



How Deep is the Rabbit Hole? Examining the Matrix and other Inventor® Math and Geometry Objects

Brian Ekins – Autodesk

DE205-2 This session discusses concepts and objects within the Inventor API that are used to represent and manipulate geometry. We'll discuss how solids are represented within Inventor and the API (B-Rep), as well as the various geometry objects and how they're related to solids. We'll also discuss the various mathematical utility objects including matrices and vectors. Finally, we'll look at how Inventor provides geometry within assemblies using proxy objects.

About the Speaker:

Brian is a designer for 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 was the original designer of the Inventor® API and has presented at conferences and taught classes throughout the world to thousands of users and programmers.

brian.ekins@autodesk.com

As Morpheus says to Neo, “This is your last chance. After this, there is no turning back. You take the blue pill - the story ends, you wake up in your bed and believe whatever you want to believe. You take the red pill - you stay in Wonderland and I show you how deep the rabbit-hole goes.” It’s time to take the red pill and continue reading.

Working interactively with Inventor is a bit like being in the Matrix. It does a great job of providing a nice sensible view of the world and hiding the gory details. This paper takes away that veneer and examines the guts of Inventor parts and assemblies.

Geometry

The primary use of Inventor is to either create geometry or use existing geometry as input for a task (i.e. drawings, analysis, or machining). When creating a part you start by defining 2-D geometry in a sketch and then use that as input to features to create a 3-D solid model. Because Inventor is parametric, you can quickly modify the model by changing parameters or other inputs and then recompute it. Inventor is doing a lot of work behind the scenes to turn these logical commands into the corresponding geometry. Here’s an overview of the various types of geometry in Inventor.

Entities vs. Geometry

The API uses consistent terminology to differentiate between some concepts. The terms *entity* and *geometry* might often be used interchangeably but in the API they have distinct meanings. The term *entity* is used for any object that is selectable within the user-interface. This could be a feature, dimension, a face of a solid, or a circle in a sketch. The term *geometry* is used to refer to the actual geometric definition of an entity. For example you can select the edge of a part. In this case the entity is an Edge object. From the entity you can get the geometry which can be one of several types of geometry objects (line, arc, spline, etc.).

Geometry in Inventor is exposed using something called *transient geometry*. This is a term given to a group of objects that define specific geometric shapes. The transient portion of the name indicates that these objects are temporary and aren’t saved by Inventor. Here’s an example of how this works. Let’s say you have a rectangular solid block and have selected one of the edges. You’ve selected an entity which in this case is an Edge object. From the Edge object we can get its geometry which in this example will return a LineSegment object. The LineSegment object is the transient geometry object. It has properties that allow you to get the geometric information associated with a line segment; the start and end points.

An interesting aspect of the transient geometry objects is that they’re provided as “tear-off” objects. What this means is that when you get the geometry from an entity, the geometry you get accurately represents the shape of the entity at that time, but it’s no longer associated with that entity. For example, the LineSegment object you get back from the edge does correctly report the start and end points of the edge but if the edge should change, (i.e. you change the length of the extrusion), the geometry object will not reflect this. It provides a snapshot of that entity’s geometry at the point in time when you got the geometry. It’s also possible to edit the geometry object. For example, you can change the position of the start and end points; however, this won’t cause the entity the geometry was obtained from to change. Once you get the geometry there is no relationship between the entity and the geometry.

It’s also possible to create transient geometry directly without getting it from an entity. This is supported by the TransientGeometry object which is accessed using the TransientGeometry property of the Application object. This object is an API utility object and doesn’t represent anything you see in Inventor’s user-interface. All it does is provide a set of methods to create various types of geometry. For example

there is the CreateLineSegment method. This takes the start and end points as input and creates and returns a LineSegment object. Nothing changes within Inventor when you do this. You don't see the line in the graphics window and if you save the current document, the line isn't saved. What you've created is only the geometric definition of a line, not a line entity.

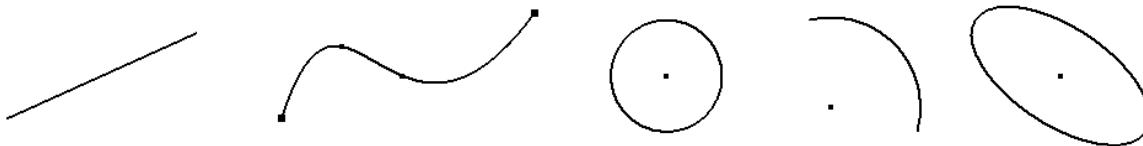
Points

There are a few entities in Inventor that represent a point. A vertex, a work point, a 3D sketch point, and a 2D sketch point. A vertex, work point, and 3D sketch point always define a 3D point in model space. A 2D sketch point always defines a 2D point in sketch space. From each of these entities you can get a transient geometry point. The 3D objects return a Point object and the 2D sketch point returns a Point2d object. The point geometry objects provide access to the x,y,z or x,y coordinates of the point.

Instead of a Point object the API could have chosen to use an array of three doubles to define a point. A point object provides the benefit of wrapping this information within a single object but probably most important is that it supports additional functions that are useful when working with points. For example in addition to being able to get and set the x, y, z coordinates of the point, it supports the DistanceTo method that returns the distance between this point and another point. There are several other useful methods it supports that make a Point object much easier to work with than an array of three doubles.

Curves

When you hear the term "curve" you probably think of a spline but in the general case a curve is any wireframe geometry; spline, circle, arc, line, etc. There are a lot of 3D and 2D curve entities supported by Inventor including Edge, SketchLine, SketchArc, WorkAxis, SketchSpline, etc. From each of these entities you can get a geometry object that tells you the shape of the entity. Some example geometry objects you get from the various curve entities are LineSegment, BSplineCurve, Circle, Arc, etc.



From a curve *entity* you can get its *geometry*. Let's look at an example. The sample below gets the currently selected edge, which the examples below will use.

```
Public Sub CurveGeometry()
    ' Get the active document.
    Dim oPartDoc As PartDocument
    Set oPartDoc = ThisApplication.ActiveDocument

    ' Get the selected edge.
    On Error Resume Next
    Dim oEdge As Edge
    Set oEdge = oPartDoc.SelectSet.Item(1)
    If Err Then
        MsgBox "An edge must be selected."
        Exit Sub
    End If
    On Error GoTo 0
End Sub
```

With many entities you know the type of geometry it represents because the entity type is specific to the geometry. For example, a SketchLine will always return a LineSegment2d geometry object as its

