**Welcome to EvoluteTools PRO for Rhino3D !**

Thank you for purchasing EvoluteTools PRO for Rhino3D. It bundles Evolute's extensive geometry processing expertise into a user-friendly and comfortable application by using the excellent framework of Rhinoceros. The **PRO** version of the plugin offers access to the following features:

- RhinoScript and Half-Edge data structure support
- unlimited mesh size
- multi-resolution mesh modeling
- global subdivision rules
- local subdivision rules
- mesh editing tools
- mesh optimization for:
  - closeness to reference surfaces
  - fairness
  - hybrid meshes (mixtures of triangles, planar quads, planar pentagons, planar hexagons, ...)
  - planar quad meshes
  - ideal edge length, with localized control
  - ideal panel
  - conical meshes
  - circular meshes
  - sphere packings
  - coplanarity
  - developable strips
- modeling and optimization of N-gon meshes (PolyMeshes)
- export and import tools for N-gon meshes
- constraining vertex movement to normals
- local fairing controls
- specification of creases/kinks
- specification of vertices as anchor/corner points
- specification of additional reference curves, e.g. in the interior of reference surfaces, useful for:
  - alignment of vertices with floor slabs
  - alignment of vertices with predefined panelisation curves
  - fine-grained control of panel layouts
- coplanarity constraints for sets of vertices, useful for:
  - alignment of vertices with structural elements
  - simplification of panelisation curves
  - constraining vertices to general planes
  - constraining vertices to special planes (horizontal, vertical, user defined, etc)
- analysis modes:
  - principle curvature lines
  - conjugate curve field analysis and extraction
  - closeness to a reference object
  - planarity of panels
  - edge lentgth distribution
- pattern mapping (points, polylines, meshes)

If you are interested in using EvoluteTools Academic PRO for teaching, please enquire about the **Evolute.Link** program.

Please check http://www.evolutetools.com for details and upcoming releases.

We would like to encourage you to post any questions or problems you might have, as well as bugs you might find, to the appropriate discussions in our forum. Additionally, as development of this software constantly continues and details about future functionality have yet to be determined, you are strongly encouraged to provide us with feedback about all aspects of EvoluteTools PRO for Rhino via the forum. We are especially interested in learning which parts of EvoluteTools PRO you find most useful, and if something important is still missing. So if you like what you see but there is some functionality that would make using EvoluteTools PRO for Rhino even more convenient, please let us know!

We hope you will enjoy working with EvoluteTools PRO for Rhino and we are looking forward to hearing from you soon!

The Evolute development team



### Installation Instructions

Start Rhino and use the Plugin Manager (command "_PluginManager") to install EvoluteTools PRO for Rhino (EvoluteToolsForRhino.rhp), or simply drag & drop EvoluteToolsForRhino.rhp into a Rhino window. You can install the EvoluteTools PRO for Rhino Toolbar by dragging & dropping EvoluteToolsForRhino.tb to the main Rhino Window. Be sure to keep the help file EvoluteToolsForRhino.chm in the same directory as the plugin.

You should have received a serial number during the purchasing process. EvoluteTools PRO for Rhino will ask you to enter the serial number upon first use. You may use the command "etLicenseManager" in Rhino to add or transfer a serial number later. If a file "serial.txt" exists in the same directory as the plugin, its contents will be filled into the serial number dialog.

If you want to use syntax auto-completion in Rhino's script editor (aka Monkey editor), please run "InstallRhinoScriptSyntaxDescription.bat" and restart Rhino. Please watch out for errors, you might need to run this script using administrator privileges.

If you receive this following error while trying to load the Plugin: "Unable to load EvoluteToolsForRhino.rhp plug-in: could not open the plugin file" - then install this missing Microsoft Update:
http://www.microsoft.com/download/en/details.aspx?id=26347

Stay updated by subscribing to our our forum at http://www.food4rhino.com/project/evolutetools-lite and to the newsletter.

### Acknowledgements

This work was partially supported by the European Community's 7th Framework "People" Programme through an IAPP project under grant agreement 230520 (ARC).

# etLoftDevelopable

*Creates developable surfaces from arbitrary pairs of curves or quad meshes.* Only available in EvoluteTools PRO for Rhino, developable lofting module.

This command creates a single developable strip surface or a collection of developable strip surfaces, depending on the user input. If the input is a pair of arbitrary curves, the command creates a single developable strip between those two curves, if the input is a quad mesh then the user can choose to create a single developable strip by selecting a specific edge or multiple strips for the entire mesh.

**Typical inputs:**

- *Pairs of arbitrary curves:* preferably the curves should be smooth and continuous, without kinks, however, densely sampled polylines can also be used.
- *Quad meshes:* regular quad meshes are preferred, however, quad meshes with interior holes or quad meshes with irregular boundaries can also be used.

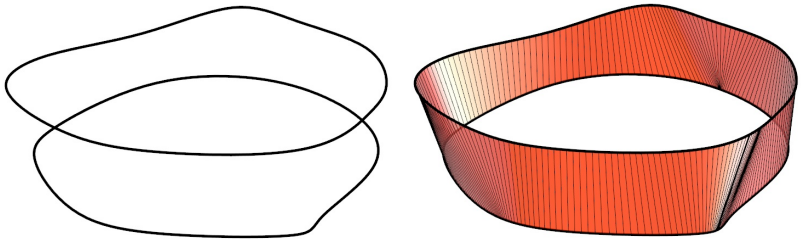
If the command is unsuccessful, the user is asked whether to continue or not, if 'yes', a series of parameters can be controlled by the user in the following step.
**Input parameters:**

- *Maximum distance to curves ( default = unit absolute tolerance ):* this is the maximum deviation allowable from the initial curves.
- *Maximum developability tolerance ( default = 0.001 ):* this measure controls the amount of twist (double curvature) allowed in the lofted strip surface. The measure itself is the *Planarity Scale Invariant* value, used and displayed in the Analysis Modes for meshes in EvoluteTools PRO. A very low value produces virtually single curved surfaces (developable), while a higher value allows a certain amount of double curvature in the lofted surface. It is recommended to keep the default value and first increase the *Maximum Distance To Curves* parameter if the command fails at the first attempt.

---

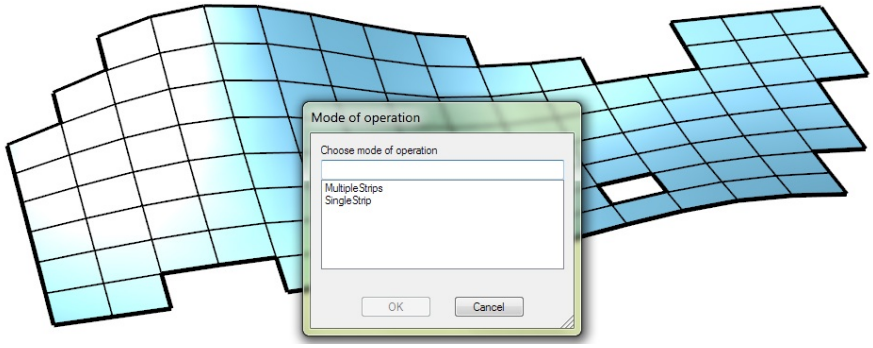
**Typical usage:**

1. Pairs of curves

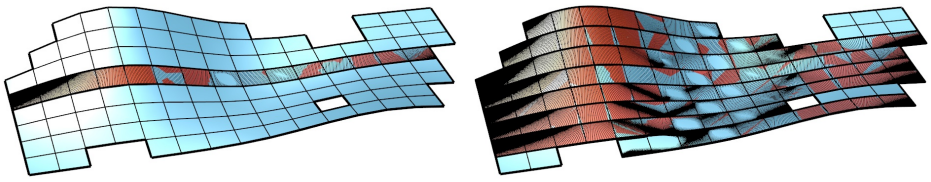


Curves can be open or closed, smooth (upper left) or polylines. The user needs to invoke the command *etLoftDevelopable*, select the desired curves and if the optimized lofting succeeds, the result will be immediately displayed (upper right). If the tolerances are quite tight, the optimization might run for a few seconds. If the lofting does not succeed at the first attempt, the user is asked whether to continue and to adjust the input parameters. It is important to note that not all arbitrary curves can be lofted into a smooth developable surface due to inherent geometric constraints ( twist, kinks, etc).

2. Meshes



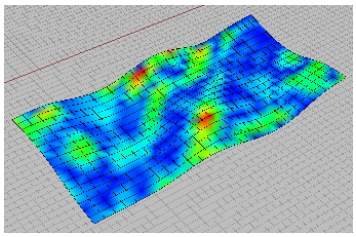

Almost any quad mesh can be used as an input for the *etLoftDevelopable* command. When a mesh is used as an input, the user is asked whether it is preferred to create a single developable surface strip corresponding to a row of quads (below - left), or several developable surface strips corresponding to all rows of quads (below - right) along one direction of the mesh. The direction of the created surfaces is orthogonal to the starting edge picked by the user. If the lofting fails to find a solution at the first attempt, the user is asked whether to continue and to adjust the input parameters. If the command succeeds, the resulting developable surfaces are displayed, along with the curves that served as references for the lofting (the command creates curves in the background, corresponding to the rows of vertices of the mesh).



## etAnalyzeCloseness

*Displays a color-coded analysis mesh visualizing the closeness to reference surfaces.*

If a reference surface is set, this command starts Mesh Analysis for the selected meshes, opens the Mesh Analysis dialog, and switches Mesh Analysis mode to Closeness.



Hints

Color coding is updated as you update the meshes for which Mesh Analysis is activated. Mesh analysis can be deactivated using etAnalyzeOff or by closing the Mesh Analysis dialog.

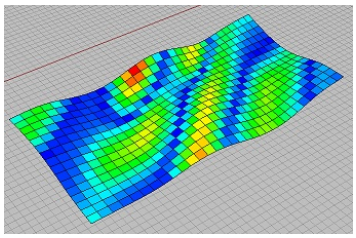
## etAnalyzeOff

*Deactivates Mesh Analysis for all meshes.*

## etAnalyzePlanarity

*Displays a color-coded analysis mesh visualizing the planarity of mesh faces.*

This command starts Mesh Analysis for the selected meshes, opens the Mesh Analysis dialog, and switches Mesh Analysis mode to Planarity.



**Hints**

Color coding is updated as you update the meshes for which Mesh Analysis is activated. Mesh analysis can be deactivated using etAnalyzeOff or by closing the Mesh Analysis dialog.

