Geometric Progression: Further Analysis of Geometry using the Autodesk Revit 2012 API

Scott Conover - Autodesk

In Revit 2012, we introduced new powerful tools to the API for geometry analysis, calculation and display. In this lecture, you will get an introduction to the new API toolset and see examples and recommendations on their use. One set of new APIs offers the ability to create three-dimensional construction geometry. Others include new analysis tools such as Boolean operations, extrusion analyzers, and room and space geometry calculators targeted towards specific kinds of problems and calculations. This class will also highlight some new geometric capabilities related to specific entity types such as construction parts, walls, point clouds, and energy analysis.

Learning Objectives

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

- Extract and analyze the geometry of existing Revit elements
- Create and manipulate temporary curve and solid geometry
- Find elements by 3D intersection
- Apply an ExtrusionAnalyzer to geometry
- Utilize parts to analyze geometry of HostObjects and their layers
- Extract and analyze the boundary geometry of rooms and spaces
- Analyze the geometry of point clouds

About the Speaker

Scott is a Software Development Manager for Autodesk, leading the effort to expand the Autodesk® Revit® API. For 12 years, he has used customization interfaces to parametric 3D CAD systems in order to automate repetitive tasks, extend the application user interface, and transfer data between the CAD application and different data formats. Since joining Autodesk, he has worked on the design, implementation, and testing of Revit APIs; including most recently, on the design of the 2012 geometry API tools and IFC open source effort. Scott holds a Master of Computer Systems Engineering degree from Northeastern University with a concentration on CAD/CAM/CAE.

Introduction

Last year at Autodesk University, I presented a course: “Analyzing Geometry of Buildings using the Autodesk Revit API”. In this course I covered the fundamentals of geometry extraction and specific toolsets like ray tracing, material quantity extraction, and use of transforms and coordinates.

In Revit 2012, Autodesk introduced new powerful tools to the API for geometry analysis, calculation and display. One set of new tools offers the ability to create three-dimensional construction geometry. Others include new analysis tools such as Boolean operations, extrusion analyzers, and room and space geometry calculators targeted towards specific kinds of problems and calculations. This handout also highlights some new geometric capabilities related to specific entity types such as construction parts, walls, point clouds, and energy analysis.

Wiki documentation

Most of the material presented today is available in the API developers guide hosted on the Autodesk wikihelp system. The wikihelp system allows Autodesk employees, partners, vendors and customers to contribute information, tips and techniques to the wider user community. For the Revit API developer, the Developer’s guide contains valuable overviews and useful code snippets covering all of the new Revit 2012 API features as well as most other Revit API capabilities. Please consider the Developer’s Guide when learning more about Revit programming, and consider uploading your own examples and tips when you find something that could benefit the wider development community. The section heading in this handout include links to the relevant wiki topics for each section.

Fundamentals

The first part of this handout deals with the extraction of element geometry. Although this was covered thoroughly in a previous course, there are some new utilities and capabilities in Revit 2012 to be highlighted. This information provided also should help with concepts introduced later in the specific toolsets.

Extraction of Element geometry

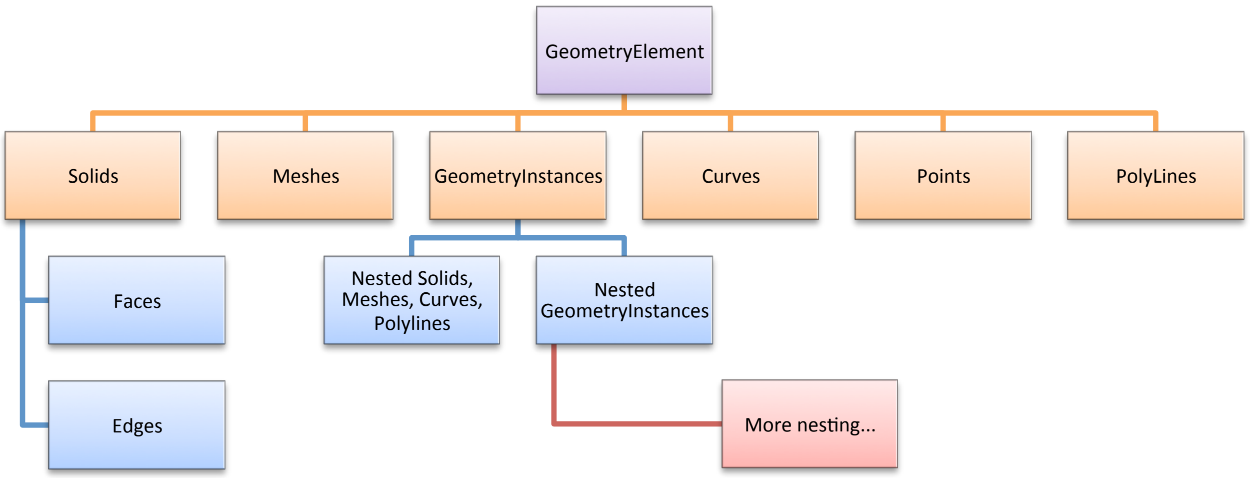
The indexed property Element.Geometry[] can be used to pull the geometry of any model element (3D element). This applies both to system family instances such as walls, floors and roofs, and also to family instances of many categories, e.g. doors, windows, furniture, or masses.

The extracted geometry is returned to you as Autodesk.Revit.DB.GeometryElement. You can look at the geometry members of that element by iterating the .Objects property.

Typically, the objects returned at the top level of the extracted geometry will be one of:

- Solid – a boundary representation made up of faces and edges
- Mesh – a 3D array of triangles
- Curve – a bounded 3D curve
- Point – a visible point datum at a given 3D location
- PolyLine – a series of line segments defined by 3D points
- GeometryInstance – an instance of a geometric element positioned within the element

This figure illustrates the hierarchy of objects found by geometry extraction.



Curves

A curve represents a path in 2 or 3 dimensions in the Revit model. Curves may represent the entire extent of an element's geometry (e.g. CurveElements) or may appear as a single piece of the geometry of an element (e.g. the centerline of a wall or duct). Curves and collections of curves are used as inputs in many element creation methods in the API.

Curve Parameterization

Curves in the Revit API can be described as mathematical functions of an input parameter "u", where the location of the curve at any given point in XYZ space is a function of "u".

Curves can be bound or unbound. Unbound curves have no endpoints, representing either an infinite abstraction (an unbound line) or a cyclic curve (a circle or ellipse).

In Revit, the parameter "u" can be represented in two ways:

- A 'normalized' parameter. The start value of the parameter is 0.0, and the end value is 1.0. For some curve types, this makes evaluation of the curve along its extents very easy, for example, the midpoint of a line is at parameter 0.5. (Note that for more complex curve equations like Splines this assumption cannot always be made).
- A 'raw' parameter. The start and end value of the parameter can be any value. For a given curve, the value of the minimum and maximum raw parameter can be obtained through Curve.get_EndParameter(int) (C#) or Curve.EndParameter(int) (VB.NET). Raw parameters are useful because their units are the same as the Revit default units (feet). So to obtain a location 5 feet along the curve from the start point, you can take the raw parameter at the start and add 5 to it. Raw parameters are also the only way to evaluate unbound curves.

The methods Curve.ComputeNormalizedParameter() and Curve.ComputeRawParameter() automatically scale between the two parameter types. The method Curve.IsInside() evaluates a raw parameter to see if it lies within the bounds of the curve.

You can use the parameter to evaluate a variety of properties of the curve at any given location:

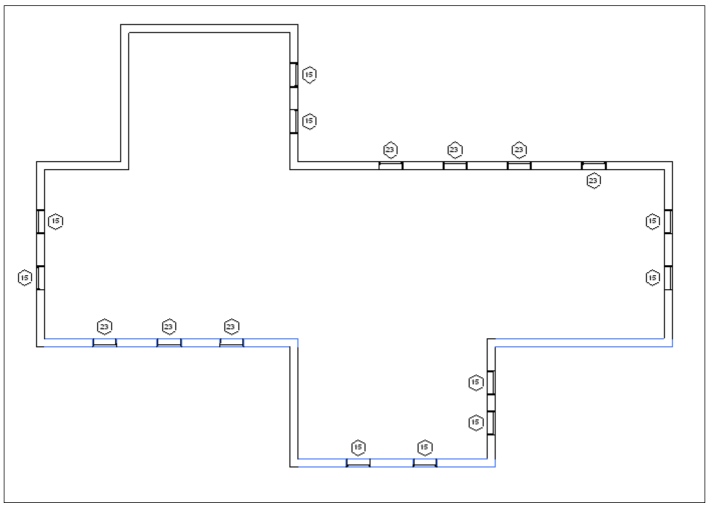
- The XYZ location of the given curve. This is returned from Curve.Evaluate(). Either the raw or normalized parameter can be supplied. If you are also calling

ComputeDerivatives(), this is also the .Origin property of the Transform returned by that method.

- The first derivative/tangent vector of the given curve. This is the .BasisX property of the Transform returned by Curve.ComputeDerivatives().
- The second derivative/normal vector of the given curve. This is the .BasisY property of the Transform returned by Curve.ComputeDerivatives().
- The *binormal* vector of the given curve, defined as the cross-product of the tangent and normal vector. This is the .BasisZ property of the Transform returned by Curve.ComputeDerivatives().

