an assembly, then is calls TraverseAsm again and passes in the collection of occurrences for the current occurrence. It also increases the level value by one since it’s dropping one more level down in the assembly. This continues until it runs out of subassemblies.

```
Public Sub AssemblyTraversal()
  ' Get the active document, assuming it's an assembly.
  Dim oAsmDoc As AssemblyDocument
  Set oAsmDoc = ThisApplication.ActiveDocument

  ' Begin the assembly traversal.
  Call TraverseAsm(oAsmDoc.ComponentDefinition.Occurrences, 1)
End Sub

' The Level argument is used to control the amount of indent for the output.
Private Sub TraverseAsm(oOccurrences As ComponentOccurrences, Level As Integer)
  ' Iterate through the current list of occurrences.
  Dim oOcc As ComponentOccurrence
  For Each oOcc In oOccurrences
    ' Print the name of the current occurrence.
    Debug.Print Space(Level * 3) & oOcc.Name

    ' If the current occurrence is a subassembly then call this sub
    ' again passing in the collection for the current occurrence.
    If oOcc.DefinitionDocumentType = kAssemblyDocumentObject Then
      Call TraverseAsm(oOcc.SubOccurrences, Level + 1)
    End If
  Next
End Sub
```

This example doesn’t do much besides traverse the assembly structure. Most programs will do more besides printing out the name of the current occurrence.

Other Ways to Access Occurrences
Traversing the assembly, as demonstrated above gets the assembly structure and every occurrence in the assembly. Sometimes you don’t need the structure or just need a subset of occurrences. The API provides some shortcut methods that will make it easier in these case.
The methods, along with a brief description are:

ComponentOccurrences.AllLeafOccurrences( [LeafDefinition As Variant] ) – This method finds all of the parts in the assembly, regardless of the level. Parts are often referred to as the leaves of the assembly tree structure. Optionally, you can specify a specific document and then only the occurrences that represent that document will be returned. The LeafDefinition argument must be a PartComponentDefinition.

ComponentOccurrences.AllReferencedOccurrences( Object As Variant ) – This method returns all occurrences (parts or assemblies) that represent the specified object. The required input object defines the part or assembly that you want to find the associated occurrences for. It can be a PartDocument, AssemblyDocument, PartComponentDefinition, AssemblyComponentDefinition, or DocumentDescriptor object.

Building an Assembly

Now that we’ve looked at the API representation of an assembly and how to query the structure and occurrences, let’s look at how you use the API to build an assembly and define the structure. It’s all done using the various Add methods of the ComponentOccurrences object. The most commonly used method is the Add method, which is used in the code below. This example, creates a new assembly document using the default assembly template and places an occurrence into that assembly.

```
Public Sub CreateAssembly()
    ' Create a new assembly document, using the default assembly template.
    Dim asmDoc As AssemblyDocument
    Set asmDoc = ThisApplication.Documents.Add(kAssemblyDocumentObject, 
        ThisApplication.FileManager.GetTemplateFile(kAssemblyDocumentObject))

    ' Get the ComponentOccurrences collection.
    Dim occurrences As ComponentOccurrences
    Set occurrences = asmDoc.ComponentDefinition.occurrences

    ' Place an occurrence into the assembly.
    Dim occ As ComponentOccurrence
    Set occ = occurrences.Add("C:\Temp\Part1.ipt", 
        ThisApplication.TransientGeometry.CreateMatrix)
End Sub
```

The Add method takes two arguments. The first argument is the full filename to a part or assembly. The second argument defines the position and orientation of the part within the assembly. It does this using a Matrix object. This is easily the most complicated thing about creating an occurrence and warrants spending some time to discuss.
Matrix Objects
It's general practice in the world of computer graphics to use a matrix to define a position and orientation in 3D space. The problem with this is it's a bit strange for those of us that didn't just finish a linear algebra class. Luckily, it's really not all that difficult to use and if you do get into some more advanced creation of assemblies, using a matrix makes doing many things easier. The definition of a matrix is, “a rectangular array of numbers”. In most computer graphics applications, a 3D position and orientation is defined using a 4x4 rectangular array, (four columns and four rows), like that shown to the right. The Inventor API supports the Matrix object, which is an API object that wraps a 4x4 rectangular array.