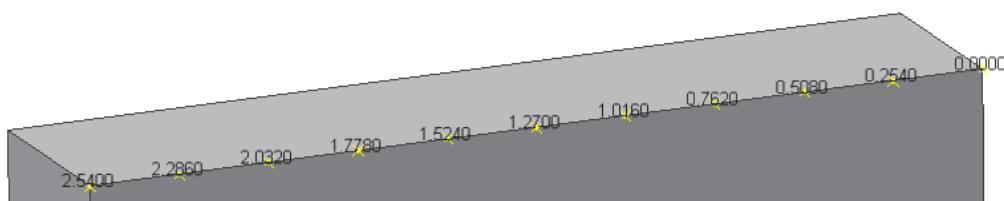
geometry, a SketchLine3D object will always return a LineSegment as its geometry, and a SketchCircle3D object will always return a Circle object as its geometry. An Edge object is unique in that it can be many different shapes. This means that an Edge object can return different types of objects as its geometry depending on what shape the edge is. The code below, which is an extension of the previous sample, uses the CurveType property of the edge to know what type of geometry will be returned and then prints some of the geometry information.

```
' Check the geometry type and print out some geometry specific information.
Select Case oEdge.GeometryType
    Case kLineSegmentCurve
        Dim oLineSegment As LineSegment
        Set oLineSegment = oEdge.Geometry
        Debug.Print "Start point: " & PointString(oLineSegment.StartPoint)
        Debug.Print "End point: " & PointString(oLineSegment.EndPoint)
    Case kCircleCurve
        Dim oCircle As Inventor.Circle
        Set oCircle = oEdge.Geometry
        Debug.Print "Center point: " & PointString(oCircle.Center)
        Debug.Print "Radius: " & Format(oCircle.Radius, "0.000000")
    Case kCircularArcCurve
        Dim dPi As Double
        dPi = Atn(1) * 4
        Dim oArc As Inventor.Arc3d
        Set oArc = oEdge.Geometry
        Debug.Print "Center point: " & PointString(oArc.Center)
        Debug.Print "Radius: " & Format(oArc.Radius, "0.000000")
        Debug.Print "Sweep: " & Format(oArc.SweepAngle * (180 / dPi), "0.000000")
    Case Else
        Debug.Print "Unsupported geometry type selected."
    End Select
End Sub

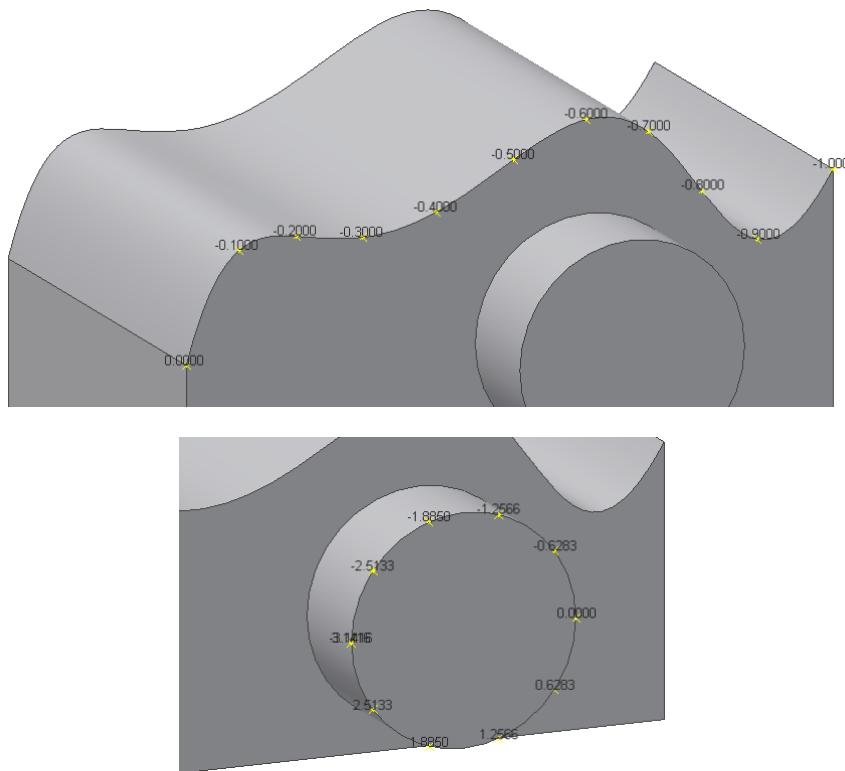
' Given a point or vector return a string containing the X,Y,Z coordinates.
Private Function PointString(PointOrVector As Object) As String
    PointString = Format(PointOrVector.X, "0.000000") & "," & _
                  Format(PointOrVector.Y, "0.000000") & "," & _
                  Format(PointOrVector.Z, "0.000000")
End Function
```

You can also perform other queries on curves using something called the CurveEvaluator object. You get a CurveEvaluator object from any of the transient geometry curve objects. You can also get a CurveEvaluator object directly from an Edge object. The types of analysis that the curve evaluator performs are general and can be used for any shape curve.

To use the curve evaluator there is one concept that's important to understand. That is the idea of *parametric space*, or in the case of curves you can think of it as *curve space*. To illustrate this, look at the picture below. If you treat a curve as a number line you can specify any location on the curve given a single value. The picture below illustrates this with an edge. The curve space along the curve goes from 0 to 2.54. A series of locations are identified along the edge but any position can be identified by its position in "curve space".



This concept applies to all shapes of edges. Below, the parameter space of a spline and a circle are illustrated. As you can see from these three examples, the range of parameter values can be different for each curve. In the example above it goes from 0 to 2.54, in the examples below it goes from -1 to 0 and from $-\pi$ to π .



Understanding the concept of parameter space is critical to using most of the functionality supported by the CurveEvaluator object. Most of the methods on the CurveEvaluator object either take parameter values as input or return them as output. Here's a list of some of the more commonly used methods on the CurveEvaluator object.

GetParamExtents – Returns the minimum and maximum parameter values of the curve.

GetPointAtParam – Returns the model space point for a given parameter value.

GetParamAtPoint – Given a position in model space on the curve it returns the parameter value at that point.

GetLengthAtParam – Returns the actual length of the curve between two parameter values. Using the minimum and maximum parameters returned by GetParamExtents will give the entire length of the curve.

GetParamAtLength – Returns the parameter as measured a given distance along the curve from another parameter point.

GetTangent – Returns tangent vector on the curve at a given parameter.

GetCurvature – Returns the curvature at the given parameter.

Here's the code that was used as a basis for the previous pictures that show a curve's parameter space. The GetParamExtents and GetPointAtParam methods are used. It uses the selection functionality illustrated in the first example above.

```

' Get the evaluator from the curve.
Dim oCurveEval As CurveEvaluator
Set oCurveEval = oEdge.Evaluator

' Get the parametric range of the curve.
Dim dMinParam As Double
Dim dMaxParam As Double
Call oCurveEval.GetParamExtents(dMinParam, dMaxParam)

' Set a reference to the TransientGeometry object.
Dim oTG As TransientGeometry
Set oTG = ThisApplication.TransientGeometry

' Iterate 10 steps over the curve length and print the
' parameter values and corresponding model points.
Dim i As Integer
For i = 0 To 10
    ' Calculate the current parameter to evaluate.
    Dim currentParam As Double
    currentParam = dMinParam + ((dMaxParam - dMinParam) / 10) * i

    ' Assign the value to an array since the GetPointAtParam
    ' takes an array as input.
    Dim adParam(0) As Double
    adParam(0) = currentParam

    ' Get the coordinates of the parameter point in model space.
    Dim adPoints(2) As Double
    Call oCurveEval.GetPointAtParam(adParam, adPoints)

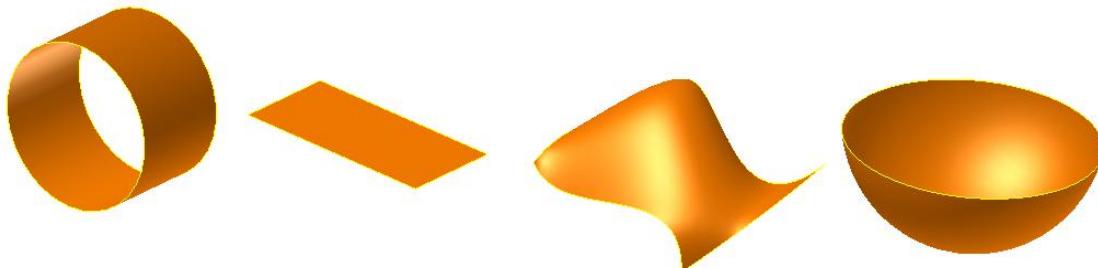
    ' Print information about this point.
    Debug.Print "Parameter: " & Format(currentParam, "0.0000") &
               " Coordinate: " & Format(adPoints(0), "0.000000") & "," & _
               Format(adPoints(1), "0.000000") & "," & _
               Format(adPoints(2), "0.000000")

    Next
End Sub

```

Surfaces

Surfaces are represented in Inventor as solid and surface models. Work planes can also be considered surfaces in many cases. Surfaces have a lot of the same concepts as curves. In a couple of aspects they're simpler because they only exist in 3D space and the only entities that represent a surface are the Face and WorkPlane objects.