All of the vectors returned are non-normalized (but you can normalize any vector in the Revit API with XYZ.Normalize()). Note that there will be no value set for the normal and binormal vector when the curve is a straight line. You can calculate a normal vector to the straight line in a given plane using the tangent vector.

The API sample “DirectionCalculation” uses the tangent vector to the wall location curve to find exterior walls that face south.



Finding and highlighting south facing exterior walls

Curve types

Revit uses a variety of curve types to represent curve geometry in a document. These include:

Curve type	Revit API class	Definition	Notes
Bound line	Line	A line segment defined by its endpoints.	Obtain endpoints from Curve.get_Endpoint()
Unbound line	Line	An infinite line defined by a location and direction	Identify these with Curve.IsBound. Evaluate point and tangent vector at raw parameter= 0 to find the input parameters for the equation of the line.

Arc	Arc	A bound circular arc	Begin and end at a certain angle. These angles can be obtained by the raw parameter values at each end of the arc.
Circle	Arc	An unbound circle	Identify with Curve.IsBound. Use raw parameter for evaluation (from 0 to 2π)
Elliptical arc	Ellipse	A bound elliptical segment	
Ellipse	Ellipse	An unbound ellipse	Identify with Curve.IsBound. Use raw parameter for evaluation (from 0 to 2π)
NURBS	NurbSpline	A non-uniform rational B-spline	Used for splines sketched in various Revit tools, plus imported geometry
Hermite	HermiteSpline	A spline interpolate between a set of points	Used for tools like Curve by Points and flexible ducts/pipes, plus imported geometry

Mathematical representations of all of the Revit curve types can be found on the wiki.

Curve analysis and processing

There are several Curve members which are tools suitable for use in geometric analysis. In some cases, these APIs do more than you might expect by a quick review of their names.

Intersect()

The Intersect method allows you compare two curves to find how they differ or how they are similar. It can be used in the manner you might expect, to obtain the point or point(s) where two curves intersect one another, but it can also be used to identify:

- Collinear lines
- Overlapping lines
- Identical curves
- Totally distinct curves with no intersections

The return value identifies these different results, and the output IntersectionSetResult contains information on the intersection point(s).

Project()

The Project method projects a point onto the curve and returns information about the nearest point on the curve, its parameter, and the distance from the projection point.

Tessellate()
This splits the curve into a series of linear segments, accurate within a default tolerance. For Curve.Tessellate(), the tolerance is slightly larger than 1/16". This tolerance of approximation is the tolerance used internally by Revit as adequate for display purposes.

Note that only lines may be split into output of only two tessellation points; non-linear curves will always output more than two points even if the curve has an extremely large radius which might mathematically equate to a straight line.

Curve creation
Curves are often needed as inputs to Revit API methods. Curves can be created in several ways:

- Factory methods on Autodesk.Revit.Creation.Application
  - NewLineBound()
  - NewLineUnbound()
  - NewLine()
  - NewArc()
  - NewEllipse()
  - NewNurbSpline()
  - NewHermiteSpline()
- Members of the Curve class
  - Curve.Clone()
  - Curve.Transformed property

Collections of curves
The Revit API uses different types of collections of curves as inputs:

- CurveLoop – this represents a specific chain of curves joined end-to-end. It can represent a closed loop or an open one. Create it using:
  - CurveLoop.Create()
  - CurveLoop.CreateViaCopy()
  - CurveLoop.CreateViaThicken()
- CurveArray – this collection class represents an arbitrary collection of curves. Create it using its constructor.
- CurveArrArray – this collection class is a collection of CurveArrays. When this is used, the organization of the sub-elements of this array have meaning for the method this is passed to; for example, in NewExtrusion() multiple CurveArrays should represent different closed loops.

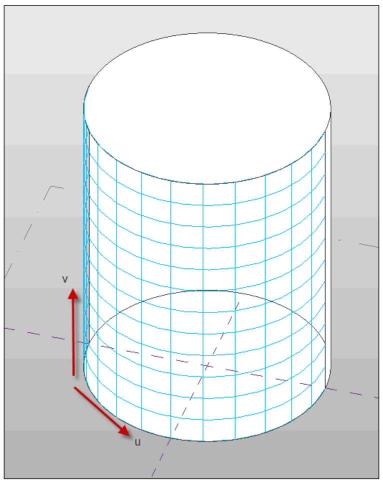
Newer API methods use .NET collections of Curves in place of CurveArray and CurveArrArray.

Solids, Faces and Edges
A Solid is a Revit API object which represents a collection of faces and edges. Typically in Revit these collections are fully enclosed volumes, but a shell or partially bounded volume can also be encountered. Note that sometimes the Revit geometry will contain unused solids containing zero edges and faces. Check the Edges and Faces members to filter out these solids.

The Revit API offers the ability to read the collections of faces and edges, and also to compute the surface area, volume, and centroid of the solid.

Faces

Faces in the Revit API can be described as mathematical functions of two input parameters “u” and “v”, where the location of the face at any given point in XYZ space is a function of the parameters. The U and V directions are automatically determined based on the shape of the given face. Lines of constant U or V can be represented as gridlines on the face, as shown in the example below:



U and V gridlines on a cylindrical face

You can use the UV parameters to evaluate a variety of properties of the face at any given location:

- Whether the parameter is within the boundaries of the face, using Face.IsInside()
- The XYZ location of the given face at the specified UV parameter value. This is returned from Face.Evaluate(). If you are also calling ComputeDerivatives(), this is also the .Origin property of the Transform returned by that method.
- The tangent vector of the given face in the U direction. This is the .BasisX property of the Transform returned by Face.ComputeDerivatives()
- The tangent vector of the given face in the V direction. This is the .BasisY property of the Transform returned by Face.ComputeDerivatives().
- The normal vector of the given face. This is the .BasisZ property of the Transform returned by Face.ComputeDerivatives().

All of the vectors returned are non-normalized.

Edge and face parameterization

Edges are boundary curves for a given face.

Iterate the edges of a Face using the EdgeLoops property. Each loop represents one closed boundary on the face. Edges are always parameterized from 0 to 1. It is possible to extract the Curve representation of an Edge with the Edge.AsCurve() and Edge.AsCurveFollowingFace() functions.

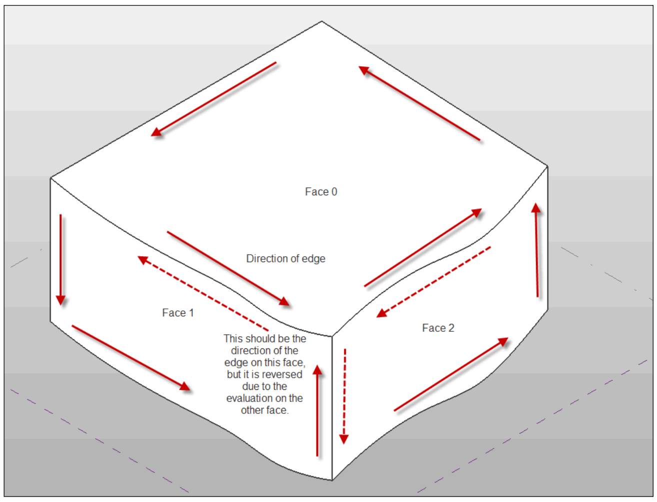
An edge is usually defined by computing intersection of two faces. But Revit doesn’t recompute this intersection when it draws graphics. So the edge stores a list of points - end points for a straight edge and a tessellated list for a curved edge. The points are parametric coordinates on the two faces. These points are available through the TessellateOnFace() method.

Sections produce “cut edges”. These are artificial edges - not representing a part of the model-level geometry, and thus do not provide a Reference.

Edge direction

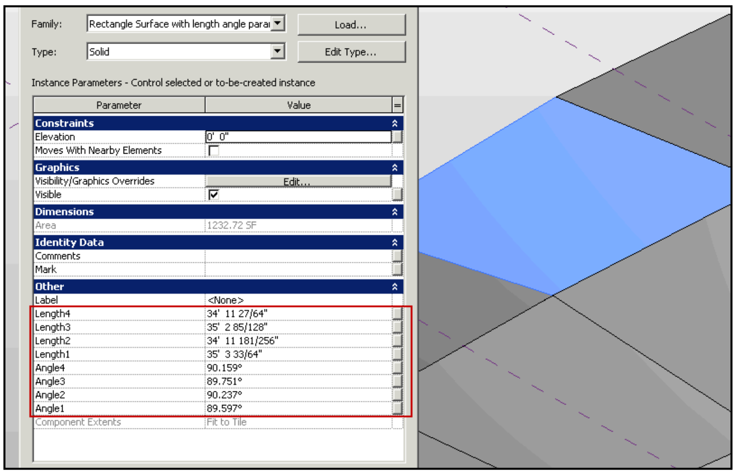
Direction is normally clockwise on the first face (first representing an arbitrary face which Revit has identified for a particular edge). But because two different faces meet at one particular edge, and the edge has the same parametric direction regardless of which face you are concerned with, sometimes you need to figure out the direction of the edge on a particular face.