The following describes an approach that has helped me to understand how a matrix defines a position and orientation and hopefully will also be useful for you. When working with Inventor, if you need to define a position and orientation in space you probably think of using a user defined coordinate system. To define a coordinate system you need to define a point in space to specify the origin and you need to define the direction of the X, Y, and Z axes. A matrix defines these same things; an origin and axis directions.

The figure to the right illustrates how a matrix contains this information. The first column defines the direction of the x-axis direction (1,0,0). The second column defines the direction of the y-axis (0,1,0), and the third column defines the direction of the z-axis (0,0,1). Each column defines a vector whose X,Y,Z components are the values in the column. The last column defines the X,Y,Z coordinates of the coordinate system’s origin (0,0,0). You can ignore the bottom row.

A vector defines a direction and magnitude. For example, the vector (10,5,0) defines moving 10 units in the X direction, 5 units in the Y direction, and 0 units in the Z direction, for a total distance (or magnitude) of 11.18 units, as shown to the left. An important thing to understand about vectors is that they don’t define a position, only a movement. For example, you can apply this vector to any object, regardless of its current position and it will result in moving the object 10 units in the x direction and 5 units in the Y direction from whatever its current position is.

There’s also a special type of vector known as a unit vector whose length is always 1. Unit vectors are used when only the direction is needed. You can see that the X, Y, and Z vectors in the example matrix above are all unit vectors. For defining a coordinate system they always need to be unit vectors.

The matrix shown above is also a special type of matrix known as an identity matrix. What makes it special is that the origin is at (0,0,0) and the X axis in pointed in the X direction, the Y axis in the Y direction, and the Z axis in the Z direction. It defines a coordinate system that is exactly identical with the model coordinate system.
Now, let’s look at an example of how to define a general coordinate system using a matrix. In this example I want to define a coordinate system whose position is at (10,5,0) and is rotated 45° around the Z axis, like the coordinate system shown below. How do I create a matrix that defines that coordinate system?

The easiest part to define is the origin, which in this case is (10,5,0). The matrix below defines that by changing the values in the last column to the coordinates of the origin.

\[
\begin{bmatrix}
1 & 0 & 0 & 10 \\
0 & 1 & 0 & 5 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

The harder part now is to define the orientation. Looking at the picture of the desired coordinate system above you can see that the X and Y axes are rotated 45° around the Z axis. The Z axis still points in the Z direction so it doesn’t require any change. A vector of (1,1,0) correctly defines the direction of the desired X axis, and a vector of (-1,1,0) defines the Y axis. However, there is still a problem. A vector of (1,1,0) has a magnitude of 1.414. Remember that all vectors in a matrix defining a coordinate system need to be unit vectors, or have a magnitude of 1. Using a bit of trig we end up with the correct matrix below that correctly defines the desired coordinate system.

\[
\begin{bmatrix}
0.707 & -0.707 & 0 & 10 \\
0.707 & 0.707 & 0 & 5 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]
For a rectangular coordinate system, which is what Inventor always uses, there are certain rules that are enforced. These are that the x-axis and y-axis are perpendicular to each other and the z-axis is perpendicular to both the x-axis and y-axis and follows the right-hand rule. (Another way to determine the direction of the z-axis is that if the x-axis points to the right and the y-axis points up, then the z-axis will point toward you.) Each of the three axis vectors must also have a length of 1.