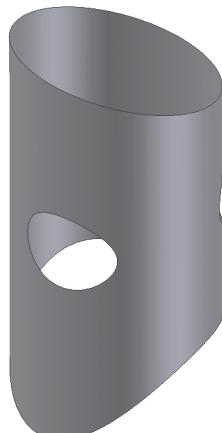
Like curves, you get a geometry object from an entity and use the geometry to understand the shape of the entity. WorkPlane objects always return a plane for its geometry. Face objects can return one of several different types of objects, depending on the shape of the face. The SurfaceType property of the Face object can be used to determine what type of geometry the Face.Geometry property will return. Here's a version of the previous curve program modified to work with faces.

```
Public Sub SurfaceGeometry()
    ' Get the active document.
    Dim oPartDoc As PartDocument
    Set oPartDoc = ThisApplication.ActiveDocument

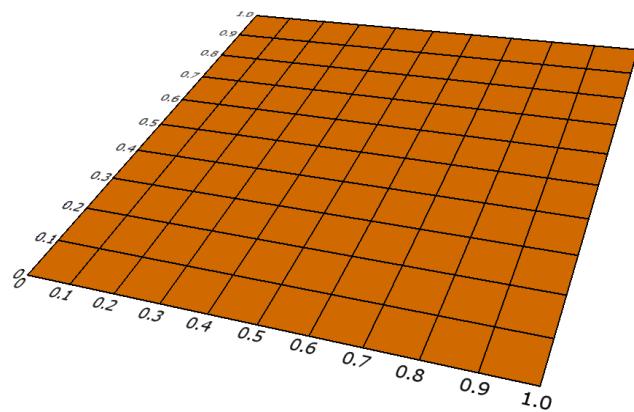
    ' Get the selected face.
    On Error Resume Next
    Dim oFace As Face
    Set oFace = oPartDoc.SelectSet.Item(1)
    If Err Then
        MsgBox "A face must be selected."
        Exit Sub
    End If
    On Error GoTo 0

    ' Check the geometry type and print out some geometry specific information.
    Select Case oFace.SurfaceType
        Case kPlaneSurface
            Dim oPlane As Plane
            Set oPlane = oFace.Geometry
            Debug.Print "Planar face"
            Debug.Print " Root point: " & PointString(oPlane.RootPoint)
            Debug.Print " Normal vector: " & PointString(oPlane.Normal)
        Case kCylinderSurface
            Dim oCylinder As Cylinder
            Set oCylinder = oFace.Geometry
            Debug.Print "Cylindrical face"
            Debug.Print " Base point: " & PointString(oCylinder.BasePoint)
            Debug.Print " Axis vector: " & PointString(oCylinder.AxisVector)
            Debug.Print " Radius: " & Format(oCylinder.Radius, "0.000000")
        Case kSphereSurface
            Dim oSphere As Sphere
            Set oSphere = oFace.Geometry
            Debug.Print "Spherical face"
            Debug.Print " Center point: " & PointString(oSphere.CenterPoint)
            Debug.Print " Radius: " & Format(oSphere.Radius, "0.000000")
        Case Else
            Debug.Print "Unsupported geometry selected: " & TypeName(oFace.Geometry)
    End Select
End Sub
```

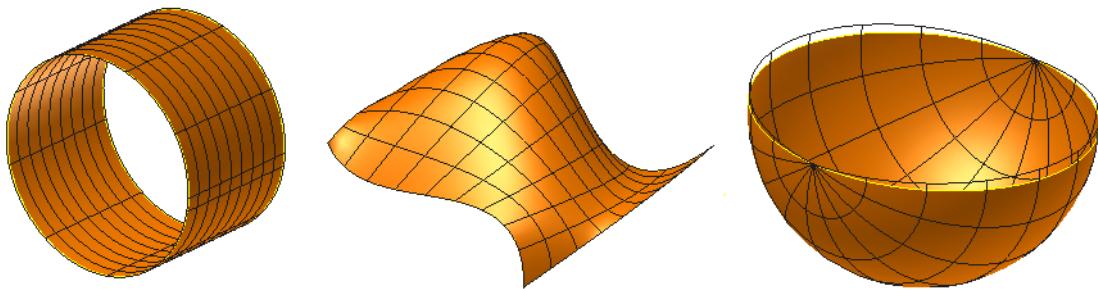
One important thing to understand is that most of the transient geometry surfaces are boundless. For example, a plane is defined only by a point and a vector, it doesn't have a size. A cylinder is defined by an origin point, axis, and radius; it doesn't have a length. The picture to the right shows a cylinder as it might exist in Inventor. Obviously this cylinder doesn't have infinite length and in addition it has holes and irregular trimmed edges. This can exist in Inventor because it's not the cylinder that's defining the edges but instead it's the edges of the face that define its boundaries. This is true for all surface types.



Just like curves, surfaces also have a parameter space. Unlike a curve, whose parameter space has a single dimension, the parameter space for a surface is two dimensional. Two parameter values can be used to specify any location on a surface. This is best visualized with a rectangular plane. To the right is an illustration showing a rectangular plane with lines drawn over it indicating the parametric space of the surface, just like any 2-D coordinate system. You can specify any point in the 2-D space given an X and Y value. For parametric space instead of X and Y, the directions are referred to as U and V.



Below are some surfaces with their parametric grid drawn over them. Even though the surfaces are not planar you can still uniquely identify any location on the surface given a U-V coordinate.



Just like curves, surfaces also support evaluations. The SurfaceEvaluator object can be obtained either from the Face object or from any of the surface geometry objects. It's generally better to use the evaluator on the Face object since it takes into account additional information provided by the solid. For example, when getting the normal from a face of a solid it will always point to the outside of the solid. The SurfaceEvaluator object supports similar functionality as the CurveEvaluator. Here are some of the more commonly used methods and properties.

ParamRangeRect – Returns the minimum and maximum u-v parameter values of the surface.

GetPointAtParam – Returns the model space point for a given u-v parameter.

GetParamAtPoint – Given a position in model space it returns the u-v parameter value of the point.

IsParamOnFace – Indicates if the specified u-v parameter is on the face. This is especially useful in the case where a face has holes in it. This tells you if the point is on a solid area of the face or not.

GetNormal – Returns the normal vectors for a given set of u-v parameters.

Below is a small program that demonstrates using some of these functions to get the normal vector for a point on a selected face. The point used is at the center of the parametric space of the surface. It uses the first portion of the previous program to get the selected face.

```

' Get the surface evaluator from the face.
Dim oSurfEval As SurfaceEvaluator
Set oSurfEval = oFace.Evaluator

' Get the parametric range of the surface.
Dim oParamRange As Box2d
Set oParamRange = oSurfEval.ParamRangeRect

' Calculate the u-v values at the parametric center of the surface.
' (This code is bigger than it should be to work around a VBA issue.)
Dim adParamCenter(1) As Double
Dim U As Double, V As Double
U = oParamRange.MinPoint.X
V = oParamRange.MaxPoint.X
adParamCenter(0) = (U + V) / 2
U = oParamRange.MinPoint.X
V = oParamRange.MaxPoint.X
adParamCenter(1) = (U + V) / 2

' Get the normal at the u-v parameter.
Dim adNormal(2) As Double
Call oSurfEval.GetNormal(adParamCenter, adNormal)

' Print the normal vector.
Debug.Print " Normal: " & Format(adNormal(0), "0.000000") & "," & _
           Format(adNormal(1), "0.000000") & "," & _
           Format(adNormal(2), "0.000000")

' Get the model space coordinate of the parameter so we
' know where in space the normal was calculated.
Dim adPoint(2) As Double
Call oSurfEval.GetPointAtParam(adParamCenter, adPoint)

' Print the coordinate.
Debug.Print " Normal coordinate: " & Format(adPoint(0), "0.000000") & "," & _
           Format(adPoint(1), "0.000000") & "," & _
           Format(adPoint(2), "0.000000")
End Sub

```

Strokes and Facets

The API also provides access to an approximation of curves and surfaces. The approximated data for a curve is a series of lines and is referred to as *strokes*. The approximated data for a surface is a set of triangles, which are referred to as *facets*. The primary use of this type of data is when you need to interact with other software that doesn't use smooth curves and/or surfaces. Inventor calculates and uses strokes and facets internally to send to the graphics card to display the Inventor model. Because of the rendering technique used it looks smooth but the picture you see is actually a set of small triangles. The model consists of smooth accurate surfaces and all modeling calculations use this model, but the display uses the triangle approximation.