The figure below illustrated how this works. For Face 0, the edges are all parameterized clockwise. For Face 1, the edge shared with Face 0 is not re-parameterized; thus with respect to Face 1 the edge has a reversed direction, and some edges intersect where both edges’ parameters are 0 (or 1).


Face 0
Direction of edge
Face 1
Face 2
This should be the direction of the edge on this face, but it is reversed due to the evaluation on the other face.

Edge parameterization

The API sample “PanelEdgeLengthAngle” shows how to recognize edges that are reversed for a given face. It uses the tangent vector at the edge endpoints to calculate the angle between adjacent edges, and detect whether or not to flip the tangent vector at each intersection to calculate the proper angle.


Family: Rectangle Surface with length angle parai
Type: Solid

Instance Parameters - Control selected or to-be-created instance

Parameter	Value
Constraints	
Elevation	0' 0"
Moves With Nearby Elements	☐
Graphics	
Visibility/Graphics Overrides	Edit...
Visible	☑
Dimensions	
Area	1232.72 SF
Identity Data	
Comments	
Mark	
Other	
Label	<None>
Length4	34' 11 27/64"
Length3	35' 2 85/128"
Length2	34' 11 181/256"
Length1	35' 3 33/64"
Angle4	90.159°
Angle3	89.751°
Angle2	90.237°
Angle1	89.597°
Component Extents	Fit to Tile

PanelEdgeLengthAngle results

Face types

Revit uses a variety of curve types to represent face geometry in a document. These include:

Face type	Revit API class	Definition	Notes
Plane	PlanarFace	A plane defined by the origin and unit vectors in U and V.	
Cylinder	CylindricalFace	A face defined by extruding a circle along an axis.	.Radius provides the "radius vectors" – the unit vectors of the circle multiplied by the radius value.
Cone	ConicalFace	A face defined by rotation of a line about an axis.	.Radius provides the "radius vectors" – the unit vectors of the circle multiplied by the radius value.
Revolved face	RevolvedFace	A face defined by rotation of an arbitrary curve about an axis.	.Radius provides the unit vectors of the plane of rotation, there is no "radius" involved.
Ruled surface	RuledFace	A face defined by sweeping a line between two profile curves, or one profile curve and one point.	Both curve(s) and point(s) can be obtained as properties.
Hermite face	HermiteFace	A face defined by Hermite interpolation between points.	

Mathematical representations of all of the Revit face types can be found on the wiki.

Face analysis and processing

There are several Face members which are tools suitable for use in geometric analysis.

Intersect()

The Intersect method computes the intersection between the face and a curve. It can be used in to identify:

- The intersection point(s) between the two objects
- The edge nearest the intersection point, if there is an edge close to this location
- Curves totally coincident with a face
- Curves and faces which do not intersect

Project()

The Project method projects a point on the input face, and returns information on the projection point, the distance to the face, and the nearest edge to the projection point.

Triangulate()
The Triangulate method obtains a triangular mesh approximating the face. Similar to Curve.Tessellate(), this mesh's points are accurate within the input tolerance used by Revit (slightly larger than 1/16").

Solid and face creation
Solids and faces are sometimes used as inputs to other utilities. The Revit API provides several routines which can be used to create such geometry from scratch or to derive it from other inputs.

Transformed geometry
The method

- GeometryElement.GetTransformed()

returns a copy of the input geometry element with a transformation applied. Because this geometry is a copy, its members cannot be used as input references to other Revit elements, but it can be used geometric analysis and extraction.

Geometry creation utilities
The GeometryCreationUtilities class is a utility class that allows construction of basic solid shapes:

- Extrusion
- Revolution
- Sweep
- Blend
- SweptBlend

The resulting geometry is not added to the document as a part of any element. However, the created Solid can be used as inputs to other API functions, including:

- As the input face(s) to the methods in the Analysis Visualization framework (SpatialFieldManager.AddSpatialFieldPrimitive()) – this allows the user to visualize the created shape relative to other elements in the document
- As the input solid to finding 3D elements by intersection
- As one or more of the inputs to a Boolean operation
- As a part of a geometric calculation (using, for example, Face.Project(), Face.Intersect(), or other Face, Solid, and Edge geometry methods)

The following example uses the GeometryCreationUtilities class to create cylindrical shapes based on a location and height. This might be used, for example, to create volumes around the ends of a wall in order to find other walls within close proximity to the wall end points:

```
private Solid CreateCylindricalVolume(XYZ point, double height, double radius)
{
    // build cylindrical shape around endpoint
    List<CurveLoop>  curveloops = new List<CurveLoop>();
    CurveLoop circle = new CurveLoop();

     // For solid geometry creation, two curves are necessary, even for closed
    // cyclic shapes like circles
    circle.Append(m_app.Create.NewArc(point, radius, 0, Math.PI, XYZ.BasisX,
XYZ.BasisY));
    circle.Append(m_app.Create.NewArc(point, radius, Math.PI, 2 * Math.PI, XYZ.BasisX,
XYZ.BasisY));
```

```
    curveloops.Add(circle);


    Solid createdCylinder =
GeometryCreationUtilities.CreateExtrusionGeometry(curveloops, XYZ.BasisZ, height);

    return createdCylinder;
}
```

Boolean operations

The BooleanOperationsUtils class provides methods for combining a pair of solid geometry objects.

The ExecuteBooleanOperation() method takes a copy of the input solids and produces a new solid as a result. Its first argument can be any solid, either obtained directly from a Revit element or created via another operation like GeometryCreationUtils.

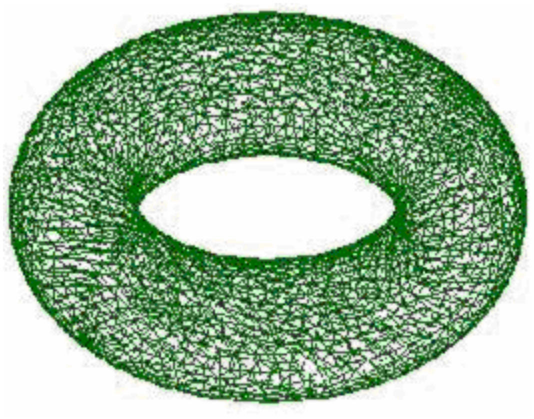
The method ExecuteBooleanOperationModifyingOriginalSolid() performs the boolean operation directly on the first input solid. The first input must be a solid which is not obtained directly from a Revit element. The property GeometryObject.IsElementGeometry can identify whether the solid is appropriate as input for this method.

Options to both methods include the operations type: Union, Difference, or Intersect. The following example demonstrates how to get the intersection of two solids and then find the volume.

```
private void ComputeIntersectionVolume(Solid solidA, Solid solidB)
{
  Solid intersection = BooleanOperationsUtils
    ExecuteBooleanOperation(solidA, solidB,
      BooleanOperationsType.Intersect);

  double volumeOfIntersection = intersection.Volume;
}
```

Meshes

A mesh is a collection of triangular boundaries which collectively forms a 3D shape. Meshes are typically encountered inside Revit element geometry if those elements were created from certain import operations or inside topography surfaces.   You can also obtain Meshes as the result of calls to Face.Triangulate() for any given Revit face.



A mesh representing a torus

The following code sample illustrates how to get the geometry of a Revit face as a Mesh:

Extracting the geometry of a mesh

```
private void GetTrianglesFromFace(Face face)
{
        // Get mesh
        Mesh mesh = face.Triangulate();
        for (int i = 0; i < mesh.NumTriangles; i++)
        {
                MeshTriangle triangle = mesh.get_Triangle(i);
                XYZ vertex1 = triangle.get_Vertex(0);
                XYZ vertex2 = triangle.get_Vertex(1);
                XYZ vertex3 = triangle.get_Vertex(2);
        }
}
```

Note that the approximation tolerance used for Revit display purposes is always used by the Triangulate() method when constructing the Mesh.

PolyLines

A polyline is a collection of line segments defined by a set of coordinate points. These are typically encountered in imported geometry. The PolyLine class offers the ability to read the coordinates:

- PolyLine.NumberOfCoordinates – the number of points in the polyline
- PolyLine.GetCoordinate() – gets a coordinate by index
- PolyLine.GetCoordinates() – gets a collection of all coordinates in the polyline
- PolyLine.Evaluate() – given a normalized parameter (from 0 to 1) evaluates an XYZ point along the extents of the entire polyline

Points

A point represents a visible coordinate in 3D space. These are typically encountered in mass family elements like ReferencePoint. The Point class provides read access to its coordinates, and an ability to obtain a reference to the point for use as input to other functions.

GeometryInstances

A GeometryInstance represents a set of geometry stored by Revit in a default configuration, and then transformed into the proper location as a result of the properties of the element. The most common situation where GeometryInstances are encountered is in Family instances. Revit uses GeometryInstances to allow it to store a single copy of the geometry for a given family and reuse it in multiple instances.

Note that not all Family instances will include GeometryInstances. When Revit needs to make a unique copy of the family geometry for a given instance (because of the effect of local joins, intersections, and other factors related to the instance placement) no GeometryInstance will be encountered; instead the Solid geometry will be found at the top level of the hierarchy.