Let's look at how to apply this information to define a specific coordinate system. What would the matrix look like if we want to define a coordinate system whose position is at (10,5,0) and is rotated 45° around the z-axis?

```vbnet
Public Sub CreateAssembly2()
    ' Create a new assembly document, using the default assembly template.
    Dim asmDoc As AssemblyDocument
    Set asmDoc = ThisApplication.Documents.Add(kAssemblyDocumentObject, _
        ThisApplication FileManager.GetTemplateFile(kAssemblyDocumentObject))

    ' Get the ComponentOccurrences collection.
    Dim occurrences As ComponentOccurrences
    Set occurrences = asmDoc.ComponentDefinition.occurrences

    ' Get a reference to the transient geometry object.
    Dim tg As TransientGeometry
    Set tg = ThisApplication.TransientGeometry

    ' Create a matrix.
    Dim coordSys As Matrix
    Set coordSys = tg.CreateMatrix

    ' Define the origin.
    Dim origin As Point
    Set origin = tg.CreatePoint(10, 5, 0)

    ' Define the axes to get a 45 deg Rotation about the Z axis.
    Dim pi As Double
    pi = 3.14159265358979
    Dim xAxis As UnitVector
    Dim yAxis As UnitVector
    Dim zAxis As UnitVector
    Set xAxis = tg.CreateUnitVector(Cos(pi / 4), Sin(pi / 4), 0)
    Set yAxis = tg.CreateUnitVector(-Cos(pi / 4), Sin(pi / 4), 0)
    Set zAxis = tg.CreateUnitVector(0, 0, 1)

    ' Redefine the matrix using the new origin and axes.
    Call coordSys.SetCoordinateSystem(origin, xAxis.AsVector, _
        yAxis.AsVector, zAxis.AsVector)

    ' Place an occurrence into the assembly.
    Dim occ As ComponentOccurrence
    Set occ = occurrences.Add("C:\Temp\Part1.ipt", coordSys)
End Sub
```
As we saw earlier, Inventor has a Matrix object. It also has Vector and UnitVector objects. These objects make it much easier when working with this type of data because they support methods and properties that make many operations easier and save you writing a lot of, sometimes difficult, code to do the same things. You create Matrix, Vector, and UnitVector objects using methods on the TransientGeometry object. In the example above, a point object is created to define the origin and three unit vectors are created for the axes.

To define the matrix using these values, this example uses the SetCoordinateSystem method which takes the origin point and the axis vectors as input and sets the values in the matrix. One thing you might notice that’s a bit strange in the SetCoordinateSystem call is that the vector arguments are using the AsVector property of the UnitVector object. What the AsVector property does is return a Vector object that has the same values as the original UnitVector. (The Vector object also supports the AsUnitVector property.) The reason it is used here is that the SetCoordinateSystem expects Vector objects for the axis vectors and not UnitVector objects. It will allow you to create a coordinate system that is invalid when you’re working with assemblies. It’s best to use UnitVector objects when creating the vectors so that you will always create valid matrices. If your matrix isn’t valid the Add method will fail. Here are some other useful things when working with matrices.

**Straightening a Coordinate System**

Besides the requirement that the axis vectors are unit vectors, there’s also the requirement that they all be 90° to each other. For example, it’s fairly obvious the coordinate system on the left is not valid, but how do you fix it to be the coordinate system on the right? You can take advantage of a behavior of vectors called *crossing* or the *cross product*.

To do this you need two vectors. The only requirement is that they can’t both point in exactly the same direction. The pictures below illustrate using the right hand rule technique to understand what result to expect. The first picture shows the two vectors. Align your hand so your fingers point along one of the vectors. Curl your fingers in the direction of the second vector as shown in the middle picture. Your thumb points in the direction of the new vector which is 90° to the original two vectors. This example crosses the red vector into the green one. If you cross the green vector into the red one the new vector will be in the opposite direction.
Assuming the red and blue vectors are correct, the green vector can be fixed by crossing the blue vector into the red, which will result in the new green vector.

![Image of vectors](image)

An example of where this can be used is a case where you want to place a part relative to a circular edge on another part, like inserting a bolt into a hole. From the circular edge you can get the center and you can get the normal from the connected planar face. This gives us the origin and the Z axis for the coordinate system but we still need the X and Y axes. Assuming we’re placing something like a bolt where the rotation around the Z axis doesn’t matter, then the following function will work to create the matrix.

```vba
Public Function CreateMatrixFromZAxis(origin As point, zAxis As UnitVector) As Matrix
    Dim tg As TransientGeometry
    Set tg = ThisApplication.TransientGeometry

    ' Create a copy of the Z axis, changing the direction slightly
    ' so it's not exactly the same as the Z axis.
    Dim tempVec As UnitVector
    Set tempVec = tg.CreateUnitVector(zAxis.X + 0.1, zAxis.Y + 0.1, zAxis.Z + 0.1)

    ' Cross the Z axis into this new vector to create the new X axis.
    Dim xAxis As UnitVector
    Set xAxis = zAxis.CrossProduct(tempVec)

    ' Cross the Z axis into the X axis to get the Y axis.
    Dim yAxis As UnitVector
    Set yAxis = zAxis.CrossProduct(xAxis)

    ' Create a matrix to return.
    Dim newMatrix As Matrix
    Set newMatrix = tg.CreateMatrix
    Call newMatrix.SetCoordinateSystem(origin, xAxis.AsVector, yAxis.AsVector, zAxis.AsVector)

    ' Return the matrix.
    Set CreateMatrixFromZAxis = newMatrix
End Function
```