The API provides access to the set of triangles and strokes that Inventor has already calculated for the display and you can also have Inventor generate triangles and strokes that represent the model to a specific tolerance. I won't spend a lot of time here discussing the details since this isn't commonly used but there are some samples in the API help that demonstrate this.

Solids

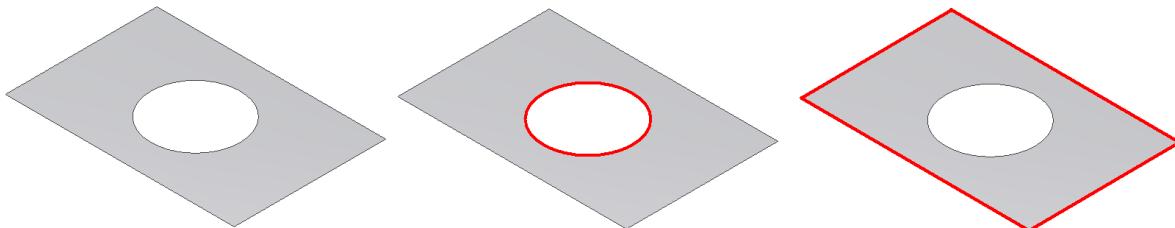
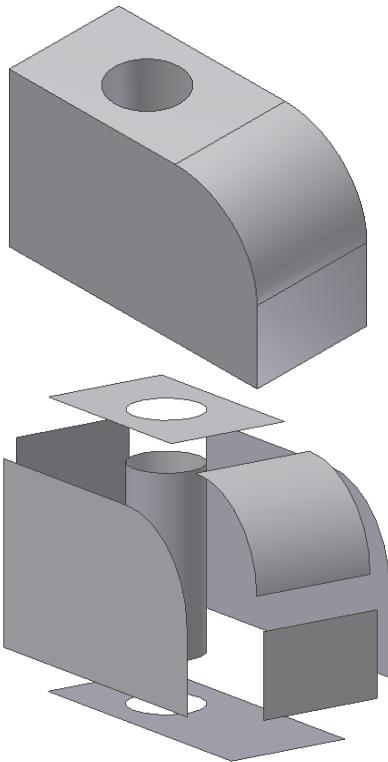
A typical part in Inventor is represented as a solid. A solid provides an intuitive way to work with geometry because it represents a real world object that you can manipulate. In Inventor, a solid is created and modified using features, which add and remove material from the solid. Because Inventor is a parametric associative modeler, you can edit the model by changing the various inputs of the feature and letting the model recompute.

Internally, the features are essentially a set of instructions. Inventor follows those instructions and constructs the model with the result being the solid model. This paper is focused on the result of those modeling operations, not the procedure to create it, so it focuses on the resulting solid and how to use the API to interrogate it.

A solid in Inventor is represented by a *Boundary Representation* (B-Rep) of the solid. This means it's a set of connected surfaces that define the outer boundary of the volume. Even though the model appears to be solid, (because you can cut into it, perform mass properties, analyze, and other solid specific operations), it's actually just surfaces. The connectivity of the surfaces is referred to as *topology* and introduces some other objects that are specific to a B-Rep model. To the right is a simple solid I'll use to discuss these concepts.

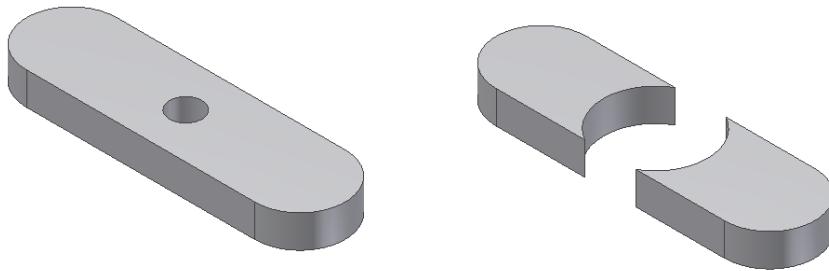
In the API a solid is represented by an object called *SurfaceBody*. Even though it has "Surface" as part of its name, a SurfaceBody is a generic object that represents any top-level B-Rep object including a solid or a surface. The primary difference being that a solid is a closed set of surfaces. In a SurfaceBody, each surface that makes up the body is represented as a *Face* object. In the example above, the solid consists of 8 faces. The exploded representation of the model to the right illustrates each of the faces. A Face object represents all surfaces regardless of its shape. The previous discussion of surfaces discusses accessing the geometry of the face.

Just having a group of faces that enclose a volume isn't enough to define a solid model. Those faces need to be connected. The connection between faces is defined by the *Edge* object and is represented geometrically as a curve. Each face knows the edges that define its boundaries and each edge knows which two faces it connects together. Another part of the topology that can be useful as a programmer is a *loop*. In the API these are represented by the EdgeLoop object. Loops represent a set of connected edges. The picture below shows a single face that has two loops. The interior loop is highlighted in the center picture and consists of a single edge. The outer loop is highlighted in the picture on the right and consists of four edges. Loops define the limits or boundaries of a face.



Another common B-Rep entity is the *Vertex* object. A vertex exists at the end of every edge and defines the connections between edges. Each edge knows about the vertex at each of its ends and each vertex knows which edges it joins together.

Another B-Rep entity, although not commonly used, is the *FaceShell* object. It's possible for a solid to be more than one piece. The picture below illustrates this. On the left is a single solid. The picture on the right shows the same solid after the diameter of the hole has been increased to cut the solid into two pieces. Even though there are two pieces it is still a single solid or through the API a single SurfaceBody object. Each of the pieces is represented as a FaceShell object. Most parts will have a single FaceShell, but it is possible to have more than one. Another example would be a hollow ball represented by two spheres. The entire ball is a single SurfaceBody but the inner and outer spheres are each a FaceShell. If you drilled a hole into the ball, which would connect the inner and outer spheres, then it will become a single FaceShell.



Assemblies

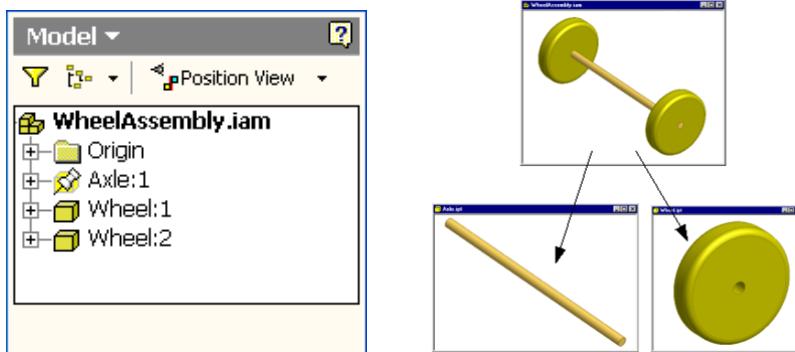
What's been discussed so far is how the geometry of a part is represented in Inventor. Part documents contain the geometric definition of a part and assemblies allow you to combine these parts together. A common need is to access the part geometry in the context of the assembly. Each part in an assembly is represented as an occurrence. In the API these are exposed as the ComponentOccurrence object. An occurrence can represent a part or a subassembly. When you run the Place Component command you're creating a new occurrence in the assembly. The ComponentOccurrence object supports the SurfaceBodies property which, in the case where the occurrence represents a part, gives you access to the geometry associated with the occurrence. There's an interesting issue here because of the fact that an assembly doesn't contain geometry but only has references to part documents that contain the geometry. If assemblies don't contain geometry how come in the user-interface it appears you can access the part geometry? This problem is solved by proxies.