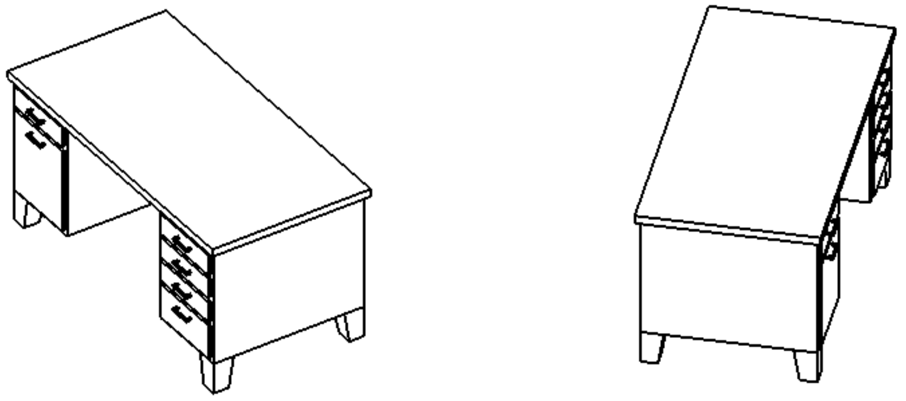
A GeometryInstance offers the ability to read its geometry through the GetSymbolGeometry() and GetInstanceGeometry() methods. These methods return another Autodesk.Revit.DB.GeometryElement which can be parsed just like the first level return.

GetSymbolGeometry() returns the geometry represented in the coordinate system of the family. Use this, for example, when you want a picture of the “generic” table without regards to the orientation and placement location within the project. This is also the only overload which returns the actual Revit geometry objects to you, and not copies. This is important because operations which use this geometry as input to creating other elements (for example, dimensioning or placement of face-based families) require the reference to the original geometry.

GetInstanceGeometry() returns the geometry represented in the coordinate system of the project where the instance is placed. Use this, for example, when you want a picture of the specific geometry of the instance in the project (for example, ensuring that tables are placed parallel to the walls of the room). This always returns a copy of the element geometry, so while it would be suitable for implementation of an exporter or a geometric analysis tool, it would not be appropriate to use this for the creation of other Revit elements referencing this geometry.

There are also overloads for both GetInstanceGeometry() and GetSymbolGeometry() that transform the geometry by any arbitrary coordinate system. These methods always return copies similar to GetInstanceGeometry().

The GeometryInstance also stored a transformation from the symbol coordinate space to the instance coordinates. This transform is accessible as the Transform property. It is also the transformation used when extracting a the copy of the geometry via GetInstanceGeometry().

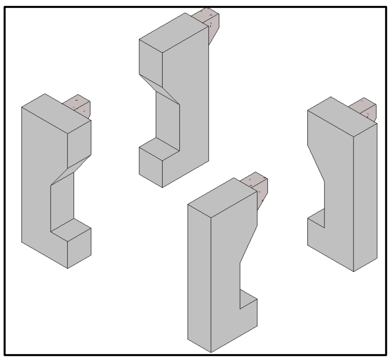


The same symbol geometry placed at a different instance transform

Instances may be nested several levels deep for some families. If you encounter nested instances they may be parsed in a similar manner as the first level instance.

Example: Placing corbel on correct face

This example demonstrates how to parse the geometry of family instances and correctly use the members of GeometryInstance to obtain a useful reference for placement of a different face-based family. Because a reference is needed, the symbol geometry has to be used, and transforms applied where necessary.



Geometry Analysis Tools

The following tools were discussed in the previous course and are described in detail on wiki:

- Finding geometry by ray projection
- Material quantity takeoff
- Making temporary changes for geometric calculations
- Bounding box filters

The following tools are described in this course and are new in 2012:

- Temporary solid geometry – creation of Solid geometry was described above
- Boolean operations – use of Boolean operations to combine Solid geometry was described above
- Element intersection filters
- Extrusion analysis
- Element.GetGeneratingElementIds
- HostObjects
- Parts
- Room & space geometry
- Energy analytical model
- Point cloud analysis

Element intersection filters

The element filters:

- ElementIntersectsElementFilter
- ElementIntersectsSolidFilter

pass elements whose actual 3D geometry intersects the 3D geometry of the target object.

With ElementIntersectsElementFilter, the target object is another element. The intersection is determined with the same logic used by Revit to determine if an interference exists during generation of an Interference Report. (This means that some combinations of elements will never pass this filter, such as concrete members which are automatically joined at their intersections). Also, elements which have no solid geometry, such as Rebar, will not pass this filter.

With ElementIntersectsSolidFilter, the target object is any solid. This solid could have been obtained from an existing element, created from scratch using the routines in GeometryCreationUtilities, or the result of a secondary operation such as a Boolean operation. Similar to the ElementIntersectsElementFilter, this filter will not pass elements which lack solid geometry.

Both filters can be inverted to match elements outside the target object volume.

Both filters are slow filters, and thus are best combined with one or more quick filters such as class or category filters.

In this example an input solid is constructed to represent the required clearance for the approach to a door (as per the Americans with Disabilities Act), and an ElementIntersectsSolidFilter finds physical elements which intersect the volume.

```
/// <summary>
/// Finds any Revit physical elements which interfere with the target solid region surrounding a
door.
/// </summary>
/// <remarks>This routine is useful for detecting interferences which are violations of the
Americans with
/// Disabilities Act or other local disabled access codes.</remarks>
/// <param name="doorInstance">The door instance.</param>
/// <param name="doorAccessibilityRegion">The accessibility region calculated to surround the
approach of the door.</param>
/// <returns>A collection of interfering element ids.</returns>
private ICollection<ElementId> FindElementsInterferingWithDoor(FamilyInstance doorInstance, Solid
doorAccessibilityRegion)
{
    // Setup the filtered element collector for all document elements.
    FilteredElementCollector interferingCollector
= new FilteredElementCollector(doorInstance.Document);

    // Only accept element instances
    interferingCollector.WhereElementIsNotElementType();

    // Exclude intersections with the door itself or the host wall for the door.
    List<ElementId> excludedElements = new List<ElementId>();
    excludedElements.Add(doorInstance.Id);
    excludedElements.Add(doorInstance.Host.Id);
    ExclusionFilter exclusionFilter = new ExclusionFilter(excludedElements);
    interferingCollector.WherePasses(exclusionFilter);

    // Set up a filter which matches elements whose solid geometry intersects with the accessibility
region
    ElementIntersectsSolidFilter intersectionFilter
= new ElementIntersectsSolidFilter(doorAccessibilityRegion);
    interferingCollector.WherePasses(intersectionFilter);

    // Return all elements passing the collector
    return interferingCollector.ToElementIds();
}
```

Extrusion analysis

The utility class ExtrusionAnalyzer allows you to attempt to “fit” a given piece of geometry into the shape of an extruded profile. An instance of this class is a single-time use class which should be supplied a solid geometry, a plane, and a direction. After the ExtrusionAnalyzer has been initialized, you can access the results through the following members:

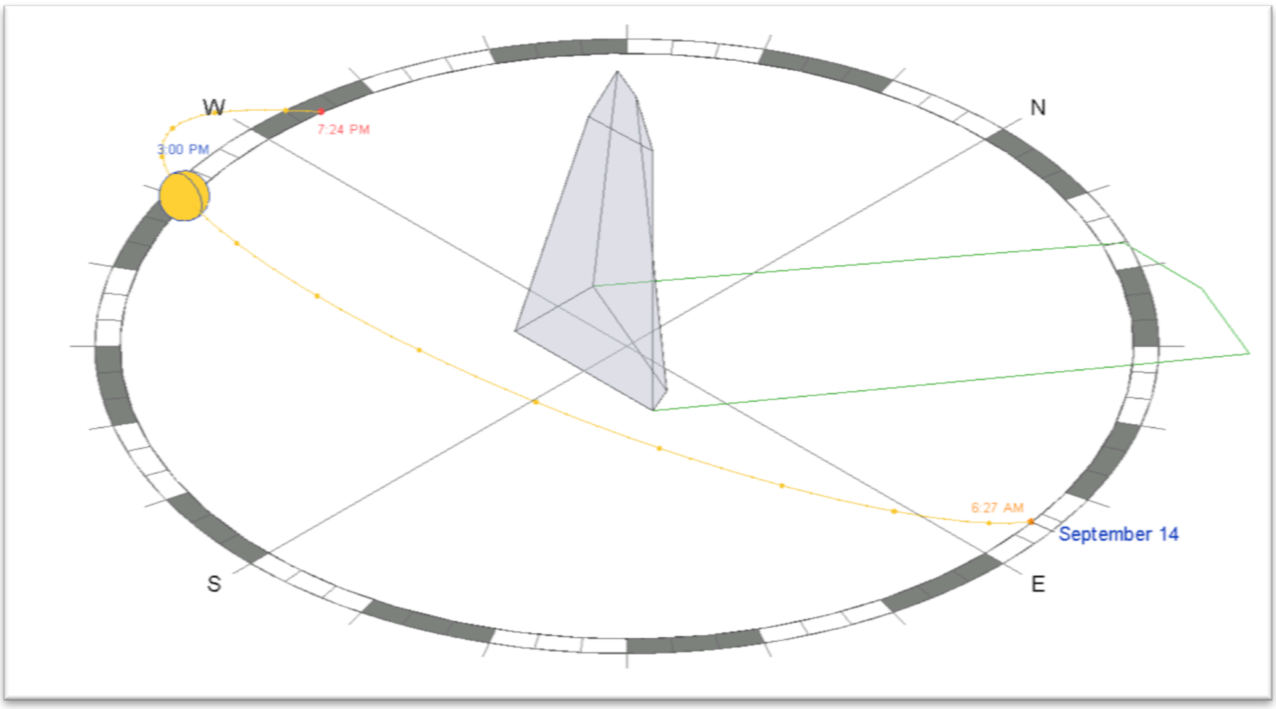
- The GetExtrusionBase() method returns the calculated base profile of the extruded solid aligned to the input plane.
- The CalculateFaceAlignment() method can be used to identify all faces from the original geometry which do and do not align with the faces of the calculated extrusion. This could be useful to figure out if a wall, for example, has a slanted join at the top as would be the case if there is a join with a roof. If a face is unaligned, something is joined to the geometry that is affecting it.
- To determine the element that produced the non-aligned face, pass the face to Element.GetGeneratingElementIds(). For more details on this utility, see the following section.