## etBallPackingMeshExtractBalls

*Extracts balls centered at vertices of a ball packing mesh, and adds them to the document.*

This command requires a triangle mesh which has been optimized for the ball packing property. If you want to optimize a triangle mesh for the ball packing property, use etOptionsImportance to set an importance value for BallPacking and then run etOptimize.

 (right mouse button)

## etClearReference

*Clears the list of reference surfaces and curves.*

This command empties the list of reference geometries set using the etSetReference command. It can be invoked by right-clicking on the etSetReference button.

## etDecoupleSubdivision

*Delete the logic connection between a mesh and its subdivided child mesh and/or coarse parent mesh.*

Amesh created by the command etSubdivide is logically connected to its parent mesh and will reproduce deformations and other alterations performed on that mesh. To be able to edit the meshes separately, using this command you can decouple a mesh from both its parent and/or children meshes. See also "Single mesh mode" and "Subdivision mode" of etOptimize.

# etExtractConjugateLines

*Allows tracing of conjugate lines for surfaces, polysurfaces, and meshes.*
*Only available in EvoluteTools PRO*

This command is useful for determining potential mesh connectivities for approximating a smooth surface by a planar quad mesh or a developable strip model. Given a family of curves on a smooth surface (defining a family of panelisation curves), the conjugate family of curves suggests how to design the transverse panelisation curves such that quads will be close to planar (respectively strips will be close to developable). A mesh topology designed by following such conjugate families of curves is best suitable for further optimization towards planarity respectively developability.

A single surface, polysurface, or mesh object needs to be specified before conjugate lines can be traced. Furthermore the user is asked to select the curves which should be used for computing the conjugate direction vector field. Those curves need not be exactly on the surface, they will be projected to it. Surface and polysurface objects are meshed before tracing, in this case the user is prompted for a sensible tolerance value based on the document's absolute tolerance. Resulting conjugate lines are always polylines on this mesh. After curvature estimation, the user may pick points on the object to start tracing conjugate lines, or enter a number of lines that should be traced automatically.

**Curvature estimation options**

- *Order:* Degree of fitting polynomial used for curvature estimation.
- *Area:* Area of local neighbourhood used for polynomial fitting when estimating curvature.

**Tracing options**

- *MaxLength:* Stop conjugate line tracing once the traced polyline has reached the given length.
- *MaxAngle:* Stop conjugate line tracing if the traced polyline would possess a kink higher than the given angle.
- *ContinueWithGeodesic:* If set to Yes and the MaxAngle criterion is violated, continue tracing along a geodesic curve, until the MaxAngle criterion is fulfilled again.
- *TraceCurveField:* If set to Yes, blue polylines will be traced along the input vector field, which was generated using the selected curves.
- *TraceConjugateField:* If set to Yes, red polylines will be traced along the conjugate vector field.
- *SmoothCurveField:* Smooth the input vector field and recompute the conjugate vector field. This is useful for sparsely given input curves.

## etExtractCurvatureLines

*Allows tracing of principal curvature lines for surfaces, polysurfaces, and meshes.*
*Only available in EvoluteTools PRO*

Asingle surface, polysurface, or mesh object needs to be specified before principal curvature lines can be traced. Surface and polysurface objects are meshed before tracing, in this case the user is prompted for a sensible tolerance value based on the document's absolute tolerance. Resulting principal curvature lines are always polylines on this mesh. After curvature estimation, the user may pick points on the object to start tracing minimum and maximum principal curvature lines, or enter a number of lines that should be traced automatically.

**Curvature estimation options**

- *Order:* Degree of fitting polynomial used for curvature estimation.
- *Area:* Area of local neighbourhood used for polynomial fitting when estimating curvature.

**Tracing options**

- *MaxLength:* Stop curvature line tracing once the traced polyline has reached the given length.
- *MaxAngle:* Stop curvature line tracing if the traced polyline would possess a kink higher than the given angle.
- *ContinueWithGeodesic:* If set to Yes and the MaxAngle criterion is violated, continue tracing along a geodesic curve, until the MaxAngle criterion is fulfilled again.
- *TraceMax:* If set to Yes, red polylines will be traced along the maximum principal curvature directions.
- *TraceMin:* If set to Yes, blue polylines will be traced along the minimum principal curvature directions.

## etLicenseManager

*Lets you manage your Licenses for EvoluteTools for Rhino.*

This command offers the following options for managing your licenses for EvoluteTools for Rhino:

- *Add*: Allows you to add a license by entering a serial number which you received when purchasing EvoluteTools for Rhino.
- *GiveBackAllFloating*: Transfer all floating licenses back to the online license server. Use this option if you want to transfer all floating licenses back to the license server, making it available for other users immediately. Usually floating licenses would stay checked out to your workstation until you close Rhino. You may check out the floating licenses again using the option Validate. Floating licenses will be checked out again automatically when loading the plugin the next time.
- *CurrentLicense*: Choose which license to view, validate or transfer. This option is only visible if at least one license is stored locally on your computer.
- *View*: View description of current license and check if it is still valid.
- *Validate*: Renew and check status of current license.
- *GiveBackFloating*: Transfer current license back to the online license server. This option is only available for floating licenses. Use it if you want to transfer a floating license back to the license server, making it available for other users immediately. Usually floating licenses would stay checked out to your workstation until you close Rhino. You may check out the floating license again using the option Validate. Floating licenses will be checked out again automatically when loading the plugin the next time.
- *KeepLocal=Yes / KeepLocal=No*: Use this option to configure whether a floating license should stay checked out to your workstation when you close Rhino. Keeping a floating license local allows you to use it when working offline. Be aware that floating licenses need to be renewed from the online license server once in a week (see option Validate). Contact support@evolute.at if you would like this interval to be changed.
- *Transfer*: Transfer current license permanently back to online license server. Use this option in the following cases:
  - You want to transfer a floating license permanently back to the license server, making it available for other users immediately. You may check out the floating license again using the option Add (you will need to re-enter the serial number).
  - You want to transfer a node-locked license to another workstation.
- *ManualTransfer*: Use this only if you were asked to do so by a representative of Evolute. Asks you for a serial number, deletes any licenses with this serial number from your workstation, and transfers the serial number back to the license server.

The floating license feature requires Internet access and correctly configured network / proxy settings for Internet Explorer. If an Internet connection is unavailable the license cannot be retrieved from the license server, therefore the Plugin will not work.

## etMeshAddDiagonal

*Cuts a quad face into two triangle faces by inserting a diagonal edge.*

This command takes two vertices as input. If the vertices are diagonally opposite vertices of a quadrilateral mesh face, the diagonal between them is inserted as a new edge.

---

Image 1: Initial Mesh

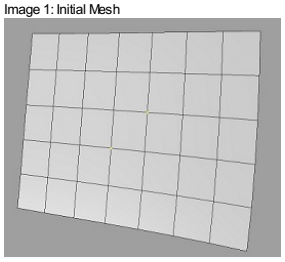
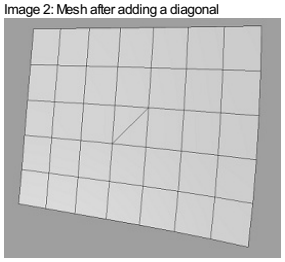


Image 2: Mesh after adding a diagonal



## etMeshAddDiagonalLine

*Inserts a diagonal line to a quad mesh.*

This command takes two vertices as input. If the vertices are diagonally opposite vertices of a quadrilateral mesh face, the diagonal between them is inserted as a new edge. If possible, this diagonal is continued until either a non-quad face or a mesh boundary is encountered.

---

Image 1: Initial Mesh

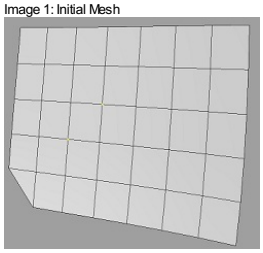
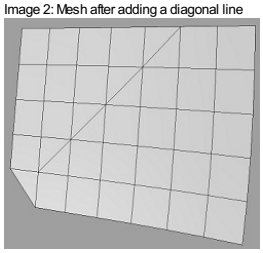


Image 2: Mesh after adding a diagonal line



## etMeshAddDiagonalLinesParallel

*Inserts parallel diagonal lines to a quad mesh, converting it to a triangle or hybrid mesh.*

This command takes two vertices as input. Additionally, an integer valued interval can be provided. If the vertices are diagonally opposite vertices of a quadrilateral mesh face, the diagonal between them is inserted as a new edge. If possible, this diagonal is continued until either a non-quad face or a mesh boundary is encountered. This process is repeated for the whole area consisting of quad faces by inserting a number of parallel lines. The number of faces left out between each pair of lines is determined by the value of interval. If interval is zero, the resulting faces are all triangular; otherwise a hybrid mesh consisting of quad and triangle faces is created.

Image 1: Initial Mesh

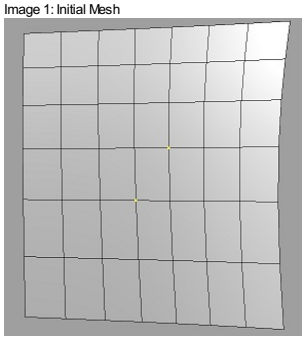


Image 2: Diagonal lines with interval 0

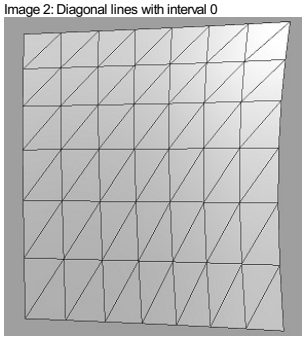
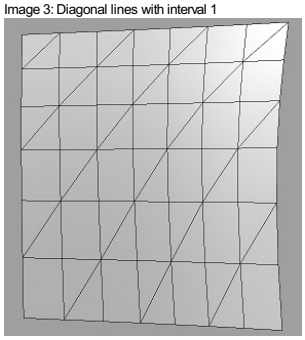


Image 3: Diagonal lines with interval 1



## etMeshCut

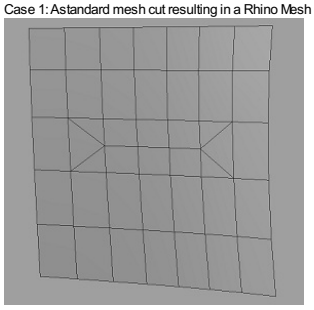
*Divides a row of quadrilateral mesh faces into two by cutting each face in half.*

This command takes two mesh edges as input. If a row of quadrilateral mesh faces connects these edges, the faces along the row are cut in half by inserting new edges starting from the midpoints of the input edges. The command has two modes of operation:

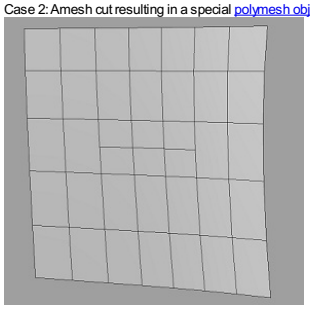
- By default, the option AcceptNgons is set to false. This causes EvoluteTools to split the quadrilateral mesh faces at both ends of the newly inserted cut into triangles. This is done because otherwise these quad faces would contain 5 mesh vertices and therefore, technically, become pentagons. As Rhino does not support mesh faces with more than 4 vertices, this is the only way to facilitate such a mesh cut and still create a standard Rhino mesh object.
- If the option AcceptNgons is set to true, the triangle split is not performed. This usually leads to the creation of a polymesh object containing pentagons. These objects are not supported by Rhino and not fully functional in this version of EvoluteTools for Rhino. See the specific documentation topic about this issue for more information.