The pictures below illustrate the steps taken in the code above.

![Illustration of steps](image)
Setting Matrix Values
All of the examples so far have used the SetCoordinateSystem method of the Matrix object to set the values, but there are also other ways to manipulate the matrix. For example, there are the SetToRotateTo and SetToRotation methods to rotate the coordinate system the matrix defines. There is also the SetTranslation method which repositions the coordinate system. (Translation in this context means to move the occurrence.) Another useful feature of the Matrix object is the Cell property which lets you get and set the values of the matrix directly by specifying the row and column. Instead of calling SetCoordinateSystem, you could use the Cell property to create the same result, as shown below. Of course this isn’t as intuitive or efficient as using the SetCoordinateSystem, but it can be useful when you only need to modify a portion of the matrix.

' Create the matrix and define the desired coordinate system.
Dim oMatrix As Matrix
Set oMatrix = oTG.CreateMatrix
oMatrix.Cell(1, 1) = oXAxis.X
oMatrix.Cell(2, 1) = oXAxis.Y
oMatrix.Cell(3, 1) = oXAxis.Z
oMatrix.Cell(1, 2) = oYAxis.X
oMatrix.Cell(2, 2) = oYAxis.Y
oMatrix.Cell(3, 2) = oYAxis.Z
oMatrix.Cell(1, 3) = oZAxis.X
oMatrix.Cell(2, 3) = oZAxis.Y
oMatrix.Cell(3, 3) = oZAxis.Z
oMatrix.Cell(1, 4) = oOrigin.X
oMatrix.Cell(2, 4) = oOrigin.Y
oMatrix.Cell(3, 4) = oOrigin.Z

The Sub below also takes advantage of the Cell property and is a useful utility function when you’re working with matrices. It prints the values of the input matrix to the Immediate window.

' Print out the contents of a matrix to the VBA Immediate window.
Public Sub DumpMatrix(oMatrix As Matrix)
    Dim i As Integer
    For i = 1 To 4
        Debug.Print Format(oMatrix.Cell(i, 1), "0.000000") & ", " & _
        Format(oMatrix.Cell(i, 2), "0.000000") & ", " & _
        Format(oMatrix.Cell(i, 3), "0.000000") & ", " & _
        Format(oMatrix.Cell(i, 4), "0.000000")
    Next
End Sub

Here’s an example of the output of the above program when run against the matrix first discussed above. When you’re not getting the expected results when placing occurrences, this can be a useful tool in debugging what’s wrong with your matrix.

0.707107, -0.707107, 0.000000, 10.000000
0.707107, 0.707107, 0.000000, 5.000000
0.000000, 0.000000, 1.000000, 0.000000
0.000000, 0.000000, 0.000000, 1.000000
Other Add Methods

The Add method that's been used so far is the most common way of adding new occurrences to an assembly. There are some other add methods to support special cases of adding occurrences. For example, the AddCustomiPartMember is for adding a custom iPart, AddiPartMember is for adding a regular iPart, and AddiAssemblyMember is for adding an iAssembly. The AddUsingiMates method will take iMates into account during the placement. The AddVirtual method will create an occurrence for a virtual part.

The two other methods besides the Add method that are most used are the AddByComponentDefinition and AddWithOptions methods. The AddByComponentDefinition is similar to the Add method but instead of providing the filename of the part you provide a ComponentDefinition object. This means the part needs to be already open in Inventor. This is useful in cases where you've just created the part and it hasn't been saved to disk yet and doesn't have a filename. When you create a part in-place within an assembly, this functionality is used internally. Inventor creates a new part and inserts it into the assembly using the ComponentDefinition of the new part.