The ExtrusionAnalyzer utility works best for geometry which are at least somewhat “extrusion-like”, for example, the geometry of a wall which may or may not be affected by end joins, floor joins, roof joins, openings cut by windows and doors, or other modifications. Rarely for specific shape and directional combinations the analyzer may be unable to determine a coherent face to act as the base of the extrusion – an InvalidOperationException will be thrown in these situations.

In this example, the extrusion analyzer is used to calculate and outline a shadow formed from the input solid and the sun direction:

```
        /// <summary>
        /// Draw the shadow of the indicated solid with the sun direction specified.
        /// </summary>
        /// <remarks>The shadow will be outlined with model curves added to the document.
        /// A transaction must be open in the document.</remarks>
        /// <param name="document">The document.</param>
        /// <param name="solid">The target solid.</param>
        /// <param name="targetLevel">The target level where to measure and draw the shadow.</param>
        /// <param name="sunDirection">The direction from the sun (or light source).</param>
        /// <returns>The curves created for the shadow.</returns>
        /// <throws cref="Autodesk.Revit.Exceptions.InvalidOperationException">Thrown by
ExtrusionAnalyzer when the geometry and
        /// direction combined do not permit a successful analysis.</throws>
        private static ICollection<ElementId> DrawShadow(Document document, Solid solid, Level
targetLevel, XYZ sunDirection)
        {
            // Create target plane from level.
            Plane plane =
document.Application.Create.NewPlane(XYZ.BasisZ, new XYZ(0, 0, targetLevel.ProjectElevation));

            // Create extrusion analyzer.
            ExtrusionAnalyzer analyzer = ExtrusionAnalyzer.Create(solid, plane, sunDirection);

            // Get the resulting face at the base of the calculated extrusion.
            Face result = analyzer.GetExtrusionBase();

            // Convert edges of the face to curves.
            CurveArray curves = document.Application.Create.NewCurveArray();
            foreach (EdgeArray edgeLoop in result.EdgeLoops)
            {
                foreach (Edge edge in edgeLoop)
                {
                    curves.Append(edge.AsCurve());
                }
            }

            // Get the model curve factory object.
            Autodesk.Revit.Creation.ItemFactoryBase itemFactory;
            if (document.IsFamilyDocument)
                itemFactory = document.FamilyCreate;
            else
                itemFactory = document.Create;

            // Add a sketch plane for the curves.
            SketchPlane sketchPlane =
itemFactory.NewSketchPlane(document.Application.Create.NewPlane(curves));
            document.Regenerate();

            // Add the shadow curves
            ModelCurveArray curveElements = itemFactory.NewModelCurveArray(curves, sketchPlane);

            // Return the ids of the curves created
            List<ElementId> curveElementIds = new List<ElementId>();
            foreach (ModelCurve curveElement in curveElements)
            {
                curveElementIds.Add(curveElement.Id);
            }

            return curveElementIds;
        }

/// <summary>
/// Draw the shadow of the indicated solid with the sun direction specified.
/// </summary>
/// <remarks>The shadow will be outlined with model curves added to the document.
/// A transaction must be open in the document.</remarks>
/// <param name="document">The document.</param>
/// <param name="solid">The target solid.</param>
/// <param name="targetLevel">The target level where to measure and draw the shadow.</param>
/// <param name="sunDirection">The direction from the sun (or light source).</param>
/// <returns>The curves created for the shadow.</returns>
/// <throws cref="Autodesk.Revit.Exceptions.InvalidOperationException">Thrown by ExtrusionAnalyzer
when the geometry and
/// direction combined do not permit a successful analysis.</throws>
private static ICollection<ElementId> DrawShadow(Document document, Solid solid, Level
targetLevel, XYZ sunDirection)
{
    // Create target plane from level.
    Plane plane =
document.Application.Create.NewPlane(XYZ.BasisZ, new XYZ(0, 0, targetLevel.ProjectElevation));

    // Create extrusion analyzer.
    ExtrusionAnalyzer analyzer = ExtrusionAnalyzer.Create(solid, plane, sunDirection);

    // Get the resulting face at the base of the calculated extrusion.
    Face result = analyzer.GetExtrusionBase();

    // Convert edges of the face to curves.
    CurveArray curves = document.Application.Create.NewCurveArray();
    foreach (EdgeArray edgeLoop in result.EdgeLoops)
    {
        foreach (Edge edge in edgeLoop)
        {
            curves.Append(edge.AsCurve());
        }
    }

    // Get the model curve factory object.
    Autodesk.Revit.Creation.ItemFactoryBase itemFactory;
    if (document.IsFamilyDocument)
        itemFactory = document.FamilyCreate;
    else
        itemFactory = document.Create;

    // Add a sketch plane for the curves.
    SketchPlane sketchPlane =
itemFactory.NewSketchPlane(document.Application.Create.NewPlane(curves));
    document.Regenerate();

    // Add the shadow curves
    ModelCurveArray curveElements = itemFactory.NewModelCurveArray(curves, sketchPlane);

    // Return the ids of the curves created
    List<ElementId> curveElementIds = new List<ElementId>();
    foreach (ModelCurve curveElement in curveElements)
    {
        curveElementIds.Add(curveElement.Id);
    }

    return curveElementIds;
}
```

The utility above can be used to compute the shadow of a given mass with respect to the current sun and shadows settings for the view:

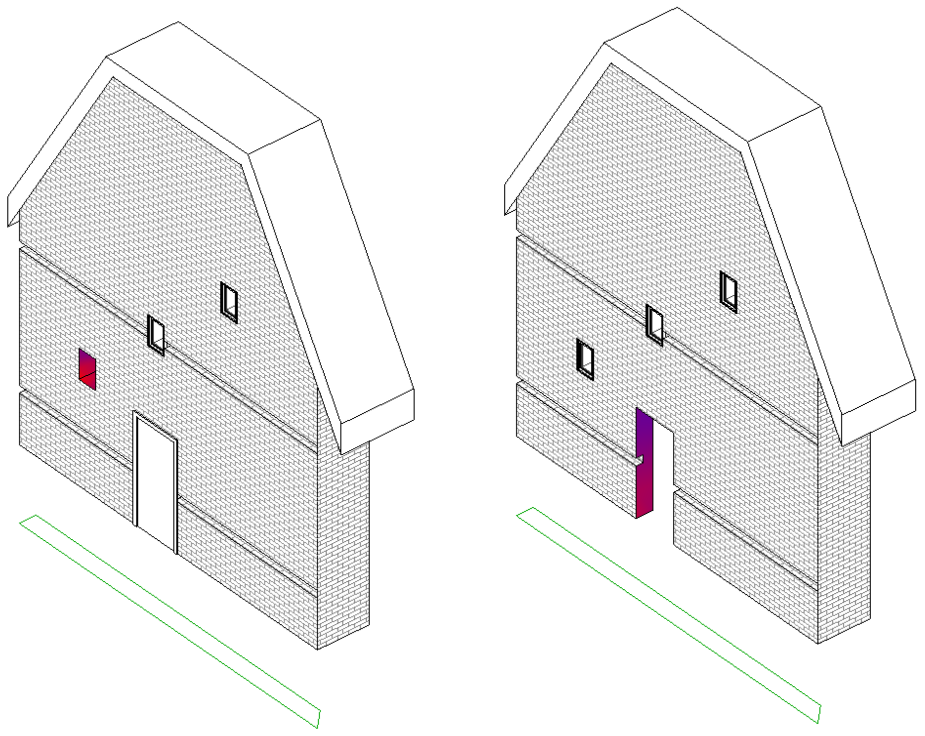


Element.GetGeneratingElementIds()

This function examines relationships among elements to find the element that generated a particular face. Most of these relationships will return a single element, for example:

- Window and door cutting walls
- Openings cutting hosts
- Face splitting faces
- Wall sweep or reveal traversing wall
- A few relationships have the potential for returning multiple elements, including:
  - Walls joining to other wall(s)
  - Elements extending to roof(s)
- If more than one id is returned, one of them (unspecified) is the id of the element that generated the geometry object and the others are ids of related elements. For example, if a wall A is joined to two walls B and C in a way that creates two end faces, and if this function is called for one of the two end faces, it will return the ids of walls B and C.