---

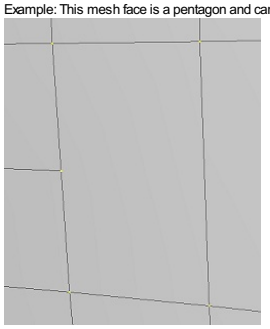
Case 1: A standard mesh cut resulting in a Rhino Mesh Object.



Case 2: A mesh cut resulting in a special polymesh object introduced by EvoluteTools but unsupported by Rhino.



Example: This mesh face is a pentagon and can therefore not be realized using standard Rhino Mesh Objects.



## etMeshDeleteEdge

*Deletes mesh edges while leaving all optimization settings intact.*

This command allows you to delete any number of mesh edges while keeping all vertex optimization settings, like curve-points, corners and coplanarity of vertices, intact. Subdivision connections will also be preserved. If deleting an edge leaves an interior vertex with only two remaining adjacent edges, these edges will be replaced by a single one and the vertex will be deleted.
Please note that deleting edges can lead to polymeshes.

---

Image 1: Initial Mesh

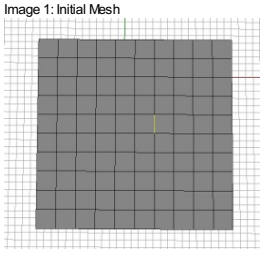
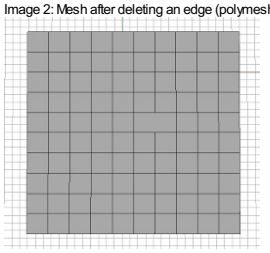


Image 2: Mesh after deleting an edge (polymesh)



## etMeshDeleteFace

*Deletes mesh faces while leaving all optimization settings intact.*

This command allows you to delete any number of mesh faces while keeping all vertex optimization settings, like curve-points, corners and coplanarity of vertices, intact. Subdivision dependencies will also be preserved. This should be used instead of either Rhino's DeleteMeshFaces command or the standard Delete command, as both of these will destroy all the optimization settings.

## etMeshDeletePolyline

*Removes polylines from a mesh while leaving all optimization settings intact.*

This command allows you to remove any number of mesh polylines while keeping all vertex optimization settings, like curve-points, corners and coplanarity of vertices, intact. Subdivision connections will also be preserved. Polylines can be chosen by selecting any mesh edge belonging to them.

---

Image 1: Initial Mesh

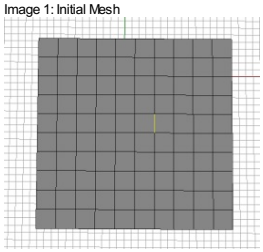
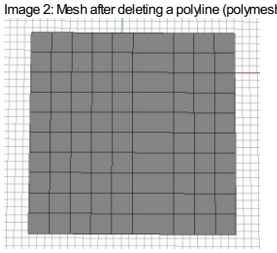


Image 2: Mesh after deleting a polyline (polymesh)



## etMeshDeleteVertex

*Deletes mesh vertices while leaving all optimization settings intact.*

This command allows you to delete any number of mesh vertices while keeping all vertex optimization settings, like curve-points, corners and coplanarity of vertices, intact. Subdivision dependencies will also be preserved. This should be used instead of the standard Rhino Delete command which will destroy all these settings.

## etMeshExtractPolylines

*Extracts all polylines of a mesh and adds them to the document.*

This command converts an input mesh or polymesh into a number of polyline objects following the combinatorics of the mesh.

## etMeshLoopCut

*Divides a row of quadrilateral mesh faces into two by cutting each face in half.*

This command takes a mesh edge as input. The adjacent faces are cut in half by inserting a new edge starting from the midpoint of the input edge. The cut is then continued in both directions until a non-quad face or the mesh boundary is reached.

---

Image 1: Initial mesh

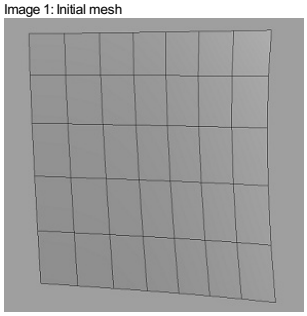


Image 2: Mesh after loop cut

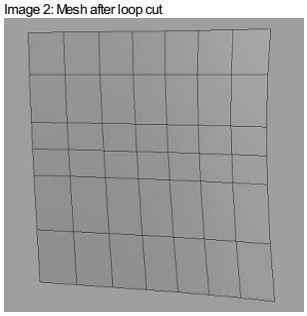


Image 3: Initial mesh

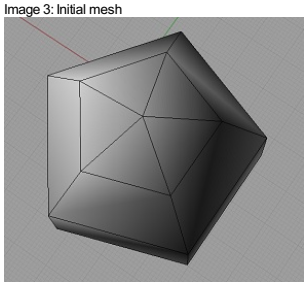
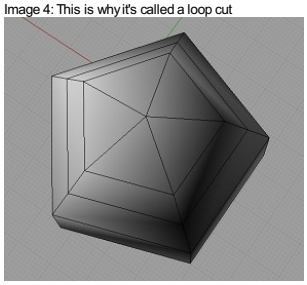


Image 4: This is why it's called a loop cut



## etMeshOffset

*Offsets any mesh with a constant spacing.*
*Only available in EvoluteTools PRO*

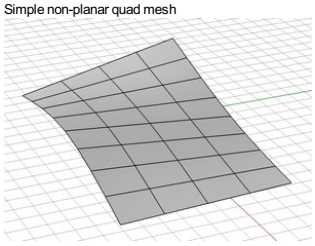
This command takes any mesh as input (polymesh, hybrid mesh, quad mesh). It outputs a mesh offset with a user defined constant spacing, and has the following options:

- CreateNodeAxes: Connects the vertices of the input mesh with the respective vertices of the offset mesh by line segments. The generated line segments are automatically grouped for easy selection.
- Distance: Specifies the constant distance of the offset.

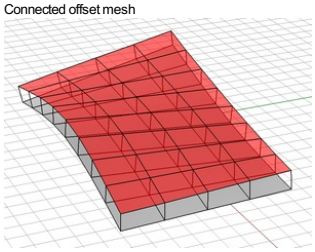
The command does not delete the input mesh.

---

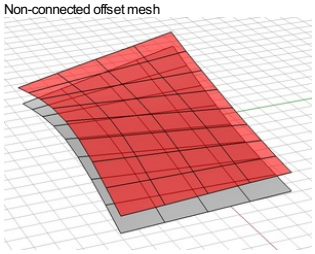
Simple non-planar quad mesh



Connected offset mesh



Non-connected offset mesh



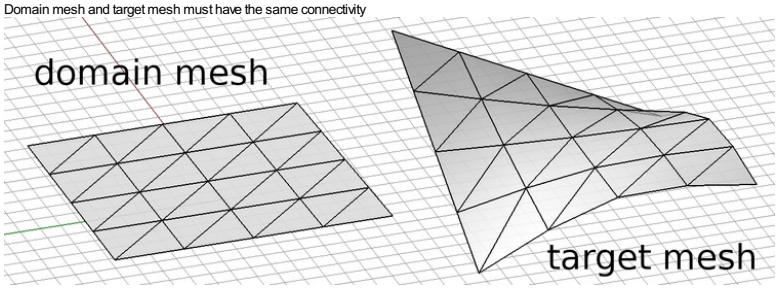
## etMeshPatternMapper

*Maps objects found on a domain mesh to a target mesh.*
*Only available in EvoluteTools PRO*

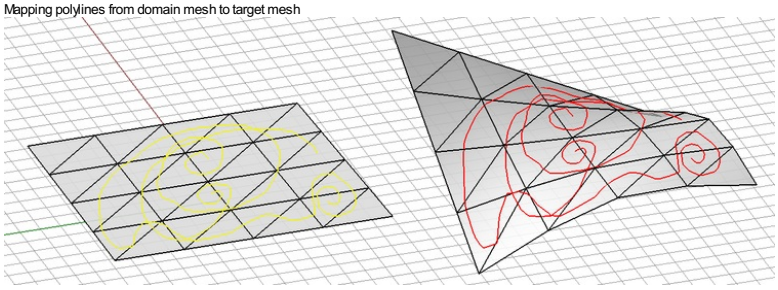
Domain and target mesh must be standard triangular meshes and have the same connectivity. The command takes points, polylines, standard meshes, hybrid meshes or T-Splines meshes as input and maps them to a user picked target mesh. The input objects must be found on a domain mesh, or within a specified tolerance outside the borders of the domain mesh. Object selected away from the domain mesh or outside the tolerance limit will not be mapped. Polymeshes or hybrid polymeshes are not supported.