The AddWithOptions method supports the various options you see in the Place Component dialog when you place a subassembly into an assembly. For example, you can specify which level of detail, positional representation, and design view to use when placing the occurrence. The example below demonstrates this.

```vba
Public Sub AddOccurrenceWithRepresentations()
    ' Set a reference to the assembly component definition.
    ' This assumes an assembly document is active.
    Dim asmCompDef As AssemblyComponentDefinition
    Set asmCompDef = ThisApplication.ActiveDocument.ComponentDefinition

    ' Create a matrix. A new matrix is initialized with an identity matrix.
    Dim coordSys As Matrix
    Set coordSys = ThisApplication.TransientGeometry.CreateMatrix

    ' Create a new NameValueMap object.
    Dim options As NameValueMap
    Set options = ThisApplication.TransientObjects.CreateNameValueMap

    ' Set the representations to use when creating the occurrence.
    Call options.Add("LevelOfDetailRepresentation", "MyLODRep")
    Call options.Add("PositionalRepresentation", "MyPositionalRep")
    Call options.Add("DesignViewRepresentation", "MyDesignViewRep")
    Call options.Add("DesignViewAssociative", True)

    ' Add the occurrence.
    Dim occ As ComponentOccurrence
    Set occ = asmCompDef.Occurrences.AddWithOptions("C:\Temp\Reps.iam", coordSys, options)"
End Sub
```
Editing Occurrences

You edit existing occurrences by using the many properties and methods on the ComponentOccurrence object. Some of the more commonly used methods and properties are Visible, Name, SetRenderStyle, Suppress, Delete and Transformation. The Transformation property is the property that lets you get and set the matrix that’s defining the position and orientation of the occurrence. By setting this property you can reposition an occurrence. The example below uses some of the matrix functionality discussed earlier to reposition an assembly occurrence.

```vba
Public Sub ModifyOccurrence()
    ' Get a reference to an existing occurrence.
    Dim oAsmDoc As AssemblyDocument
    Set oAsmDoc = ThisApplication.ActiveDocument
    Dim oOcc As ComponentOccurrence
    Set oOcc = oAsmDoc.ComponentDefinition.Occurrences.ItemByName("Arrow:1")
    Dim oTG As TransientGeometry
    Set oTG = ThisApplication.TransientGeometry
    Dim oMatrix As Matrix
    Set oMatrix = oOcc.Transformation
    oOcc.Transformation = oMatrix
    ' Move the occurrence to (3,2,1), leaving its current orientation.
    Call oMatrix.SetTranslation(oTG.CreateVector(3, 2, 1))
    oOcc.Transformation = oMatrix
    ' Move the occurrence 5 cm in the X direction by changing the matrix directly.
    Set oMatrix = oOcc.Transformation
    oMatrix.Cell(1, 4) = oMatrix.Cell(1, 4) + 5
    oOcc.Transformation = oMatrix
End Sub
```

One important thing to notice from this sample is that when editing the matrix of an occurrence a copy of the matrix is first gotten from the occurrence using its Transformation property. This copy is then edited and then it is assigned back to the occurrence using the Transformation property. You can’t ever directly edit the matrix of an occurrence. You always have to get a copy, edit it, and assign it back.

Proxies

Proxies are an interesting concept that is completely hidden from the end-user but is something as a programmer will be helpful to understand. It’s a topic that can sometimes be a bit difficult to feel completely comfortable with. Before digging into what proxies are, let’s look at the internals of how Inventor works and why proxies are needed.

The first thing to understand is that assemblies only contain the top-level occurrences. They don’t contain any geometry or occurrences from subassemblies. The geometry and subassembly occurrences you see in the assembly are referenced through the top-level occurrences. As a user of Inventor, it doesn’t appear this is the case because you can select and use geometry for constraints, measuring, analysis, etc. as if that geometry exists in the top-level assembly.