In this example, the ExtrusionAnalyzer is used in conjunction with Element.GetGeneratingElementIds() to find all faces that result from each element that cuts into the standard shape of an extruded wall.   The wall extruded base is displayed, and each element modifier highlighted sequentially.



CompoundStructure & HostObject utilities

Walls, floors, ceilings and roofs are all children of the API class HostObject. HostObject (and its related type class HostObjAttributes) provide read only to the CompoundStructure.

The CompoundStructure class offers read and write access to a set of layers consisting of different materials:

- CompoundStructure.GetLayers()
- CompoundStructure.SetLayers()

Normally these layers are parallel and extend the entire host object with a fixed layer width. However, for walls the structure can also be “vertically compound”, where the layers vary at specified vertical distances from the top and bottom of the wall. Use CompoundStructure.IsVerticallyCompound to identify these. For vertically compound structures, the structure describes a vertical section via a rectangle which is divided into polygonal regions whose sides are all vertical or horizontal segments. A map associates each of these regions with the index of a layer in the CompoundStructure which determines the properties of that region.

It is possible to use the compound structure to find the geometric location of different layer boundaries. The method CompoundStructure.GetOffsetForLocationLine() provides the offset from the center location line to any of the location line options (core centerline, finish faces on either side, or core sides).

With the offset to the location line available, you can obtain the location of each layer boundary by starting from a known location and obtaining the widths of each bounding layer using CompoundStructure.GetLayerWidth().

Some notes about the use of CompoundStructure:

- The total width of the element is the sum of each CompoundStructureLayer's widths. You cannot change the element's total width directly but you can change it via changing the

CompoundStructureLayer width. The index of the designated variable length layer (if assigned) can be obtained from CompoundStructure.VariableLayerIndex.

- You must set the CompoundStructure back to the HostObjAttributes instance (using the HostObjAttributes.SetCompoundStructure() method) in order for any change to be stored.
- Changes to the HostObjAttributes affects every instance in the current document. If you need a new combination of layers,you will need to create a new HostObjAttributes (use ElementType.Duplicate()) and assign the new CompoundStructure to it.
- The CompoundStructureLayer DeckProfileId, and DeckEmbeddingType, properties only work with Slab in Revit Structure. For more details, refer to Revit Structure.

The HostObjectUtils class offers methods as a shortcut to locate certain faces of compound HostObjects. These utilities retrieve the faces which act as the boundaries of the object's CompoundStructure:

- HostObjectUtils.GetSideFaces() – applicable to Walls and FaceWalls; you can obtain either the exterior or interior finish faces.
- HostObjectUtils.GetTopFaces() and HostObjectUtils.GetBottomFaces() – applicable to roofs, floors, and ceilings.

Parts

Part elements in Revit support the construction modeling process by letting you divide certain elements from the design intent model into discrete parts. These parts, and any smaller parts derived from them, can be independently scheduled, tagged, filtered, and exported. Parts can be used by construction modelers to plan delivery and installation of pieces of more complex Revit elements.

Parts can be generated from elements with layered structures, such as:

- Walls (excluding stacked walls and curtain walls)
- Floors (excluding shape-edited floors)
- Roofs (excluding those with ridge lines)
- Ceilings
- Structural slab foundations

Parts are dependent to elements. Parts are automatically updated and regenerated when the original element from which they are derived is modified. Such edits might include adding/removing layers, or changing wall type, layer thickness, wall orientation, geometry, materials, or openings.

Deleting the original element from which parts have been derived will delete all those parts as well as any parts derived from them.

Deleting a part will also delete all other parts derived from the original element.

Copying the original element from which parts have been derived will copy all the associated parts as well.

Parts in the Revit API

In the Revit API, elements can be divided into parts using the PartUtils class. The static method PartUtils.CreateParts() is used to create parts from one or more elements. Note that unlike most element creation methods in the API, CreateParts() does not actually create or return the parts, but rather instantiates an element called a PartMaker. The PartMaker uses its embedded rules to drive creation of the needed parts during regeneration.

The API also offers an interface to subdivide parts. PartUtils.DivideParts() accepts as input a collection of part ids, a collection of “intersecting element” ids (which can be layers and grids), and a collection of curves. The routine uses the intersecting elements and curves as boundaries from which to divide and generate new parts.

The GetAssociatedParts() method can be called to find some or all of the parts associated with an element, or use HasAssociatedParts() to determine if an element has parts.

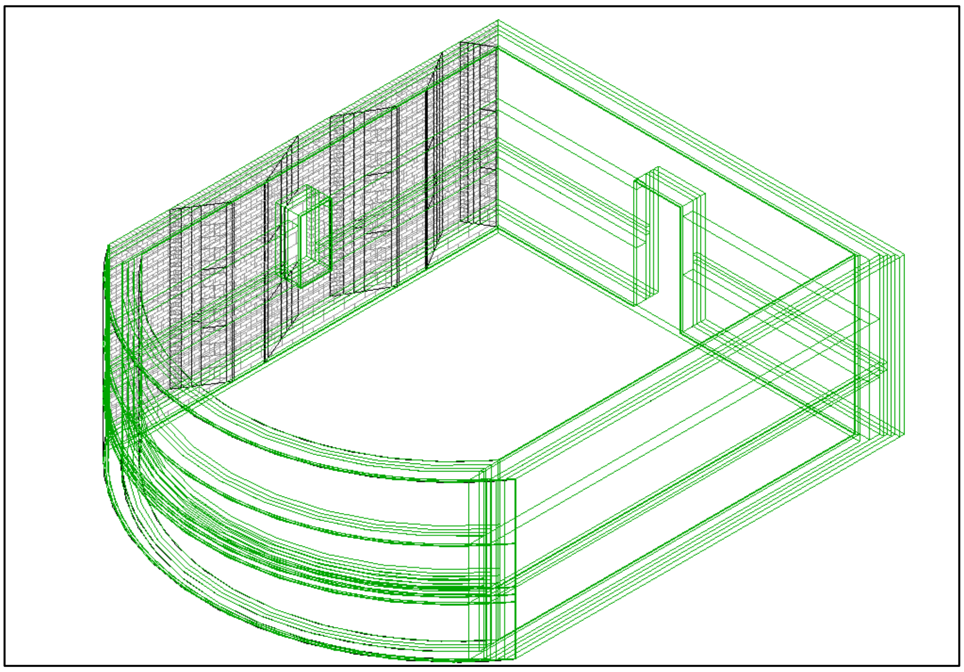
You can delete parts through the API either by deleting the individual part elements, or by deleting the PartMaker associated to the parts (which will delete all parts generated by this PartMaker after the next regeneration).

Parts can be manipulated in the Revit API much the same as they can in the Revit user interface. For example, the outer boundaries of parts may be offset with PartUtils.SetFaceOffset().

Parts as a geometric analysis tool

The capabilities that parts offer to the Revit API developer can be useful even when you don’t wish to store permanently the divisions as elements. Particularly, use of part creation provides easy access to the geometry of each of the layers of materials within a compound object. If you are looking for the boundaries, volumes, or intersections between layers, it is a natural approach to start with dividing the elements into parts. This is especially useful for vertically compound structures within a wall where the alternative would be to get the properties of the CompoundStructure and interpret in the context of the host wall’s height and extents. If you wrap the part creation into a transaction which is later cancelled, you can extract the geometry of these layers easily for any needed processing.