---

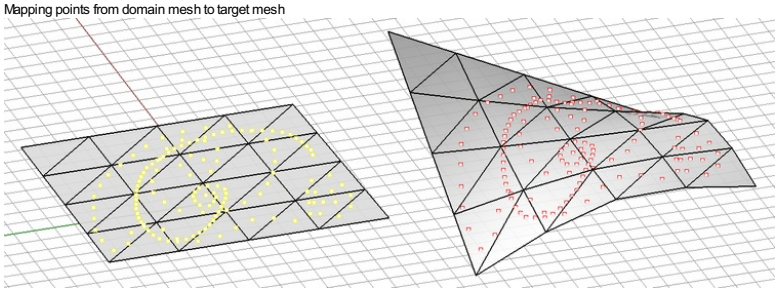
Domain mesh and target mesh must have the same connectivity



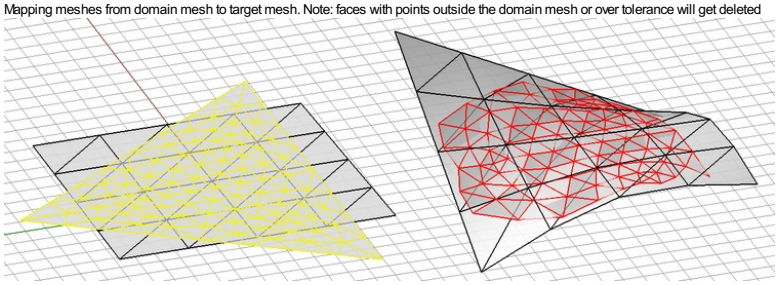

Mapping polylines from domain mesh to target mesh



Mapping points from domain mesh to target mesh



Mapping meshes from domain mesh to target mesh. Note: faces with points outside the domain mesh or over tolerance will get deleted

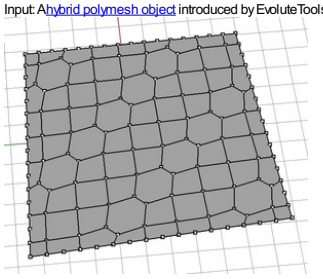


## etMeshRemoveNGons

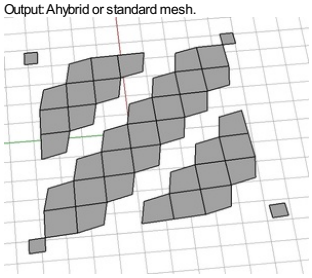
*Deletes all N-gons from a hybrid polymesh.*

This command takes hybrid polymeshes (N-gon meshes) as input. The output can be a hybrid mesh (quads and triangles) or a standard mesh (quads or triangles).

Input: Ahybrid polymesh object introduced by EvoluteTools but unsupported by Rhino.



Output: Ahybrid or standard mesh.



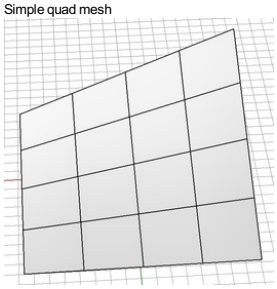
## etMeshSplitFace

*Splits quadrangular or triangular faces in smaller triangles starting from user picked points.*

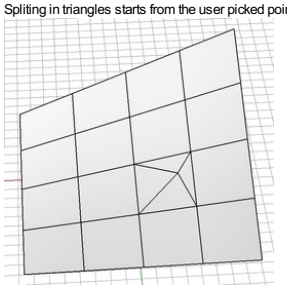
This command takes standard quad meshes or hybrid meshes (quads + triangles) as input. After selecting the mesh the user can pick points inside the quadrangular or triangular faces, which will be split into triangles. The command will not work with polymeshes !

---

Simple quad mesh



Spliting in triangles starts from the user picked points towards the face corners



# etMeshToPlanarPanels

*Creates NURBS planar panels with gaps from a watertight non-planar quad mesh or polymesh.*
*Only available in EvoluteTools PRO*

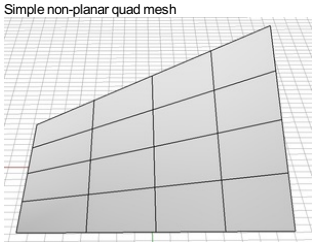
This command takes any mesh as input (polymesh, hybrid mesh, quad mesh), and it will output planar NURBS surfaces with gaps respective to the polygonal or quad faces present in the input mesh. Fitting of the planar NURBS surfaces uses two methods:

- LeastSquaresFittingPlanes: Minimizes the sum of squared distances from each corner of a non-planar face to the fitted plane. This method is used by default for non-planar polygonal faces.
- NormalToDiagonalDistance: Fitted plane is normal to and starts from the middle of the shortest diagonal distance of a non-planar quad face. Also see Mesh Analysis.

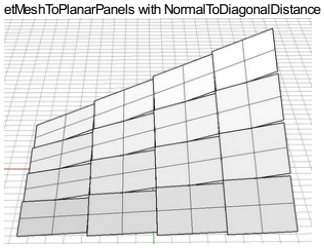
The command does not delete the input mesh.

---

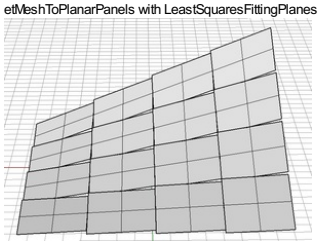
Simple non-planar quad mesh



etMeshToPlanarPanels with NormalToDiagonalDistance method



etMeshToPlanarPanels with LeastSquaresFittingPlanes method



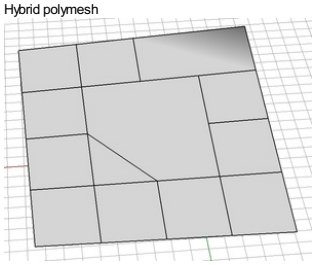
## etMeshTriangulateNGons

*Splits all NGons found in a mesh into triangular faces.*

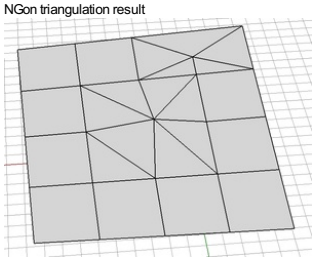
This command takes polymeshes or hybrid polymeshes as input. The output is a hybrid mesh or standard triangular mesh.

---

Hybrid polymesh



NGon triangulation result



## etOptimize

*Optimizes a mesh according to the current optimization settings.*

This is the main command used for mesh optimization. It uses the optimization importance settings set by etOptionsImportance and the toggles set by etOptionsToggles. Using this command by default performs one iteration of the mesh optimization - several iterations might be necessary to obtain an optimum solution.

etOptimize has two modes of operation:

- Single mesh mode: This mode is used if the selected mesh is neither a subdivision parent or child mesh.
- Subdivision mode: If the selected mesh is the subdivision child of another mesh ( it has been created using etSubdivide ), the vertices of the child mesh depend linearly on those of the parent. In this case the optimization is performed on the vertices of the parent mesh, but optimizing for goals of the child mesh. This has drawbacks and advantages:
  - Drawbacks:
    - The optimization is performed using much less vertices than if it was performed directly on the child mesh. This leads to less degrees of freedom which limits the possibilities of the optimization. In general the optimization goals for the child mesh will be fulfilled better if optimizing it in single mesh mode.
    - Since the vertices of the coarser mesh are moved solely with respect to optimizing those of the finer one, the optimization can lead to a distorted parent mesh.
  - Advantages:
    - When using multiscale modeling, intermediate optimization steps can be used without decoupling the dependent meshes. This is useful in iterative approaches where editing the coarser as well as the finer meshes is alternated with mesh optimization. Ideally, the final optimization step is done in single mesh mode to facilitate the advantages of having more degrees of freedom.

## etOptimizeFairness

*Optimizes mesh fairness.*

This button runs a script which calls etOptionsImportance to set the following importance values, before calling etOptimize:

- Planarity = 0
- FairnessCurvature = 1
- SurfaceCloseness = 1
- CurveCloseness = 1

Hint

Hold Shift while right-clicking the button to adapt the script to your needs.

## etOptimizePlanarity

*Optimizes mesh planarity.*
*Only available in EvoluteTools PRO.*

This button runs a script which calls etOptionsImportance to set the following importance values, before calling etOptimize:

- Planarity = 1
- FairnessCurvature = 0.1
- SurfaceCloseness = 1
- CurveCloseness = 1

**Hint**

Hold Shift while right-clicking the button to adapt the script to your needs.

# etOptionsImportance

*Allows specifying values for the importance of optimization goals.*

This is the command used to set the global optimization importance values for the goals of the mesh optimization command etOptimize. These values specify the importance of the different optimization goals during the optimization. In some cases the optimization goals may contradict each other and lead the optimizer to move the mesh vertices in different directions. In these cases, the term with the higher importance prevails.

**Optimization importance values:**

- *SurfaceCloseness*: minimizes the distance of the mesh vertices to the reference surface(s) set by etSetReference.
- *CurveCloseness*: minimizes the distance of boundary vertices to the nearest boundary of the reference surface(s). The boundaries of the mesh and the reference surface(s) are automatically determined by the optimizer. With EvoluteTools PRO for Rhino the user can designate additional mesh vertices as curve points using etSetVertexCurvepoint, and reference curves using etSetReference. This can be used to align interior mesh vertices along predefined curves. See also DefaultBoundaryCloseness.
- *OriginalCloseness*: minimizes the distance of vertices to their original position, i.e. the change of vertex coordinates is minimized.
- *NormalCloseness*: minimizes the deviation of vertices from their current vertex normal. This term tries to restrict vertex movement to moving along these vertex normals. Use in conjuction with etSetVertexRestrictedToNormal. Only available in EvoluteTools PRO for Rhino
- *FairnessSprings*: minimizes the edge lengths, mimics what would happen if all edges were springs.
- *FairnessCurvature*: is used to minimize the visible kinks in the mesh and giving it a smooth appearance. In short, if a mesh is viewed as network of polylines, this term tries to make them as straight as possible.
- *FairnessCurvatureVariation*: instead of trying to straighten the polylines mentioned before ( by reducing their curvature to 0 ), this term tries to keep their curvature constant - ideally circular.
- *IdealEdgeLength*: Optimizes the edge lengths to be equal, or to be equal to the factor of the same name, set in etOptionsToggles. The optimization will ONLY affect the edges flagged with etSetEdgeLengthOptimization. Only available in EvoluteTools PRO for Rhino
- *IdealPanel*: Rather experimental optimization term, applies to triangle and quad faces. The shapes of triangular faces are optimized towards being equilateral with side lengths as set in the *IdealEdgeLength* parameter in etOptionsToggles. Quad faces are optimized towards squares of the same side length. Only available in EvoluteTools PRO for Rhino
- *Ballpacking*: Triangle meshes are optimized such that the incircles of neighboring triangles touch tangentially. This leads to visually balanced meshes and allows to derive several architectural structures. See the scientific article "Packing Circles and Spheres on Surfaces" on Evolute's homepage for details.
- The optimization goals for the following values are only included in EvoluteTools PRO for Rhino:
  - *Planarity*: optimizes the mesh for planarity of its faces - obviously only faces with more than three vertices are included into this optimization term.
  - *Conical*: The mesh is optimized to become a conical mesh.
  - *Circular*: The mesh is optimized to become a circular mesh.
  - *Coplanarity*: Sets of vertices, which can be configured using the command etSetVerticesCoplanar, are optimized to become coplanar. Constraints on the plane used may apply, which can be specified using the same command.

## etOptionsReset

*Sets all optimization settings back to their default values.*

This command sets all values changed by etOptionsImportance and etOptionsToggles back to their default values.