One way to conceptualize this is to think of an assembly as a container. When you place a part into an assembly, you’re not copying the geometry into the assembly, but you’re creating a reference to the part. The part contains the real geometry. One way to visualize this is to think of what you’re seeing in the
assembly as a hologram of the part. It looks real and you can even “touch” it by selecting individual pieces of it, but it doesn’t really exist in the assembly.

Here’s an example to illustrate the need for proxies. We start with a simple part, shown below in the picture on the left. This part has been inserted twice into the assembly on the right. Using the previous analogy, what you see in the assembly are two holograms of the real part. The real geometry only exists in the part. What a proxy does is represent the real geometry in the assembly as if it actually exists in the assembly. It doesn’t do this by copying the geometry but by transparently redirecting geometry queries to the real geometry in the part and then massaging the results so they are correct in the context of the assembly. In this example, I’ve highlighted one of the circular edges in the part. If I query that edge, I can get the center of the circle in the part’s model space. In the assembly, that same edge has two representations; one for each occurrence. If I query the edge in the assembly and get the center of the circle, it will return the correct center of the edge in the context of the assembly. Each of the two edges in the assembly will return a different coordinate for its center. This works because of proxies.

Internally, when any queries are made on the edge, Inventor actually performs the query on the part where the real geometry exists. It then transforms the result from part space to assembly space and returns that result. Nothing extra was created in the assembly. The only difference is that Inventor did a little more work to calculate the result.

Internally, a proxy is essentially a reference to the real object, the edge in this example, along with a path. The path for a proxy is the occurrence chain from the occurrence in the top-level assembly down to the real object. The picture to the right shows the browser for the sample assembly. Conceptually, the path for each of the edges in the assembly is: “Widget:1/CircularEdge” and “Widget:2/CircularEdge”. The CircularEdge is the single circular edge in the part, but the path to get there is through a different occurrence and the results of any queries on the edge will be transformed based on the occurrence path.
Here’s another example to show that the path is not limited to a single occurrence. In fact the length of the proxy path is based on the number of levels in the assembly, so it can be any length. In this example, the assembly from the previous example has been inserted twice into another assembly. The path to one of the four holes now is “SubAsm:1/Widget:1/CircularEdge”. An interesting fact to point out here is that the top assembly only contains the two occurrences SubAsm:1 and SubAsm:2. The lower level occurrences in the subassembly are represented by ComponentOccurrenceProxy objects. Those occurrences only exist in SubAsm.iam, and are just being referenced into the top assembly. If the Transformation property on any of those occurrences is called, the matrix that’s returned will be in the context of the top-level assembly because Inventor is automatically taking care of the differences between the actual object and how it appears in the assembly.

All objects that can be selected in a part or assembly have a corresponding proxy object in the API. For example, for the Edge object there is an EdgeProxy object and for a SketchLine object there is a SketchLineProxy object. The EdgeProxy object supports all of the same methods and properties that an Edge object does. In fact, the EdgeProxy object is derived from the Edge object so in your code you can declare a variable as an Edge and it will be able to reference an Edge or an EdgeProxy. Because of this, even when writing code, the existence of proxies is mostly transparent. You get an object and perform queries on it, and the results you get back are correct, whether it was the real object or a proxy object you don’t need to do anything different.

As mentioned above, proxy objects support all of the same methods and properties the real object supports, but they also support two additional properties; ContainingOccurrence and NativeObject. The ContainingOccurrence property returns the ComponentOccurrence object that the proxy is referenced through. In the last example, the ContainingOccurrence property of one of the highlighted EdgeProxy objects will return the ComponentOccurrenceProxy for Widget:1. Since this is a proxy it also supports the ContainingOccurrence property which returns the ComponentOccurrence object for SubAsm:1. The occurrence for SubAsm:1 is a real occurrence and not a proxy because we’ve reached the top level of the assembly.
Another unique property that proxy objects support is *NativeObject*. The NativeObject property returns the real object the proxy represents. In the last example, the NativeObject property of the EdgeProxy of the circular edge will return the real Edge object in the part.

A property that is common between the real objects and proxy objects is the Parent property. Although they both share this property, the real object and its corresponding proxy object will return a different parent. In the previous example, the Parent property of the real edge will return the SurfaceBody object in the part, while the Parent property of the EdgeProxy will return the SurfaceBodyProxy object that represents the body in the assembly.