Proxies

As already mentioned, assemblies only contain references to other files, not any geometry. However, this isn't the experience of the end-user when using Inventor to create and edit an assembly. For them, the geometry of the parts appears to be within the assembly. For example, when you're adding a mate constraint between two parts you're free to select a face on a part without any thought of where the actual geometry of that part exists. This same type of feeling is also presented through the API. That is you can traverse through the assembly and access the occurrence geometry as if it exists in the top assembly.

Let's look at the simple axle assembly shown below to see how this works. There are two instances of the same wheel part in this assembly. Each instance is represented by a ComponentOccurrence object. Let's say we need the user to select the cylinder that represents the hole through each of the wheels. The first question is how can the user select these cylinders if geometry doesn't exist in the assembly?

The second question is that assuming they can somehow select geometry, how does Inventor tell the hole cylinder of one wheel apart from the hole cylinder of the other wheel, since in reality there is only one hole cylinder which exists in the wheel part file?



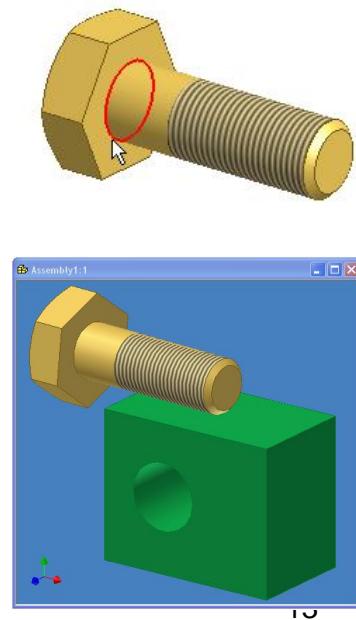
The answer to both of these questions is proxies. Proxies represent objects in an assembly as if that object actually exists in the assembly. Getting back to the hole face of the wheels, when the user selects a face in the assembly they're actually selecting a FaceProxy object. The FaceProxy object is derived from the Face object which means it supports all of the methods and properties that the Face does plus a few additional ones that are unique to proxies. Because it's derived from the Face object, in most cases you can treat it as a regular face object and don't even need to be aware that you're working with a proxy. You use the same methods and properties to work with the proxy object; it's just that the answers you receive will be adjusted to account for the position of the object within the assembly.

When selecting these cylindrical faces in the assembly, a different FaceProxy object is returned for each of the cylinders even though they both represent the same cylindrical face in the part. Queries on each one will return difference information since they're in different positions within the assembly.

Proxies are supported for a large number of objects. Proxy support is required for anything that is selectable within an assembly. Typically when you're obtaining a proxy object, either through traversing an assembly or through user selection, you don't need to be concerned that you're using a proxy object and not the actual object. This is what makes the proxy concept so powerful in that it simplifies working with objects from different files and provides the view that everything exists within the assembly, which is the same view the end-user has.

There are some cases where you need to create a proxy to use as input for other methods. The most common case for this 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 find this specific edge when this part is used within the assembly we'll assume that an attribute has been added to the edge.

Here's an example assembly where we have two parts that we want to create an insert constraint between. Let's assume we've obtained the occurrence of the box and used the B-Rep functionality of the API to find the circular edge of the hole. Since we accessed this edge by going through the occurrence and its associated SurfaceBody, it will be an EdgeProxy object, which means it behaves as if that edge actually exists in the assembly. 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. The API provides support to construct a proxy. The sample code below illustrates accessing the part document, querying for the edge, constructing the proxy, and finally creating the assembly constraint. It's assumed there is already an EdgeProxy for the circular edge of the box and a reference to the bolt occurrence.

```
' Access the bolt component definition from the bolt occurrence.
Dim oBoltCompDef As ComponentDefinition
Set oBoltCompDef = oBoltOccurrence.Definition

' Use the attribute manager of the bolt document to query for the edge.
Dim oAttribManager As AttributeManager
Set oAttribManager = oBoltCompDef.Document.AttributeManager
Dim oObjects As ObjectCollection
Set oObjects = oAttribManager.FindObjects("AutoBolts")

Dim oBoltEdge As Edge
Set oBoltEdge = oObjects.Item(1)

' Create a proxy for this edge.
Dim oAsmBoltEdge As EdgeProxy
Call oBoltOccurrence.CreateGeometryProxy(oBoltEdge, oAsmBoltEdge)

' Create the assembly constraint.
Call oDoc.ComponentDefinition.Constraints.AddInsertConstraint(oBlockEdge, _ 
    oAsmBoltEdge, True, 0)
```

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. Whenever you use the Definition property of the occurrence any calls you make on that ComponentDefinition will be in the context of the part, not the assembly. Because you now have access to the part document you have access to all of the part document API functionality. For example you could access and change parameter values and could even create additional features.

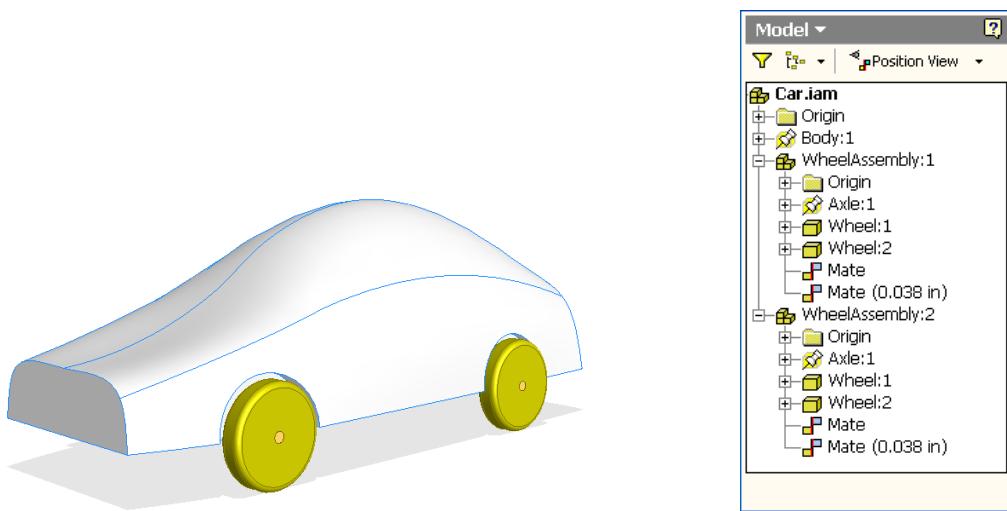
Once the edge within the part is obtained we need to create the proxy to represent this edge in the assembly. This is done using the CreateGeometryProxy method of the ComponentOccurrence object. This method takes in the object you want to create the proxy for and returns a proxy of the object that's 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 proxy of the bolt edge and the proxy of the box edge to create an insert constraint.

As discussed earlier, the various proxy objects are derived from the real object they represent. Because of this they support all of the same methods and properties as the original object, but they also support two additional properties that the original object does not; ContainingOccurrence and NativeObject. The ContainingOccurrence property returns the ComponentOccurrence object that the proxy is exposed through. In the example above, the ContainingOccurrence property of the bolt's edge proxy will return the occurrence of the bolt. The NativeObject property returns the actual object the proxy represents. In the example above, the NativeObject property of the EdgeProxy will return the real Edge object in the part.