## etOptionsToggles

*Allows editing of optimization settings.*

Using this command, settings for the mesh optimization command etOptimize of Evolutetools for Rhino can be adjusted. These settings can be reset to their default values using etOptionsReset.The settings in detail:

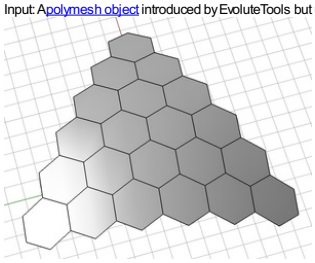
- *Fairing* (toggle):
  - absolute: Minimize curvature (FairnessCurvature) and curvature variation (FairnessCurvatureVariation).
  - relative: Minimize difference of curvature to curvature of current mesh (FairnessCurvature) and difference of curvature variation to curvature variation of current mesh (FairnessCurvatureVariation).
- *FairingMeasureScaleInvariant* (toggle, this setting applies to FairnessCurvature):
  - off: Use a scale-variant curvature measure, which causes smaller edge lengths in highly curved regions.
  - on: Use a scale-invariant curvature measure, which causes more equally distributed edge lengths.
- *FairingMeasurePerEdgeScaling* (toggle, only available in EvoluteTools PRO, this setting applies to fairing on strip meshes, around T-junctions and in other special circumstances. It should generally be switched on. ):
  - off: Every edge is treated equally, using the general fairing importances.
  - on: Individual scaling of the fairness term for single edges. This is used by several commands, among them by etSubdivide for its strip subdivision rule.
- *PlanarityMeasureScaleInvariant* (toggle, only available in EvoluteTools PRO, this setting applies to Planarity, see also Mesh Analysis):
  - off: Minimize diagonal distance of mesh faces.
  - on: Minimize diagonal distance divided by mean diagonal length of mesh faces.
- *Iterations* (integer):
  - How many iterations should be done by etOptimize. Caution: Increasing this value will cause etOptimize to take longer before returning control to you, but overall optimization time until convergence will decrease.
- *DefaultBoundaryCloseness* (toggle, applies to CurveCloseness):
  - on: All mesh boundary vertices will be flagged as curve points, see also etSetVertexCurvePoint. This will cause boundary vertices to snap to their closest reference curve. Furthermore all reference surface boundaries will automatically be added as reference curves.
  - off: Use etSetVertexCurvePoint to configure which vertices should snap to the closest reference curve. Reference surface boundaries will NOT automatically be added as reference curves.
- *CutOffSurfaceDistance* (number, applies to SurfaceCloseness):
  - Closeness optimisation will only be done for vertices which are within this distance from one of the reference surfaces.
- *CutOffCurveDistance* (number, only available in EvoluteTools PRO, applies to CurveCloseness):
  - Curve closeness optimisation will only be done for vertices which are within this distance from one of the reference curves.
- *IdealEdgeLength* (number, only available in EvoluteTools PRO, used in conjunction with the optimization importance of the same name):
  - Edge lengths are optimized to be close to this value if the respective importance setting is turned to a value larger than zero.
- *StripFairingScalesRatio* (number, only available in EvoluteTools PRO, used in conjunction with the strip subdivision rule):
  - The fairing importance strips (along the ruling direction) on a mesh edited with the strip subdivision rule is scaled down by this value, if *FairingMeasurePerEdgeScaling* is switched on. If this value is zero, a scaling ratio is computed using the average lengths of ruling and non-ruling edges.

## etPolymeshToNURB

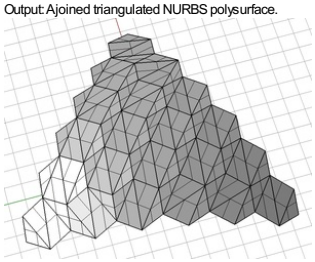
*Converts polymeshes (N-gon) meshes into joined triangulated NURBS surfaces.*

This command takes polymeshes (N-gon meshes) ,hybrid polymeshes (N-gons + triangles/quads), hybrid meshes (triangles+quads), or standard meshes as input (triangles or quads). The output is a joined triangulated NURBS polysurface, keeping the edge structure of the polymesh and adding new edges as a result of polygon triangulation. In case the mesh contains non planar quads these will not be triangulated, just transformed into non planar NURBS surfaces with 4 edges.

Input: Apolymesh object introduced by EvoluteTools but unsupported by Rhino.



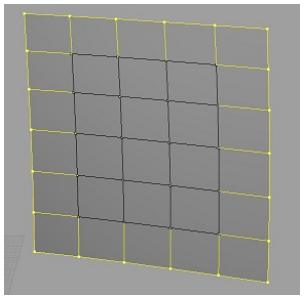
Output: A joined triangulated NURBS polysurface.



## etSelectMeshBoundary

*Selects all vertices on a mesh boundary.*

Quickly select all vertices on a single mesh boundary by clicking on a single boundary edge or vertex. Note that corner vertices divide boundaries, so if vertices along the boundary are flagged as corners, this command will only select the vertices between the nearest corner in each direction.



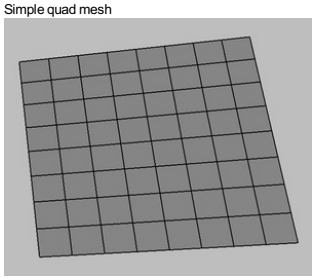
## etSelectMeshCorners

*Quickly select all corners of a mesh by clicking on a mesh, mesh edge or mesh vertex (polymeshes require clicking a vertex).*

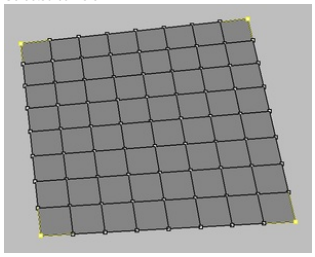
This command takes any mesh as an input.

---

Simple quad mesh



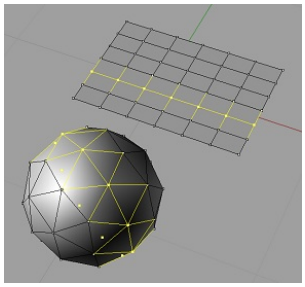
Selected corners



## etSelectMeshPolyline

*Selects all vertices on a mesh boundary.*

Quickly select all vertices along a single mesh polyline by clicking on a single edge on that polyline.

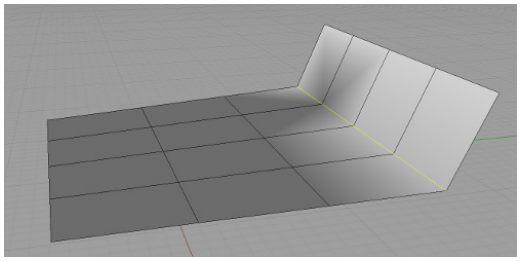


## etSetEdgeFeature

*Designates mesh edges as creases.*
*Only available in EvoluteTools PRO*

Using this command, mesh edges can be flagged to be treated as creases during optimization. This prevents any fairing across the flagged edge which permits sharp features within an otherwise smooth mesh. When invoked, the command highlights previously defined feature edges, which can be selected and unselected using several options:

- *SelectByVertices* this toggle allows selection of Polymesh edges by picking vertices one at a time.
- *SingleEdge* allows you to flag a single edge by selecting it.
- *Polyline* flags all edges in a polyline by selecting one edge.
- *ClearAll* unflags all edges.
- *AddAll* flags all edges.
- *Autodetect* prompts for an angle and automatically flags all edges with a dihedral angle larger than that value. All other edges are unflagged.



## etSetEdgeLengthOptimization

*Designates mesh edges to be optimized for Ideal Edge length.*
*Only available in EvoluteTools PRO*

Using this command, mesh edges can be flagged to be optimized for Ideal Edge Length. When invoked, the command highlights previously flagged edges, which can be selected and unselected using several options:

- *SelectByVertices* this toggle allows selection of Polymesh edges by picking vertices one at a time.
- *SingleEdge* allows you to flag a single edge by selecting it.
- *Polyline* flags all edges in a polyline by selecting one edge.
- *ClearAll* unflags all edges.
- *AddAll* flags all edges.
- *Autodetect* prompts for an angle and automatically flags all edges with a dihedral angle larger than that value. All other edges are unflagged.

## etSetReference

*Sets the reference objects for optimization.*
*Curves, points, and point clouds can only be set as reference in EvoluteTools PRO*

When this command is invoked, all objects currently set to be reference surfaces or curves are selected. Objects can be added or removed by selecting or deselecting them while the command is running. The selection is confirmed by pressing *Enter*. Permitted input formats are meshes, surfaces, polysurfaces, points, point clouds, as well as all types of curves and polylines. See etOptionsImportance for details on how the reference objects are used.

# etSetRulingDirection

*Defines the ruling directions for quad-mesh based strip-subdivision methods.*
*Only available in EvoluteTools PRO*

This command is used in conjunction with the etSubdivide command, more specifically with the *Strips* subdivision rule refining quad meshes in only one of two directions to create "strip models". Before using the strips subdivision rule on a quad mesh, you should use this command to define which of the two directions on the mesh should become the "ruling direction". This is done by selecting a single edge. The other direction will then become the strip direction, and its edges will be subdivided by the respective rule. Please refer to the images below for examples.

---

Image 1a: Selecting an edge using *etSetRulingDirection*

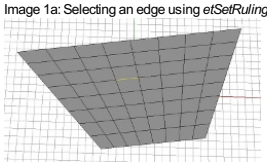


Image 1b: After a subdivision step using Strips subdivision

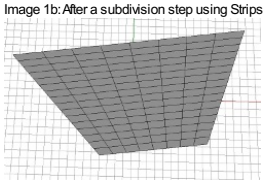


Image 1c: After a second subdivision step

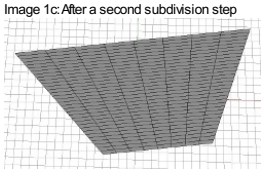


Image 2a: Selecting a different edge using *etSetRulingDirection*

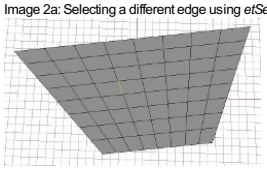
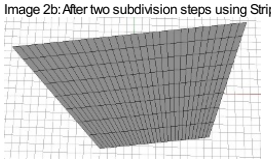


Image 2b: After two subdivision steps using Strips subdivision



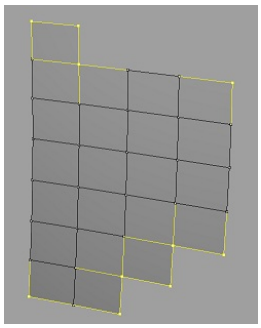
## etSetVertexCorner

*Designates mesh vertices as corners for optimization.*

During optimization, boundary mesh vertices flagged using this command will be excluded from all fairing optimization terms, preventing them from straightening the boundary around them. If a corner should remain at its exact coordinates during optimization, etSetVertexFixed should be used in conjunction with this command. The command has two options:

- AutoDetect: If true, EvoluteTools automatically selects boundary vertices according to the kink angle along the boundary at those vertices.
- AutoDetectKinkAngle: If AutoDetect is used, all boundary vertices with a kink angle higher than the value of this option will be selected.