In the example “PartExtents”, the target objects are split into layer-based parts and the geometry of these parts’ edges are used to create model curves in place. If the target element has already been split into parts or sub-parts, the code deletes extra subdivisions so that the layer geometry can be obtained. All part-related changes are a part of a transaction which is rolled back before the resulting curves are created.



```
{
    public void DrawExtents(Document document, ICollection<Element> elements)
    {
        // Temp transaction    to extract geometry of each layer
        Transaction tempTransaction = new Transaction(document, "Temp geometry extraction");
        tempTransaction.Start();

        // For each target element, determine if it has already been divided or not
        List<ElementId> ids = new List<ElementId>();
        foreach (Element e in elements)
        {
            // If parts aleady exist, just delete the subparts so that we can get geometry from the
layer parts
            if (PartUtils.HasAssociatedParts(document, e.Id))
            {
                DeleteExistingSubParts(document, e.Id);
            }
            // No parts exist; save ths id for part creation
            else
            {
                ids.Add(e.Id);
            }
        }

        // Create parts from the top level objects which don't have parts
        PartUtils.CreateParts(document, ids);

        // Regenerate: required to create and delete parts based on changes above
        document.Regenerate();

        // Walk the geometry of each element's parts to get the edges
        List<ICollection<Curve>> curvesToGenerate = new List<ICollection<Curve>>();
        foreach (Element element in elements)
        {
            curvesToGenerate.AddRange(GetOutlineOfElementParts(document, element.Id));
        }

        // Discard all changes
        tempTransaction.RollBack();

        // Permanent transaction to add curves along part edges
        Transaction transaction = new Transaction(document, "Outline layer volumes");
        transaction.Start();

        foreach (ICollection<Curve> curveSet in curvesToGenerate)
        {
            foreach (Curve curve in curveSet)
            {
                AddModelCurve(document, curve);
            }
        }

        transaction.Commit();
    }

    /// <summary>
    /// Deletes sub-parts of the given element, if they exist.
    /// </summary>
    /// <param name="id">The element.</param>
    private void DeleteExistingSubParts(Document document, ElementId id)
    {
        ICollection<ElementId> parts = PartUtils.GetAssociatedParts(document, id, true, false);

        foreach (ElementId partId in parts)
        {
            if (PartUtils.HasAssociatedParts(document, partId))
            {
                PartMaker maker = PartUtils.GetAssociatedPartMaker(document, partId);
                document.Delete(maker);
            }
        }
    }

    /// <summary>
    /// Gets a collection of curves outlining each part generated by the element.
    /// </summary>
    /// <param name="id">The element id.</param>
    /// <returns>The part collection.</returns>
    private ICollection<ICollection<Curve>> GetOutlineOfElementParts(Document document, ElementId 
id)
    {
        List<ICollection<Curve>> curveSets = new List<ICollection<Curve>>();
        ICollection<ElementId> partIds = PartUtils.GetAssociatedParts(document, id, false, false);
        Options options = new Options();
        foreach (ElementId partId in partIds)
        {
            Element partElement = document.get_Element(partId);
            GeometryElement geomElem = partElement.get_Geometry(options);
            List<Curve> curves = new List<Curve>();

            foreach (GeometryObject geomObj in geomElem.Objects)
            {
                // We expect all part geometry is solid
                if (geomObj is Solid)
                {
                    Solid solid = (Solid)geomObj;
                    EdgeArray edges = solid.Edges;
                    foreach (Edge edge in edges)
                    {
                        curves.Add(edge.AsCurve());
                    }
                }
            }

            curveSets.Add(curves);
        }

        return curveSets;
    }

    /// <summary>
    /// Adds a model curve to the document aligned with the input curve.
    /// </summary>
    /// <remarks>A sketch plane needs to be calculated for the curve before it is added.</remarks>
    /// <param name="curve">The curve.</param>
    private void AddModelCurve(Document document, Curve curve)
    {
        Plane plane;
        // Linear curves have no "normal" returned by compute derivatives
        if (curve is Line)
        {
            XYZ point1 = curve.get_EndPoint(0);
            XYZ point2 = curve.get_EndPoint(1);
            XYZ tangent = (point2 - point1).Normalize();
            // Vertical line
            if (tangent.IsAlmostEqualTo(XYZ.BasisZ) || tangent.IsAlmostEqualTo(-XYZ.BasisZ))
            {
                plane = document.Application.Create.NewPlane(XYZ.BasisX, point1);
            }
            // Non-vertical line, use cross product with vertical to generate normal vector
            else
            {
                XYZ yVector = tangent.CrossProduct(XYZ.BasisZ).Normalize();
                plane = document.Application.Create.NewPlane(tangent, yVector, point1);
            }
        }
        // Non-linear curve supplies normal for sketch plane
        else
        {
            XYZ point = curve.Evaluate(0, true);
            Transform transform = curve.ComputeDerivatives(0, true);
            XYZ normal = transform.BasisZ;
            plane = document.Application.Create.NewPlane(normal, point);
        }

        // Create sketch plane
        SketchPlane sketchPlane = document.Create.NewSketchPlane(plane);

        // Create curve
        document.Create.NewModelCurve(curve, sketchPlane);
    }
}
```

Room & space geometry

The Revit API provides access to the 3D geometry of spatial elements (rooms and spaces).

The SpatialElementGeometryCalculator class can be used to calculate the geometry of a spatial element and obtain the relationships between the geometry and the element's boundary elements. There are 2 options which can be provided to this utility:

- SpatialElementBoundaryLocation – whether to use finish faces or boundary element centerlines for calculation
- StoredFreeBoundaryFaces – whether to include faces which don't map directly to a boundary element in the results.

The results of calculating the geometry are contained in the class SpatialElementGeometryResults. From the SpatialElementGeometryResults class, you can obtain:

- The Solid volume representing the geometry (GetGeometry() method)
- The boundary face information (a collection SpatialElementBoundarySubfaces)

Each subface offers:

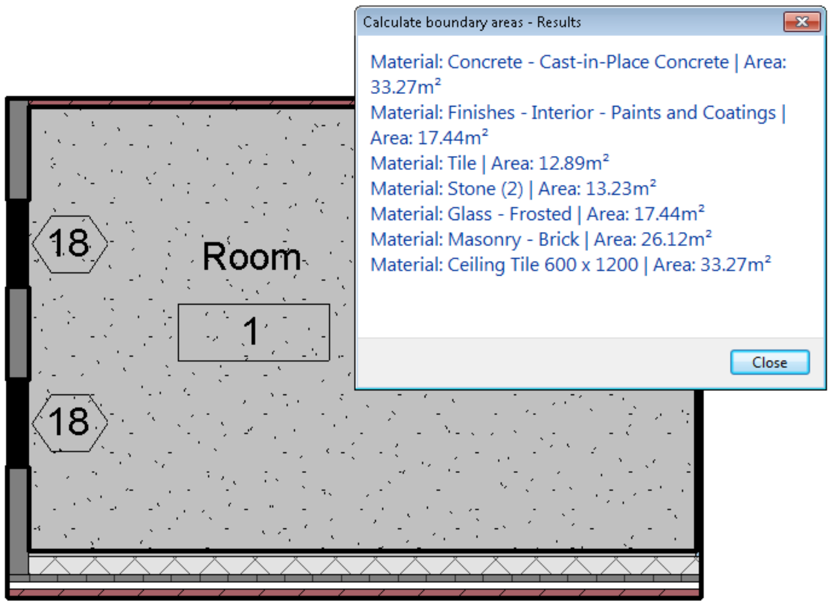
- The face of the spatial element
- The matching face of the boundary element
- The subface (the portion of the spatial element face bounded by this particular boundary element)
- The subface type (bottom, top, or side)

Some notes about the use of this utility:

- The calculator maintains an internal cache for geometry it has already processed. If you intend to calculate geometry for several elements in the same project you should use a single instance of this class. Note that the cache will be cleared when any change is made to the document.
- Floors are almost never included in as boundary elements. Revit uses the 2D outline of the room to form the bottom faces and does not match them to floor geometry.
- Openings created by wall-cutting features such as doors and windows are not included in the returned faces.

- The geometry calculations match the capabilities offered by Revit. In some cases where Revit makes assumptions about how to calculate the volumes of boundaries of rooms and spaces, these assumptions will be present in the output of the utility.

In this example, a SpatialElementGeometryCalculator is used to find the bounding faces of a room, determine the material applied to those faces, and determine total surface areas of materials for the room.



```
Material: Concrete - Cast-in-Place Concrete | Area: 33.27m²
Material: Finishes - Interior - Paints and Coatings | Area: 17.44m²
Material: Tile | Area: 12.89m²
Material: Stone (2) | Area: 13.23m²
Material: Glass - Frosted | Area: 17.44m²
Material: Masonry - Brick | Area: 26.12m²
Material: Ceiling Tile 600 x 1200 | Area: 33.27m²
```

```
/// <summary>
/// Calculates the surface area (per material) of the elements which
/// bound the spatial element.
/// </summary>
private void CalculateSpatialElementBoundaryAreas(SpatialElement element)
{
    // Setup options - finish face option is required to match
    // up to boundary faces
    SpatialElementBoundaryOptions options =
        new SpatialElementBoundaryOptions();
    options.StoreFreeBoundaryFaces = true;
    options.SpatialElementBoundaryLocation =
        SpatialElementBoundaryLocation.Finish;

    // Calculate results
    SpatialElementGeometryCalculator calculator =
        new SpatialElementGeometryCalculator(element.Document, options);
    SpatialElementGeometryResults results =
        calculator.CalculateSpatialElementGeometry(element);
    Solid solid = results.GetGeometry();


    if (solid != null)
    {
        // Traverse each face in the spatial element volume
        FaceArray faces = solid.Faces;
        foreach (Face face in faces)
        {
            IList<SpatialElementBoundarySubface> subfaces =
                results.GetBoundaryFaceInfo(face);
            foreach (SpatialElementBoundarySubface subface in subfaces)
            {
                Face spatialElementFace =
                    subface.GetSpatialElementFace();
                // Bottom faces never have a boundary face reference.
                // Find floors with another technique.
                if (subface.SubfaceType == SubfaceType.Bottom)
                {
                    double spatialElementFaceArea =
                        spatialElementFace.Area;
                    FindFloorBoundaries(element, options,
                                        spatialElementFaceArea);
                    continue;
                }

                // Get material id from bounding face.
                Face boundingElementFace =
                    subface.GetBoundingElementFace();
                ElementId materialElementId = boundingElementFace != null ?
                    boundingElementFace.MaterialElementId : ElementId.InvalidElementId;

                // Get area frm subface
                double subFaceArea = subface.GetSubface().Area;

                // Store information
                AddSurfaceArea(materialElementId, subFaceArea);
            }
        }
    }
}
```