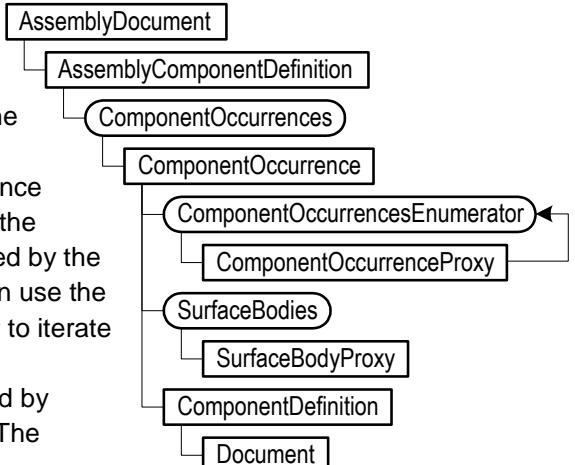
A property that is common between the real objects and proxy objects is the Parent property. Although they both share this property, the real object and the proxy object will return a different parent. In the previous example, the Parent property of the real edge will return the SurfaceBody object in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

Assembly Traversal

A common requirement for many applications is the ability to traverse through an assembly. To illustrate this let's look at a multi-level assembly. This assembly only has 2 levels, but the concepts illustrated here will work for any number of levels. In this example, the top-level assembly contains two instances of the wheel assembly and one instance of the body part.



Now let's take a look at the portion of the API that allows us to traverse not only a single level of an assembly but through all of its levels. From the AssemblyComponentDefinition you obtain the ComponentOccurrences collection object to begin the traversal. Iterating over the occurrences within the ComponentOccurrences collection returns ComponentOccurrence objects. If the occurrence represents a part, you can examine the geometry of the part using the surface body information returned by the occurrence. If the occurrence represents an assembly, you can use the SubOccurrences property of the ComponentOccurrence object to iterate over the occurrences within that subassembly. To traverse an assembly you keep descending through the occurrences owned by each ComponentOccurrence that represents a subassembly. The following sample code does this.



```

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
```

One thing to notice is that this example is not a single sub. Traversing an assembly of any depth requires recursion. Recursion is when a sub or function calls itself. The TraverseAsm sub does most of the work in this program by traversing the occurrences within each level of the current assembly and when it encounters a subassembly it calls itself to traverse that subassembly. This continues until it has gone through every level of the assembly. The AssemblyTraversal sub just kicks off the traversal process.

The results of this sample are simple in that it only prints the name of each occurrence as it encounters it. However, much more complicated processing could have been done with each occurrence. For example it could have constructed a custom BOM, where it would access the document referenced by each occurrence and extract iProperty information.

Math Utility Objects

TransientGeometry Object

We've seen how you can obtain geometry objects from various entities. You can also create many of the geometry objects directly using the TransientGeometry object. The TransientGeometry object is accessed from the Application object and supports various Add methods to create different geometry objects. All of the objects created using the TransientGeometry object have a maximum lifetime of the current session of Inventor and are not saved. In addition to the standard geometry objects, the TransientGeometry object also supports some mathematical type of objects that serve as utility objects within the API. Two of these mathematical related transient geometry objects are the Vector and Matrix objects. Because these objects don't have a visible counterpart they're probably the least understood. Below is a discussion of each one.

Vector Objects

A vector is a convenient way to specify direction and magnitude. A vector consists of three values; its x, y, and z components, (a 2d vector only has x and y components.) Although its data is very simple, the Vector object supports a rich set of methods and properties that allow you to easily query and manipulate it. Let's look at a couple of examples of how vectors can be used.

A common use of vectors is to define the movement of an object. This is typically referred to as the translation of an object, which means it is moved in space without any rotation. For example, the vector (3,1,0) would define the movement of an object 3 units in the x direction, 1 unit in the y direction, and 0 units in the z direction resulting in a total move of 3.162 units, which is illustrated below.



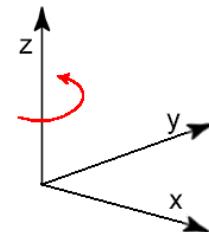
Another common use of vectors is to specify a direction. In the previous example a vector was used to define a direction and a distance to move a part. When a vector is used only to specify a direction then usually the UnitVector object is used. A UnitVector's length is always one. An example of when vectors are used to define direction is when you get orientation information from some of the transient geometry objects. For example, the Cylinder object supports the AxisVector property that returns a UnitVector object. This vector defines the direction of the axis of the cylinder.

As stated earlier, the functionality that makes the Vector object so powerful are the methods and properties it supports. Let's look at some of the more commonly used methods and how and why they're used. When defining a rectangular coordinate system you typically have a point that defines the origin and three vectors that define the x, y, and z axis directions. (These can be unit vectors since the magnitude doesn't matter.) For a valid rectangular coordinate system these vectors have to follow some rules:

1. The y-axis vector must be perpendicular to the x-axis.
2. The z-axis must be perpendicular to the x and y axes and for a right-hand coordinate system (which Inventor uses) it must be in the direction that follows the right-hand rule.

The trick is how to easily create three vectors that follow these rules. Let's look at a specific case and how to solve it. Let's say we want to define a coordinate system where the x-axis points in the direction defined by the vector $(3, 7, 6)$, and we want the y-axis to point "up", where up is in the positive Y direction. Finally the z-axis just needs to be valid based on the x and y axes. First let's look at a utility that makes what could be a nightmare to compute quite easy.

With vectors there's an operation called *cross product*, where you input two vectors and obtain a third as the result. Any two vectors (as long as they're not parallel) define a plane. If you take any two vectors and put their tails together, they will lie in a plane. Performing a cross product with these vectors returns another vector that is perpendicular to that plane. The picture to the right illustrates this by showing two vectors (labeled x and y) and crossing them to obtain the vector z. The vector z can be in either of two directions depending on which order you cross x and y. In the example shown, x is crossed into y which results in the z shown. This follows the right-hand rule: flatten your hand, extend your thumb, and point your fingers along the x-axis. Now, going from the x-axis towards the y-axis (in the direction that's the shortest between them), curl your fingers so they curl towards the y-axis. Your thumb will be pointing in the direction of the positive z-axis. If you had started with your fingers pointing in the y-axis direction and then curled them toward the x-axis your thumb would be pointing in the opposite direction.



Using this concept of crossing two vectors let's see how we can construct the coordinate system specified earlier. Here are the steps and then we'll look at the code to perform these steps.

1. Create a vector that defines the known x-axis, (3,7,6).
2. Create a vector that defines the y-axis direction, (0,1,0). (We know at this point that this is not the correct y-axis. This just defines the general direction we want the y-axis to go in.)
3. Cross the x-axis into the y-axis to get the z-axis. This results in the correct z-axis vector.
4. Cross the z-axis into the x-axis to get the y axis. This results in the correct y-axis vector.

```
Public Sub CreateCoordSystem()
    ' Set a reference to the TransientGeometry object.
    Dim oTG As TransientGeometry
    Set oTG = ThisApplication.TemporaryGeometry

    ' Create a vector for the x-axis.
    Dim oXAxis As UnitVector
    Set oXAxis = oTG.CreateUnitVector(3, 7, 6)

    ' Create a vector in the general y-axis direction.
    Dim oYAxis As UnitVector
    Set oYAxis = oTG.CreateUnitVector(0, 1, 0)

    ' Cross the x into the y to get the z-axis.
    Dim oZAxis As UnitVector
    Set oZAxis = oXAxis.CrossProduct(oYAxis)

    ' Cross the z into the x to get the correct y axis.
    Set oYAxis = oZAxis.CrossProduct(oXAxis)

    ' Display the results.
    Debug.Print "x-axis: " & oXAxis.X & ", " & oXAxis.Y & ", " & oXAxis.Z
    Debug.Print "y-axis: " & oYAxis.X & ", " & oYAxis.Y & ", " & oYAxis.Z
    Debug.Print "z-axis: " & oZAxis.X & ", " & oZAxis.Y & ", " & oZAxis.Z
End Sub
```

Here are some of the other functions supported by the vector object. These methods allow you to compare two vectors in various ways: AngleTo, IsEqualTo, IsParallelTo and IsPerpendicularTo. These methods allow you to combine two vectors: AddVector, SubtractVector, CrossProduct, and DotProduct. The Vector object (vs. the UnitVector object) also supports some functionality that deals with the magnitude of the vector. These are Length, Normalize (which makes the length 1 unit), and ScaleBy. Finally, you may have a Vector and need a UnitVector or vice versa. The AsUnitVector method on the Vector and AsVector on the UnitVector allow you to easily obtain the opposite type.