If you need to unflag vertices previously flagged as corners, type 'etSetVertexCorner' or click its corresponding icon and select vertices while holding CTRL ( basic Rhino selection/deselection mode ).



## etSetVertexCurvePoint

*Designates mesh vertices as curvepoints for optimization.*
*Designating interior mesh vertices as curve points is only possible in EvoluteTools PRO*

During optimization, distances between mesh vertices flagged using this command and the nearest reference curve will be minimized, if the *CurveCloseness* optimization importance value is set. This can be used in conjunction with etSetReference to align interior mesh vertices along predefined curves. If *DefaultBoundaryCloseness* is switched on, all boundary vertices will automatically be designated as curvepoints.

## etSetVertexFixed

*Designates mesh vertices as fixed for optimization.*

Mesh vertices flagged using this command will not be changed by etOptimize.
If you need to unflag vertices previously flagged as fixed, type 'etSetVertexFixed' or click its corresponding icon and select vertices while holding CTRL ( basic Rhino selection/deselection mode ).

## etSetVertexRestrictedToNormal

*Restricts vertex movement to its normal.*
*Only available in EvoluteTools PRO*

This command takes vertices as input. When invoked, it highlights the previously selected vertices. Users can deselect vertices by using standard Rhino selection/deselection (holding Ctrl for deselection, holding Shift for selection).

Vertexes selected with this command will be drawn to move only along their approximated vertex normals during optimization, the strength of this constraint being defined by the importance setting *NormalCloseness*. A possible use case for this command is a mesh which already has the desired appearance, but panels which do not yet fully fulfill the planarity constraints. Setting *NormalCloseness* instead of fairing constraints and combining it with the planarity constraint will possibly improve the planarity of panels without significantly changing the appearance of the mesh.

## etSetVerticesCoplanar

*Designates mesh vertices to be optimized for coplanarity.*
*Only available in EvoluteTools PRO*

Using this command, sets of mesh vertices can be flagged to be optimized towards coplanarity. Additionally, constraints on the common plane can be specified.
When invoked, the command shows previously defined coplanarity sets, or prompts to select at least 3 vertices to create a new set.
The command has the following options:

- *Add* allows you to add another set of coplanar vertices.
- *Remove* purges the currently displayed set from the list.
- *Previous* lets you switch to the previous set of vertices.
- *Next* switches to the next set.
- *Planetype* lets you specify constraints on the common plane:
  - *GeneralPlane* will not add any constraints. The best fitting plane will be determined by the optimizer.
  - *ParallelToYZPlane*, *ParallelToXZPlane* and *ParallelToXYPlane*, will optimize the vertices towards a plane parallel to the respective coordinate plane.
  - *NormalToXYPlane*, *NormalToXZPlane* and *NormalToYZPlane* will optimize the vertices towards a plane normal to the respective coordinate plane.
  - *FixedPlane* lets you define more detailed constraints on the plane:
    - *SelectPlane* asks you to select a planar surface and will fix the plane to that planar surface.
    - *SelectPlaneNormal* asks you to select a planar surface and will fix the plane's normal to the normal of the selected planar surface (translations of the plane will still be allowed).
    - *HorizontalWithFixedZCoordinate* will constrain the plane to be horizontal, and asks you to specify the height of the plane. This is a shortcut for *SelectPlane* using a horizontal plane.
- *Importance* lets you specify the optimization importance value for the current coplanarity constraint. This importance value is multiplied by the importance value for Coplanarity.
- *ShowPlane* adds a planar surface patch to the document, showing the current plane used for optimization.
- *ShowDistancesFromPlane* adds text dots specifying the distances of the respective vertices from the current plane.

Vertex list coplanarity constraints will be preserved during changes to the connectivity of the mesh if possible. In case a vertex list coplanarity constraint consists of less than 4 (*GeneralPlane*), 3 (*NormalTo??Plane*), 2 (*ParallelTo??Plane*), or 1 (*FixedPlane*) vertices after changes to the mesh connectivity, it will be removed.

## etSetVerticesFairing

*Designates mesh vertices for additional fairness optimization.*
*Only available in EvoluteTools PRO*

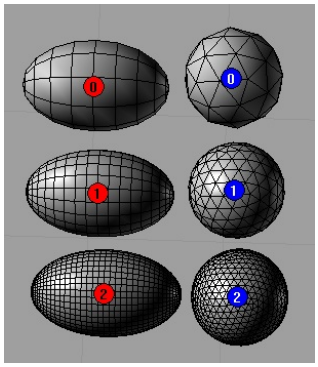
Using this command, lists of mesh vertices can be flagged to be optimized towards additional fairness (minimization of kinks). When invoked, the command shows previously defined lists of vertices, or prompts to select at least 3 vertices to create a new list.
The command has the following options:

- *Add* allows you to add another list of vertices for additional fairness optimization.
- *Remove* purges the currently displayed list.
- *Previous* lets you switch to the previous list of vertices.
- *Next* switches to the next list.
- *Importance* lets you specify the optimization importance value for the current list of vertices. This importance value is multiplied by the importance value for FairnessCurvature.

Vertex list fairing constraints will be preserved during changes to the connectivity of the mesh if possible. In case a vertex list fairing constraint consists of less than 3 vertices after changes to the mesh connectivity, it will be removed.

## etShowDependencies

*Visualizes the subdivision dependencies between meshes.*




This command displays text dots in front of all meshes with active subdivision dependencies. Dots of the same color designate a dependency, with the parent mesh being labeled with the number zero and all other meshes with integers according to the degree of relationship.

# etSubdivide

*Subdivides the selected mesh with one of the included subdivision alghorithms. The subdivided mesh is added to the document.*

This command bundles all available subdivision algorithms. After selecting a mesh, you are asked to select one of these algorithms before the subdivision is performed. In the following, the term *coarse mesh* or *parent* is used for the mesh selected for this command. The newly created mesh will be called the *subdivided mesh* or *child*. After the command has been called, these two meshes are logically connected in that all vertices of the subdivided mesh depend linearly on one or more vertices of the coarse mesh. If the coarse mesh is deformed or otherwise edited, the subdivided mesh will reproduce these alterations. By subdividing the subdivided mesh once again (using the same or any other alghorithm), a *subdivision chain* can be created, enabling *multiscale subdivision modeling*. Editing the subdivided mesh is only supported using the mesh editing tools provided by EvoluteTools, such as etMeshLoopCut, etMeshAddDiagonal, etMeshDeleteEdge, etMeshDeleteFace, etMeshDeletePolyline, etMeshDeleteVertex. Deleting vertices or faces from a subdivided mesh using Rhino's standard commands will automatically *decouple* the mesh from the coarse mesh.
Subdivision dependencies created by this command can be visualized using etShowDependencies.

#### Subdivision Algorithms

- *CatmullClark* (best used for quad meshes)
- *Loop* (only for triangular meshes)
- *Sqrt3* (best used for triangular meshes)
- *Diagonalize*
- *EdgeSplit*
- *Identity*
- *Dual*
- *DualEdge* *( Only available in EvoluteTools PRO )*
- *DualWithBoundary*
- *Strips* *( Only available in EvoluteTools PRO )*
- *TriHex*

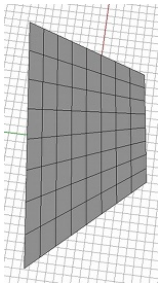
---

#### Combinatoric Examples

These simple examples are thought to explain the combinatoric meaning of the different subdivision rules. The "smoothing" capabilities of these rules are explained in a second set of more meaningful 3D meshes.

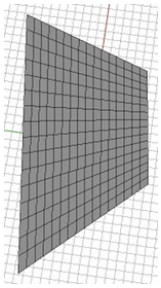
**Quad Example:**

Simple initial quad-mesh:



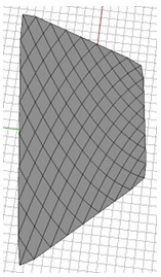
*Catmull Clark*

input: any mesh
output: always quad



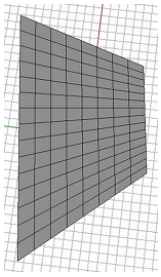
*Diagonalize*

input: any mesh
output: any mesh



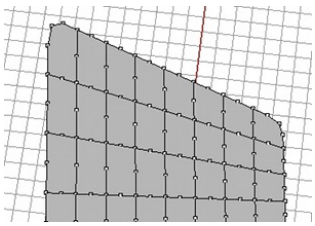
*Strips*

input: quad only
output: always quad



*EdgeSplit*

input: any mesh
output: always polymesh

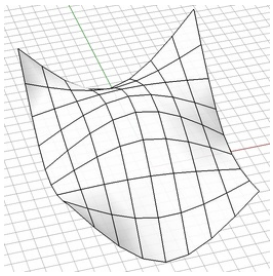


This is not a classic subdivision rule. It splits each edge into two, introducing one additional vertex per edge and always leading to the creation of a polymesh. This can be used to replace classic mesh edges with polylines, thus better approximating a surface without changing the basic connectivity.