The floors are extracted with some extra work: using the bottom outline of the room, it constructs temporary geometry extruded downward.   It then uses the ElementIntersectsSolidFilter to find floor objects which intersect this temporary geometry, and the top face of the floor (from HostObjectUtils.GetTopFaces()) to get the face and its material.

```
/// <summary>
/// Find floor boundaries for a given spatial element
/// </summary>
/// <param name="element">The spatial element.</param>
/// <param name="options">The options applied to the boundary extraction.</param>
/// <param name="area">The area of the boundary face.</param>
private void FindFloorBoundaries(SpatialElement element,
                                 SpatialElementBoundaryOptions options,
                                 double area)
{
    // Get 2D boundary of element
    IList<IList<BoundarySegment>> boundarySegments =
        element.GetBoundarySegments(options);

    foreach (IList<BoundarySegment> segmentGroups in boundarySegments)
    {
        // Build volume extending just below boundary
        List<Curve> curves = new List<Curve>();
        foreach (BoundarySegment segment in segmentGroups)
        {
            curves.Add(segment.Curve);
        }
        CurveLoop curveLoop = CurveLoop.Create(curves);
        List<CurveLoop> curveLoops = new List<CurveLoop>();
        curveLoops.Add(curveLoop);

        // Volume extruded downward 1 inch
        Solid volumeOfIntersection =
            GeometryCreationUtilities.CreateExtrusionGeometry(curveLoops,
                                                              -XYZ.BasisZ,
                                                              1/12.0);

        // Look for floors
        FilteredElementCollector collector =
            new FilteredElementCollector(element.Document);
        // Filter by category because in-place families could be floors too.
        collector.OfCategory(BuiltInCategory.OST_Floors);
        collector.WhereElementIsNotElementType();
        collector.WherePasses(new ElementIntersectsSolidFilter(volumeOfIntersection));

        // Look at intersections
        List<ElementId> materialElementIds = new List<ElementId>();
        foreach (Element intersectElement in collector)
        {
            if (intersectElement is Floor)
            {
                Floor floor = intersectElement as Floor;
                IList<Reference> topReferences =
                    HostObjectUtils.GetTopFaces(floor);

                foreach (Reference reference in topReferences)
                {
                    Face face =
                        floor.GetGeometryObjectFromReference(reference) as Face;
                    if (face != null)
                    {
                        ElementId materialElementId = face.MaterialElementId;
                        if (!materialElementIds.Contains(materialElementId))
                            materialElementIds.Add(materialElementId);
                    }
                }
            }
            else
            {
                // TODO - not a floor, in-place family perhaps.
                // This case is not handled by this sample.
            }
        }

        if (materialElementIds.Count == 1)
        {
            AddSurfaceArea(materialElementIds[0], area);
        }
        else
        {
            //TODO - multiple element faces or elements found by intersection.
            // This case is not handled by this sample.
        }
    }
}
```

Energy analytical model

The Autodesk.Revit.DB.Analysis namespace includes several classes to obtain and analyze the contents of a project's detailed energy analysis model. The Export to gbXML and the Heating and Cooling Loads features produce an analytical thermal model from the physical model of a building. The analytical thermal model is composed of spaces, zones and planar surfaces that represent the actual volumetric elements of the building.

The classes related to the detailed energy analysis model are:

- EnergyAnalysisDetailModel
- EnergyAnalysisDetailModelOptions
- EnergyAnalysisOpening
- EnergyAnalysisSpace

- EnergyAnalysisSurface
- Polyloop

Use the static method EnergyAnalysisDetailModel.Create() to create and populate the energy analysis model. Set the appropriate options using the EnergyAnalysisDetailModelOptions. The generated model is always returned in world coordinates, but the method TransformModel() transforms all surfaces in the model according to ground plane, shared coordinates and true north.

The options available when creating the energy analysis detail model include:

- The level of computation for energy analysis model - NotComputed, FirstLevelBoundaries, meaning analytical spaces and zones, SecondLevelBoundaries, meaning analytical surfaces, or Final, meaning constructions, schedules, and non-graphical data
- Whether mullions should be exported as shading surfaces
- Whether shading surfaces will be included
- Whether to simplify curtain systems - When true, a single large window/opening will be exported for a curtain wall/system regardless of the number of panels in the system

Note that the model will be returned exactly as Revit calculates it for export to gbXML. But the applications are not necessarily limited to use in Energy Analysis scenarios directly. Revit's IFC exporter (available on Open Source) uses the second-level boundary calculation to populate the related IFC entities.

Point clouds

The point cloud client API supports read and modification of point cloud instances within Revit. The points supplied by the point cloud instances come from the point cloud engine, which is either a built-in engine within Revit, or a third party engine loaded as an application. A client point cloud API application doesn't need to be concerned with the details of how the engine stores and serves points to Revit. Instead, the client API can be used to create point clouds, manipulate their properties, and read the points found matching a given filter.

The main classes related to point clouds are:

- PointCloudType - type of point cloud loaded into a Revit document. Each PointCloudType maps to a single file or identifier (depending upon the type of Point Cloud Engine which governs it).
- PointCloudInstance - an instance of a point cloud in a location in the Revit project.
- PointCloudFilter - a filter determining the volume of interest when extracting points.
- PointCollection - a collection of points obtained from an instance and a filter.
- PointIterator - an iterator for the points in a PointCollection.
- CloudPoint - an individual point cloud point, representing an X, Y, Z location in the coordinates of the cloud, and a color.

Accessing Points in a Point Cloud

There are two ways to access the points in a point cloud:

1. Iterate the resulting points directly from the PointCollection return using the IEnumerable<CloudPoint> interface
2. Get a pointer to the point storage of the collection and access the points directly in memory in an unsafe interface

Either way, you first access a collection of points from the PointCloudInstance using the method

- PointCloudInstance.GetPoints(PointCloudFilter filter, int numPoints)

Note that as a result of search algorithms used by Revit and the point cloud engine, the exact requested number of points may not be returned.

Although you must deal with pointers directly in the second option, there may be performance improvements when traversing large buffers of points. However, this option is only possible from C# and C++/CLI.

Filters

Filters are used both to limit the volume which is searched when reading points, and also to govern the display of point clouds. A PointCloudFilter can be created based upon a collection of planar boundaries. The filter will check whether a point is located on the "positive" side of each input plane, as indicated by the positive direction of the plane normal. Therefore, such filter implicitly defines a volume, which is the intersection of the positive half-spaces corresponding to all the planes. This volume does not have to be closed, but it will always be convex.

The display of point clouds can be controlled by assigning a filter to:

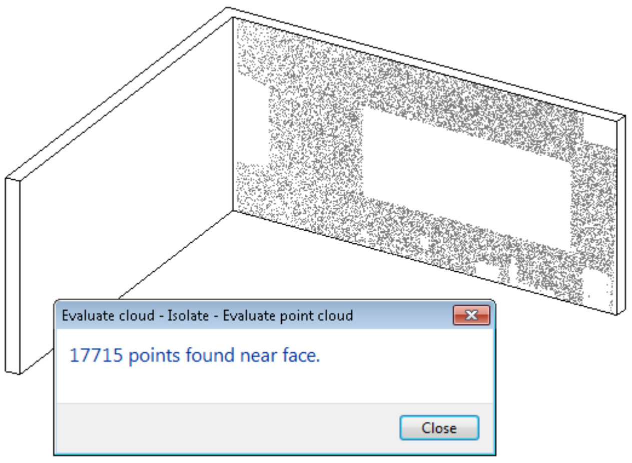
- PointCloudInstance.SetSelectionFilter()

Display of the filtered points will be based on the value of the property:

- PointCloudInstance FilterAction

If it is set to None, the selection filter is ignored. If it is set to Highlight, points that pass the filter are highlighted. If it is set to Isolate, only points that pass the filter will be visible.

In this example, we use the filter capabilities to evaluate how well a given element aligns with points in the point cloud. A filter is built based on a face of a wall, floor, or roof; this filter is constructed to find points within a narrow epsilon of the element face. The matching points can be isolated or highlighted, and a count of the matching points is returned.


Evaluate cloud - Isolate - Evaluate point cloud

17715 points found near face.

Conclusion & Caveats

This handout reviewed the capabilities of some new powerful tools in the Revit API for geometry analysis, calculation and display.   These included creation of temporary geometry, Boolean operations, extrusion analysis, and element-specific tools like spatial element geometry calculation and parts generation. Autodesk believes that the ability for Revit add-in developers to execute on specific geometric analyses and produce desired results is significantly improved over previous releases as a result.

As with all programming tools where end users are involved, developers need to keep edge cases in mind. It is certainly possible for users to model situations which don't mesh well with the tools and might result in failure situations.   In the samples and examples provided in this course, these situations are not handled robustly; anyone using these examples would need to consider many more test cases than have been demonstrated here. Developers should also consider how robust their solution should be: are there situations where it should or should not produce valid results?