As I already mentioned, when querying geometry you typically don’t need to do anything special when working with assemblies because proxies take care of everything automatically. However, there are cases where you do need to be aware of proxies and will need to create them yourself. The most common reason for creating them is when you programmatically construct an assembly and apply assembly constraints. Here’s an example of a bolt we want to automatically insert into an assembly. The circular edge being selected in the picture to the right is the edge that will be used as input for an insert constraint. In order to be able to find this specific edge when this part is used within the assembly we’ll assume that an attribute has been added to the edge. (To learn more about attributes visit blogs.autodesk.com/modthemachine and search for attributes.)

Here’s an example assembly where we have two parts that we want to create an insert constraint between. Let’s assume you’ve had the user select a circular edge on the box. The selection will automatically return an EdgeProxy. Now we need to get an EdgeProxy for the edge of the bolt. An attribute was added to the edge to allow us to find it; however this attribute exists in the bolt part, not in the assembly. If we query for that attribute from the assembly document, it won’t find it. Instead we need to query for the attributed edge from within the bolt document. Because we’re querying from within the bolt document, the edge returned will be the actual edge, not the proxy. To create an insert constraint you need proxies for both of the edges because you need to specify the edges in the context of the top level assembly. The API provides support to construct a proxy. Remember that constructing a proxy doesn’t actually create anything in the assembly; it’s just creating the API object that contains the full path to the real object. The sample code below illustrates asking the user to select an edge, adding the bolt into the assembly, finding the edge of the bolt using its attribute, creating a proxy of the edge, and finally creating a constraint between the two edges.
Automating Autodesk® Inventor® Assemblies Using the API
Public Sub AddBolt()
  ' Get the active assembly.
  Dim asmDoc As AssemblyDocument
  Set asmDoc = ThisApplication.ActiveDocument

  ' Have the user select a circular edge.
  Dim partEdge As Edge
  Set partEdge = ThisApplication.CommandManager.Pick(kPartEdgeCircularFilter, _
      "Select a circular edge.")
  If partEdge Is Nothing Then
    Exit Sub
  End If

  ' Place the bolt into the assembly.
  Dim boltOcc As ComponentOccurrence
  Set boltOcc = asmDoc.ComponentDefinition.occurrences.Add("C:\Temp\Bolt.ipt",
      ThisApplication.TransientGeometry.CreateMatrix)

  ' Get the part document of the bolt.
  Dim boltDoc As PartDocument
  Set boltDoc = boltOcc.Definition.Document

  ' Query the attributes in the part for the attribute set named "InsertEdge".
  Dim attribSets As AttributeSetsEnumerator
  Set attribSets = boltDoc.AttributeManager.FindAttributeSets("InsertEdge")

  ' Assume success and get the parent from first item returned, which will be the edge.
  Dim boltEdge As Edge
  Set boltEdge = attribSets.Item(1).Parent.Parent

  ' Create a proxy for the edge.
  Dim boltEdgeProxy As EdgeProxy
  Call boltOcc.CreateGeometryProxy(boltEdge, boltEdgeProxy)

  ' Create a constraint.
  Call asmDoc.ComponentDefinition.Constraints.AddInsertConstraint(partEdge, _
      boltEdgeProxy, True, 0)
End Sub

The sample above illustrates a few important points. The `Definition` property of the `ComponentOccurrence` object returns the `ComponentDefinition` object of the document the occurrence is referencing. In this case, the `PartComponentDefinition` of the bolt part is returned. This is the same `PartComponentDefinition` object you would get when using the `PartDocument.ComponentDefinition` property. This gives you access to all of the part document API functionality. For example you could access and change parameter values and could even create additional features.

Once the edge within the part has been found using attributes, a proxy needs to be created to represent this edge in the context of the assembly. This is done using the `CreateGeometryProxy` method of the `ComponentOccurrence` object. This method takes in the object you want to create the proxy for and returns a proxy of the object that's specific to the occurrence you called the `CreateGeometryProxy` method on. We now have an object that represents the edge in the context of the assembly. The final line of the sample uses the created proxy of the bolt edge and the selected proxy of the part edge to create an insert constraint.