---

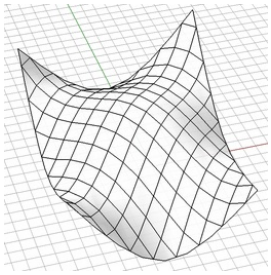
**Special subdivision - *DualEdge*:**

Simple initial quad mesh:

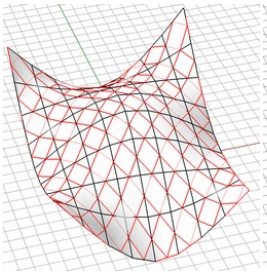


*DualEdge*

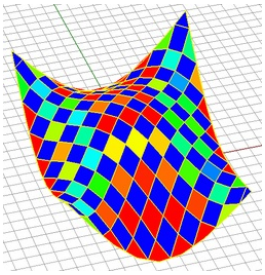
input: any mesh
output: hybrid (poly)mesh



subdivided mesh (red lines)
over the parent mesh



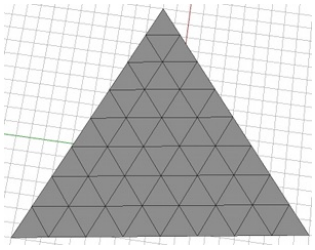
special properties after subdivision:
blue quads are planar



---

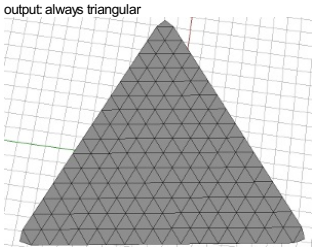
**Triangular Example:**

Simple initial triangular mesh:



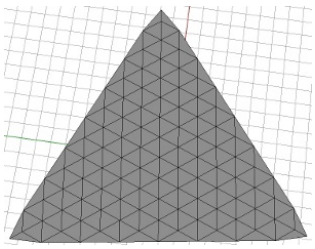
*Loop*

input: triangular only
output: always triangular



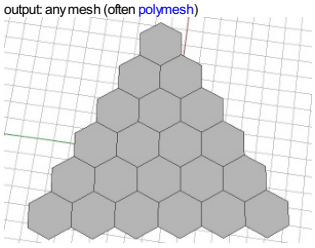
*Sqrt3*

input: any mesh
output: always triangular



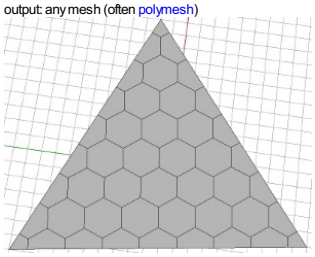
*Dual*

input: any mesh
output: any mesh (often polymesh)



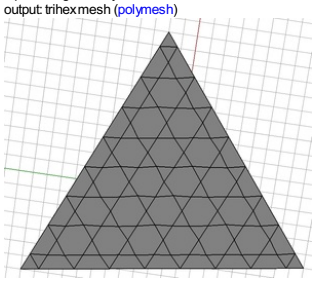
*DualWithBoundary*

input: any mesh
output: any mesh (often polymesh)



*TriHex*

input: triangular only
output: trihex mesh (polymesh)

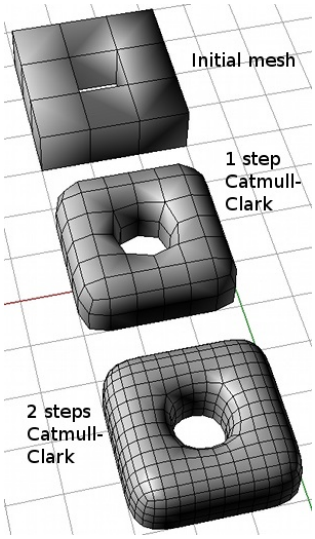


---

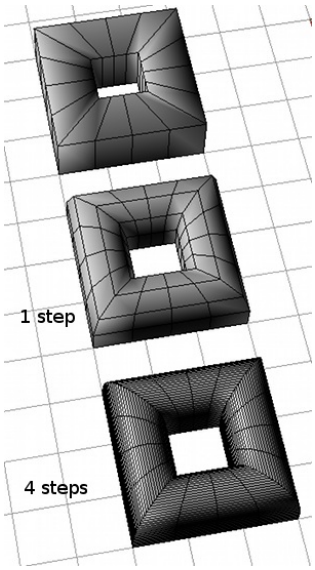
**3D Examples**

An attribute inherent in those subdivision rules which increase the number of vertices is that they automatically smoothen the input mesh. The following examples motivate this concept.

*CatmullClark*




*Strips*


1 step
4 steps

## etWeld

*Weld mesh vertices that are within a specified tolerance to each other, and consistently orient faces.*

The command asks you for a tolerance for welding. Vertices, that are closer to each other than this distance, will be merged. The resulting mesh will be an orientable manifold, which might result in some mesh faces being removed (see Moebius band example below). The plugins reports about the number of removed vertices and faces.

How to create a coarse mesh for use with etSubdivide:

1. Create mesh triangles and / or quads using Rhino's command "_3DFace".
2. Join triangles and quads using "_Join".
3. Use "etWeld" to merge vertices and consistently orient faces.

---

Image 1: Mesh consisting of 10 vertices and 3 faces before welding.

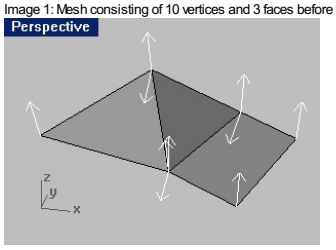



Image 2: Oriented mesh consisting of 6 vertices and 3 faces after welding.

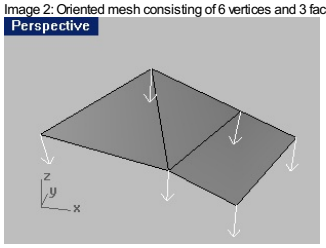



Image 3: Non-orientable mesh (a Moebius band) before welding.

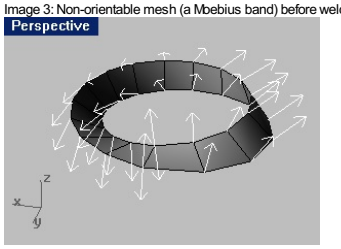
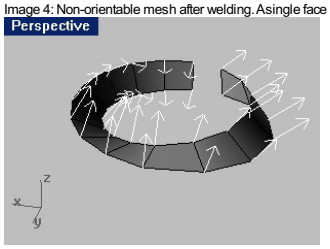



Image 4: Non-orientable mesh after welding. A single face has been removed.

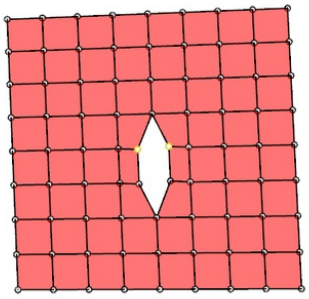


## etWeldVertices

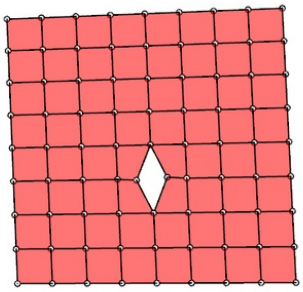
*Welds two mesh vertices picked by the user.*

This command takes two mesh vertices as input and outputs a single mesh vertex located halfway between the previously selected vertices. The command can also be used to collapse mesh edges.

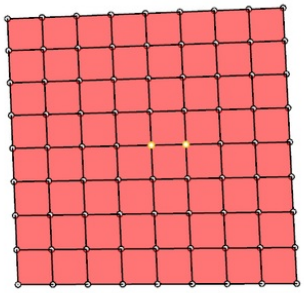
Simple quad mesh, selected vertices



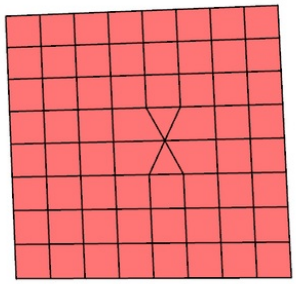
After vertex welding



Simple quad mesh, selected vertices



After vertex welding one edge collapses



## etpFit

*Creates best fit surface(s) for the selected input data (currently support input data: NURBS patch(es), mesh(es) ).* Only available in EvoluteTools PRO for Rhino, panel fitting module.

This command creates best fit surfaces corresponding to user selected NURBS patches or meshes, according to parameters configured by the user.

**Basic usage:**

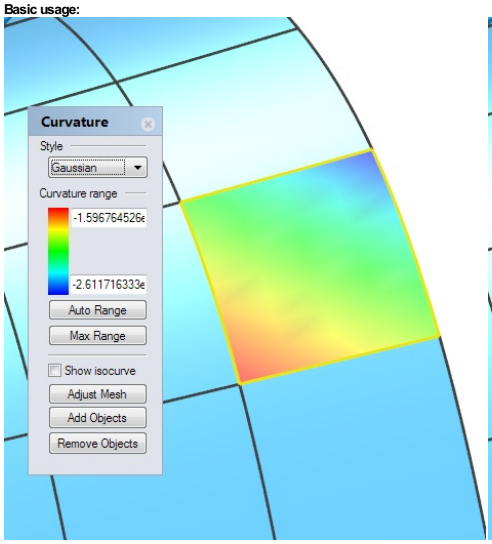



Fig.1 - double curved NURBS patch selected for fitting

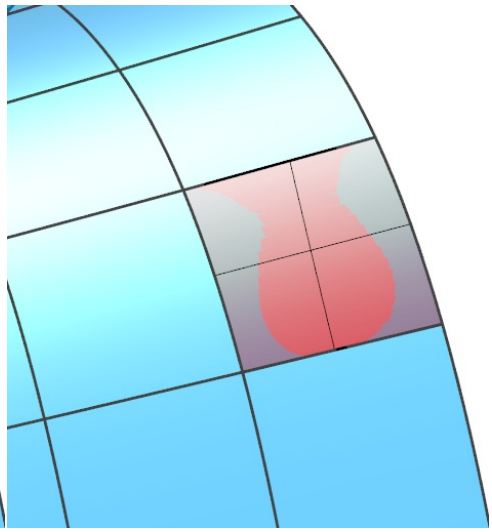
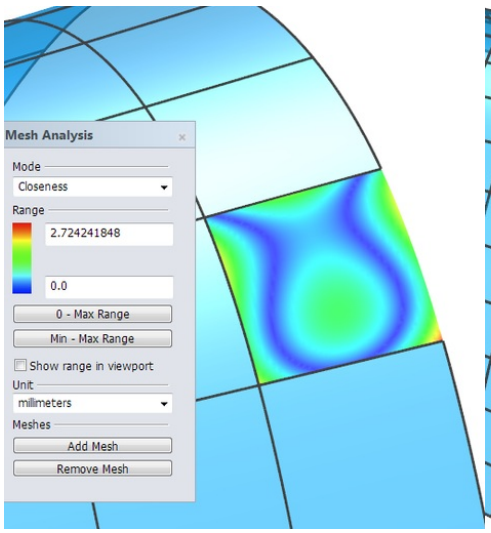
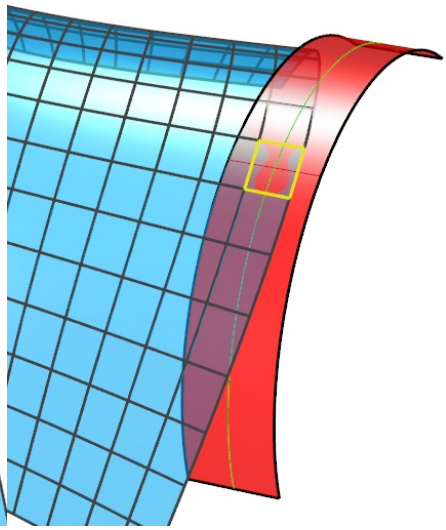


Fig.2 - cylinder fitting result (red)

- Select input data which should be used for fitting (fig.1).
- Call etpFit, which will compute a best fit surface according to the configured parameters for each of the input data in turn, and output the resulting best fit surface(s) (fig.2).
- Load EvoluteCoutHook.rhp prior to calling etpFit to see a progress bar.