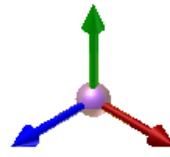
Matrix Objects

A matrix is defined as a rectangular array of numbers. For 3-D space, Inventor supports the Matrix object and for 2-D space it supports the Matrix2d object. The Matrix object is a 4x4 rectangular array (four columns and four rows), and the Matrix2d is a 3x3 rectangular array. A typical 3d 4x4 matrix is shown to the right. The Matrix object encapsulates these 16 values. Considering the data, a matrix is a simple object but it quickly gets much more complex as we try to understand how to use this information. Out of all the transient geometry objects, the Matrix is the least understood and causes programmers the most problems.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The following describes a practical approach to matrices that has helped me. Hopefully this will also make sense to you. Depending on the task I'm doing I tend to look at a matrix in one of two ways. The first is that a matrix is a way to define a coordinate system. The second is that a matrix defines a transformation. The easiest and most useful is the coordinate system view.

First, let's look at what a coordinate system consists of and then we'll see how a matrix provides this information. A coordinate system defines a position and orientation in space. For three-dimensional space the origin is a 3d coordinate point (x,y,z) defined relative to model space. This defines the position of the coordinate system. The orientation is defined by specifying the direction of the x, y, and z axes.

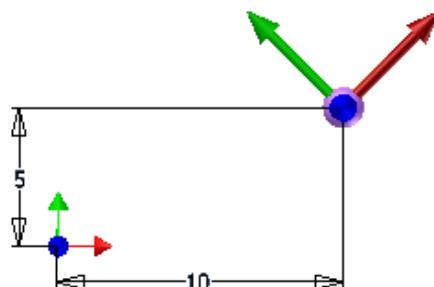


For a rectangular coordinate system, which is what Inventor always uses, there are certain rules that are enforced as was discussed earlier. These are that the x-axis and y-axis are perpendicular to each other and the z-axis is perpendicular to both the x-axis and y-axis and follows the right-hand rule. (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.

The figure to the right illustrates how a matrix contains this information. First, when looking at or manipulating a matrix you can always ignore the bottom row. The bottom row will always have the values 0, 0, 0, 1. The other twelve values define the coordinate system. The first column defines the x-axis direction (1,0,0). The second column defines the y-axis direction (0,1,0), and the third column defines the z-axis direction (0,0,1). The last column defines the position of the coordinate system's origin (0,0,0). The matrix shown to the right is a special matrix known as an identity matrix. That is, the origin is at (0,0,0) and the x, y, and z axes are in the same direction as the base coordinate system axes. It doesn't define any translation or rotation but is identical to the base coordinate system.

X-Axis	Y-Axis	Z-Axis	Origin
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	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?



Let's look first at the x-axis. If we rotate the coordinate system 45° around the z-axis then the x-axis is pointing 45° in the positive x and y directions. A simple vector that points in this direction could be defined as (1, 1, 0), but since these vectors need to be normalized (have a length of 1) it results in (0.707, 0.707, 0). The y-axis points in the negative x and positive y direction which results in the vector (-0.707, 0.707, 0). The z-axis remains unchanged (0, 0, 1). Finally, the position of the matrix is defined at (10, 5, 0). The resulting matrix is shown to the right.

$$\begin{pmatrix} 0.707 & -0.707 & 0 & 10 \\ 0.707 & 0.707 & 0 & 5 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The code below demonstrates one approach to constructing the matrix above. It takes advantage of some features in the API to do this. First, when defining the axis vectors it uses the UnitVector type of vector. This guarantees that the axis vectors will be normalized since unit vectors always have a magnitude of 1. Second, it uses the SetCoordinateSystem method of the Matrix object to define the matrix, which simplifies setting the correct values for the matrix. When defining the coordinate system, remember that all units will be internal units which means distances are always defined in centimeters.

```
Public Sub MatrixDefineSample()
    ' Set a reference to the TransientGeometry object and create a variable for Pi.
    Dim oTG As TransientGeometry
    Set oTG = ThisApplication.TemporaryGeometry
    Dim dPi As Double
    dPi = Atan(1) * 4

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

    ' Define the axis vectors. Pi/4 is 45 degrees in radians.
    Dim oXAxis As UnitVector
    Dim oYAxis As UnitVector
    Dim oZAxis As UnitVector
    Set oXAxis = oTG.CreateUnitVector(Cos(dPi / 4), Sin(dPi / 4), 0)
    Set oYAxis = oTG.CreateUnitVector(-Cos(dPi / 4), Sin(dPi / 4), 0)
    Set oZAxis = oTG.CreateUnitVector(0, 0, 1)

    ' Create the matrix and define the desired coordinate system.
    Dim oMatrix As Matrix
    Set oMatrix = oTG.CreateMatrix
    Call oMatrix.SetCoordinateSystem(oOrigin, oXAxis.AsVector,
                                    oYAxis.AsVector, oZAxis.AsVector)
End Sub
```

Instead of using the SetCoordinateSystem method you can set the matrix one value at a time. Using the **Cell** property of the matrix you can read and write an individual cell. The code below results in the same matrix as the SetCoordinateSystem method created in the sample above. As you can see, using the SetCoordinateSystem is easier and makes the program more readable. However, sometimes setting individual cells is convenient when making small changes to a 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
```

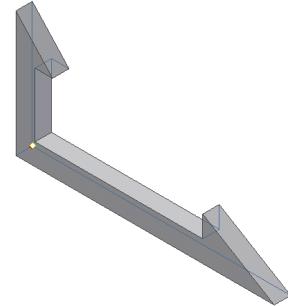
The code below is also useful when working with matrices. It dumps out the contents of a matrix to allow you to better visualize it.

```
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
```

Running this program results in the output shown below when the matrix created previously is provided as input.

```
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
```

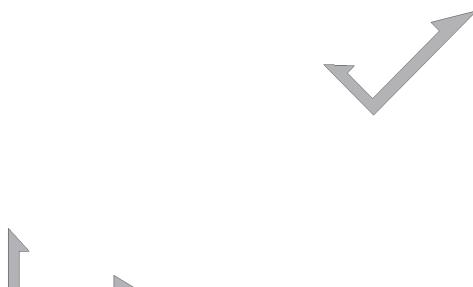
The most common use of matrices is when working with assembly occurrences. When placing an occurrence into an assembly with the API you use a matrix to specify the position and orientation of the occurrence. Let's look at a specific example using the part to the right. Creating a part similar to the one to the right can be useful while debugging your programs that use matrices because since it looks like a coordinate system it allows you to better visualize the result of the matrix. The part is constructed so the corner where the two arrows connect is at the origin of the part coordinate system and the arrow along the X axis is longer to make it easy to identify.



If you open a new assembly and manually place an instance of this part into the assembly you'll see that the part is positioned at the origin of the assembly and the axes are the same as the assembly axes. When the first part is placed manually into an assembly, internally Inventor positions it using an identity matrix. If we add the following code to the previous program it will place a part at (10,5,0) and rotated 45° around the z-axis.

```
' Place a part into the active assembly using the defined matrix.
Dim oOcc As ComponentOccurrence
Set oOcc = ThisApplication.ActiveDocument.ComponentDefinitionOccurrences.Add(
    "C:\Temp\Arrow.ipt", oMatrix)
```

This creates the resulting assembly shown below where the occurrence at the upper-right is the one placed using the API.



You can reposition an occurrence by editing its matrix. The Transformation property of the ComponentOccurrence object provides read and write access to this matrix. Below are some examples of edits to the position of an occurrence.

```
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

    ' Move the occurrence to (3,2,1).
    Dim oMatrix As Matrix
    Set oMatrix = oOcc.Transformation
    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.

A Matrix as a Transformation

Another way to think of a matrix is that it defines a change in position and orientation. This is conceptually different than using a matrix to define a coordinate system because in this case you're not defining an absolute position but instead are defining a change in position. To do this you create a matrix that defines the change you want and then apply that change to another matrix.

Here's a simple example to illustrate the concept. Let's say you want to move some parts in an assembly 5 cm in the x direction. Here's one approach using the information we learned previously about a matrix defining a coordinate system. You can see that it directly edits the matrix to reposition the coordinate system an additional 5 cm in the x direction.

```
' Iterate through the occurrences in the assembly.
Dim oOcc As ComponentOccurrence
For Each oOcc In oAsmDoc.ComponentDefinition.Occurrences
    ' Get the matrix from the current occurrence.
    Dim oMatrix As Matrix
    Set oMatrix = oOcc.Transformation

    ' Edit the cell that defines the X component of the origin.
    oMatrix.Cell(1, 4) = oMatrix.Cell(1, 4) + 5

    ' Set the transformation of the occurrence using the modified matrix.
    oOcc.Transformation = oMatrix
```

[Next](#)

Here's the equivalent but using a matrix that defines the change and applying that matrix to the existing matrix.

```
' Create the matrix that defines the transform.
Dim oTransMatrix As Matrix
Set oTransMatrix = ThisApplication.TransientGeometry.CreateMatrix
oTransMatrix.Cell(1, 4) = 5

' Iterate through the occurrences in the assembly.
Dim oOcc As ComponentOccurrence
For Each oOcc In oAsmDoc.ComponentDefinition.Occurrences
    ' Get the matrix from the current occurrence.
    Dim oMatrix As Matrix
    Set oMatrix = oOcc.Transformation

    ' Apply the transform to the matrix.
    Call oMatrix.TransformBy(oTransMatrix)

    ' Set the transformation of the occurrence using the modified matrix.
    oOcc.Transformation = oMatrix
Next
```

The two programs create the same result. For this simple example, editing the matrix directly, as in the first example, looks the simplest. However, if the change you need to apply becomes much more complicated than a simple move then it becomes much easier to use the transformation matrix to apply the change instead of having to recompute the coordinate system each time. For complicated transformations it's possible to combine multiple changes within a single matrix. You can define individual transforms but combine them into a single matrix, as shown below

To look at combining multiple changes within a single matrix, let's say we want to rotate an occurrence 45° around the x-axis, and then 30° around the y-axis, and finally move it 5 cm in the x direction and 3 cm in the z direction. How would you define a matrix that defines the result of these changes? It's much easier to do them one at a time and then use the matrix functionality to combine them. Multiplying two matrices together creates a new matrix that contains the transforms of both of them combined. The order that you multiply them makes a difference because the changes are applied in order. You can use the **TransformBy** method to do this. This takes the current matrix and multiplies another matrix with it. Conceptually, you have a matrix and transform it by the change defined by another matrix. Let's look at it in practice to solve the problem presented above.

```
Dim dPi As Double
dPi = Atn(1) * 4

Dim oTG As TransientGeometry
Set oTG = ThisApplication.TransientGeometry

' Create a matrix that defines a  $45^\circ$  rotation around the x-axis.
Dim oTransMatrix As Matrix
Set oTransMatrix = oTG.CreateMatrix
Call oTransMatrix.SetToRotation(dPi / 4, oTG.CreateVector(1, 0, 0), _
                               oTG.CreatePoint(0, 0, 0))

' Create a matrix that defines a  $30^\circ$  rotation around the y-axis and apply it.
Dim oTempMatrix As Matrix
Set oTempMatrix = oTG.CreateMatrix
Call oTempMatrix.SetToRotation(dPi / 6, oTG.CreateVector(0, 1, 0), _
                               oTG.CreatePoint(0, 0, 0))
Call oTransMatrix.TransformBy(oTempMatrix)
```

```

' Create a matrix that defines a 5 cm X and 3 cm Z move and applies it.
oTempMatrix.SetToIdentity
Call oTempMatrix.SetTranslation(oTG.CreateVector(5, 0, 3))
Call oTransMatrix.TransformBy(oTempMatrix)

' Iterate through the occurrences in the assembly.
Dim oOcc As ComponentOccurrence
For Each oOcc In oAsmDoc.ComponentDefinition.Occurrences
    ' Get the matrix from the current occurrence.
    Dim oMatrix As Matrix
    Set oMatrix = oOcc.Transformation

    ' Apply the transform to the matrix.
    Call oMatrix.TransformBy(oTransMatrix)

    ' Set the transformation of the occurrence using the modified matrix.
    oOcc.Transformation = oMatrix
Next

```

This example is not much different from what we've already looked at except it's applying each matrix to another matrix to build up a series of transforms. It also takes advantage of some other functions supported by the Matrix object to make using it easier. For example, the rotations are defined using the **SetToRotation** method and the **SetToIdentity** method is used to set the matrix to an identity matrix.

In keeping with the idea that a matrix can define a transformation, one frequent use of a matrix in Inventor is to define the transformation from one coordinate system to another. This is another way to think of what's happening when you insert a part into an assembly using a matrix. The part geometry is defined in the part with respect to the part coordinate system. The matrix that defines the position of the part in the assembly is defining the transform that takes place going from part space to assembly space. Here's a simple example of to illustrate.

```

Public Sub ModelPointInAssembly()
    Dim oAsmDoc As AssemblyDocument
    Set oAsmDoc = ThisApplication.ActiveDocument

    ' Get an occurrence in the assembly. This arbitrarily gets the
    ' first one. To make the result interesting, this occurrence
    ' should be moved and rotated from its original position.
    Dim oOcc As ComponentOccurrence
    Set oOcc = oAsmDoc.ComponentDefinition.Occurrences.Item(1)

    ' Create a point object that has (0,0,0) as its coordinates.
    ' This will represent the part space origin point.
    Dim oPartOrigin As Point
    Set oPartOrigin = ThisApplication.TransientGeometry.CreatePoint(0, 0, 0)

    ' Get the matrix from the occurrence. This defines the position
    ' of the part in the assembly. It also defines the transform
    ' from part space into assembly space.
    Dim oTransMatrix As Matrix
    Set oTransMatrix = oOcc.Transformation

    ' Transform the point using this matrix. This essentially transforms
    ' it from part space to assembly space.
    Call oPartOrigin.TransformBy(oTransMatrix)

    ' Create a work point at the point location to see where it is.
    Call oAsmDoc.ComponentDefinition.WorkPoints.AddFixed(oPartOrigin)
End Sub

```

The matrix returned by the ComponentOccurrence.Transformation property returns a matrix that defines the transform from part to assembly space. If you have a point in assembly space and need the equivalent in part space you can also do that. The Matrix object supports the **Invert** method. This will modify the matrix so it does the reverse action. If you invert the matrix returned by the ComponentOccurrence object it will define the transformation from assembly space to part space. Inventor's API provides matrices that define the transform from part to assembly space, model (part or assembly) space to drawing view space, model to sheet space, drawing view space to model space, sheet space to model space, drawing view space to sheet space, sheet space to drawing view space, sketch space to model space and model space to sketch space. Matrices are also used to report the position of pattern elements, and to define the position of client graphics, sketched symbols, texture maps, and title blocks.