Closeness analysis of the cylinder fitting.



Untrimmed cylindrical fitted surface.

Extensive parameters are available which can be configured using the following commands:

- *etpConfigInput*: parameters specific to the input data used by etpFit
- *etpConfigOptimization*: parameters specific to the optimization (fitting) used by etpFit
- *etpConfigOutput*: parameters specific to the output that should be generated by etpFit
- *etpConfigSolver*: parameters specific to the optimization solver used by etpFit
- *etpConfigTypes*: parameters specific to the surface types which are fitted by etpFit

**Supported input data:**

- *one or more NURBS patches, and / or*
- *one or more meshes*

In the near future the following input data will be supported as well:

- one or more curves (open or closed), and / or,
- point clouds

## etpConfigInput

*Allows the user to configure parameters specific to the input data for the etpFit command.* Only available in EvoluteTools PRO for Rhino, panel fitting module.

**Input parameters:**

- *BoundaryOnly (Y/N, default N):* Applies if input data is a NURBS patch or a mesh: If set to Yes, only the boundary curve(s) of the input data will be used for fitting.
- *CornersOnly (Y/N, default N):* Applies if input data is a NURBS patch or a mesh: If set to Yes, only the corner vertices of the boundary curve(s) of the input data will be used for fitting. Makes use of *SamplingMaxBoundaryKink* of etpConfigOutput.
- *MeshParamMaxEdgeLength (number, default 0):* Maximum edge length to use for sampling input data which is not already sampled (NURBS patches and curves). Will be ignored if it is 0.
- *MeshParamTolerance (number, default 0):* Tolerance to use for sampling input data which is not already sampled (NURBS patches and curves). If 0 the document absolute tolerance will be used.
- *MergeVertices (Y/N, default N):* Applies if input data is a NURBS patch or a mesh: If set to Yes, the vertices of the mesh used for fitting will be welded before fitting. In case the input data is a mesh, the document absolute tolerance will be used. In case the input data is a NURBS patch, the value of *MeshParamTolerance* applies.
- *AcceptDirtyMeshes (Y/N, default N):* Applies if input data is a mesh: If set to Yes, meshes which are not manifold will be accepted.
- *ComputeSamplingWeights (Y/N, default N):* If Yes, the local sampling density will be considered, in order to prevent biased fitting due to non-uniform sampling.
- *UseBrepNormals (Y/N, default Y):* Applies if input data is a NURBS patch: If set to Yes, the normals of the sampling mesh will be set using the normals of the NURBS patch.
- *ContinueOnError (Y/N, default Y):* If Yes, etpFit will not stop on error, but color the input data red, and attach the optimization log to the usertext "EV_LOG" of the input data.
- *OutputSampling (Y/N, default N):* If set to Yes, the sampling mesh will be output to the document as a separate object.

## etpConfigOptimization

*Allows the user to configure optimization parameters for the etpFit command.* Only available in EvoluteTools PRO for Rhino, panel fitting module.

**Optimization parameters:**

- *PointToPoint (number, default 0):* Optimization importance value to use for point-to-point fitting. Change only if you know what you are doing.
- *PointToPlane (number, default 1):* Optimization importance value to use for point-to-tangent-plane fitting. Change only if you know what you are doing.
- *NormalFitting (number, default 0):* Optimization importance value to use for normal fitting. Change only if you know what you are doing.
- *Centering (number, default 0.01):* Optimization importance value to use for panel centering. Change only if you know what you are doing.
- *Normalize (Y/N, default Y):* If Yes, the sampled input data will be normalized before fitting.
- *NonlinearInit (Y/N, default Y):* If set to yes, a non-linear initialization will be used before fitting, otherwise a faster but less accurate linear initialisation is used.

## etpConfigOutput

*Allows the user to configure parameters specific to output data of the etpFit command.* Only available in EvoluteTools PRO for Rhino, panel fitting module.

**Output parameters:**

- *Trim (Y/N, default Y):* If Yes and the input data is a NURBS patch or a mesh, the resulting best fitting surface will be split using the boundary curve of the input data. See also *MaxPanelPieces*.
- *MaxPanelPieces (number, default 1):* After splitting, the resulting surface patches will be deleted by decreasing surface area until this number of surface patches is left.
- *ShrinkDistance (number, default 0):* If > 0, the boundary curves of the resulting surface patches will be offsetted inwards by this distance.
- *NativeOrNurbs*
  - *Native (default):* If available for the selected panel type, use a native surface representation (e.g. a surface of revolution for right circular cylinders and cones).
  - *Nurbs:* Use a NURBS representation.
- *PanelSizeX (number, default 1):* If *NativeOrNurbs=Nurbs*, length of the NURBS patch to create.
- *PanelSizeY (number, default 1):* If *NativeOrNurbs=Nurbs*, width of the NURBS patch to create.
- *SamplingMaxBoundaryKink (number, default 15°):* Kink angle to use for detecting corners along the boundaries.
- *PanelSplitMethod*
  - *PulledBoundaryCurve (default):* Split the best fitting surface using the input data boundary curve pulled to the best fitting surfaces.
  - *NormalLoft:* Split the best fitting surfaces using a ruled surface generated by the normals along the input data boundary curve.
- *PanelPullSplitMeshBoundary*
  - *InterpolateSmooth (default):* If the input object is a mesh, create a smooth curve interpolating the boundary curve vertices before splitting.
  - *Polyline:* If the input object is a mesh, use the polyline boundary curve for splitting.
- *TransformToCanonicalPosition (Y/N, default N):* Not implemented yet, if Yes create copy of best fitting surface transformed to its canonical position. .
- *KeepUntransformed (Y/N, default Y):* If No, untransformed best fitting surfaces will not be created.
- *Redraw*
  - *End (default):* Do not redraw viewports after each surface fitting, but only at the end of the procedure.
  - *Immediate:* Redraw viewports after each surface fitting.

## etpConfigSolver

*Configures parameters specific to the solver for the etpFit command.* Only available in EvoluteTools PRO for Rhino, panel fitting module.

This command allows the user to configure parameters specific to the solver for the etpFit command.

**Solver parameters:**

- *MaxIterations (number, default 100):* Maximum number of iterations.
- *StopResidual (number, default 1e-18):* Stop optimization if residual is lower than this value.
- *StopResidualRelative (number, default 1):* Stop optimization if relative change of residual is bigger than this value.
- *StopStep (number, default 1e-9):* Stop optimization if maximum norm of update vector is smaller than this value.
- *LineSearchSteps (number, default 50):* Maximum number of steps for line searching.
- *DampingFactor (number, default 1e-12):* Damping factor (regularization).
- *RequireConvergence (Y/N, default Y):* If Yes and the optimization does not converge (i.e. none of the stopping criteria listed above is achieved), consider this to be an error.

## etpConfigTypes

*Allows the user to configure fitting surface type parameters for the etpFit command.* Only available in EvoluteTools PRO for Rhino, panel fitting module.

**Surface type parameters:**

- *RigidFit (Y/N, default N):* Currently only applies to surface type CircularCylinder. If Yes, the cylinder radius will not be optimized for.
- *CylinderRadius (number, default 1000):* Fixed radius to use for cylinder fitting.
- *DefaultPanelType*
  - *CircularCone:* Right circular (rotational) cone. Use with caution, not very stable yet.
  - *CircularCylinder (default):* Right circular (rotational) cylinder.
  - *CubicCylinder:* Ageneral cylinder generated by a cubic polynomial.
  - *Cubic:* Ageneral cubic polynomial surface patch.
  - *DoubleCylinder:* Atangentially smooth connection of two right circular cylinders. Use with caution, not very stable yet.
  - *Flat:* Aplane.
  - *Paraboloid:* Asimple translational surface (a parabola translated along another one).
  - *PolynomialCylinder:* Ageneral cylinder generated by a polynomial (specify the degree using *PolynomialDegree*).
  - *Polynomial:* Ageneral polynomial (specify the degree using *PolynomialDegree*).
  - *QuarticCylinder:* Ageneral cylinder generated by a quartic polynomial.
  - *Quartic:* Ageneral quartic polynomial surface patch.
  - *RuledCubic:* Aruled cubic polynomial surface patch.
  - *RuledPolynomial:* Aruled polynomial patch (specify the degree using *PolynomialDegree*).
  - *RuledQuartic:* Aruled quartic polynomial surface patch.
- *PolynomialDegree (number, default 5):* Degree of polynomial to use for the polynomial surface types.

# etPolygonsCluster

*Creates groups of repeatable panels within certain specified tolerances.* Only available in EvoluteTools PRO for Rhino, flat panel clustering module.

This command creates clusters of best fit polygons, within the tolerances specified by the users, maximizing the repeatability of the best fit polygons. It takes any number of convex polygons as an input (triangles, quads, n-gons, inputs must be closed convex polylines). If the input polylines are not planar, the command will project the polyline on its best fit plane and use that projection as an input. The command output groups input polylines that have a best fit polygon and shows the best fit polygon in a 2D matrix.

**Polygon clustering parameters:**

- *ToleranceOutside*: tolerance for increasing polygon size (shown in magenta on the 2D matrix).
- *ToleranceInside*: tolerance for decreasing polygon size (shown in magenta on the 2D matrix).
- *Alignment*:
  - *0: only translations of input polygons allowed (no rotations)*
  - *1: rotations allowed*
- *NamePolygons*: Name input polygons (Polygon 1, Polygon 2, ...). Same numbering as shown in cluster output (see below).
- *CreateGroups*: Group input panels according to clusters.
- *ShowClusters*: Show clusters in XY-plane.
- *ShowAllClusters*: Show ALL clusters in XY-plane (also the ones which have not been selected by the set-covering).
- *ShowClusterLeaders*: Show how the cluster representatives fit into the tolerance zones (blue polygons in the 2D matrix and 3D space).
- *CreateDots*: Creates grouped numbered dots corresponding to the respective polygon clusters in 3D space (toggle Yes/No).

---

**Installation notes:**

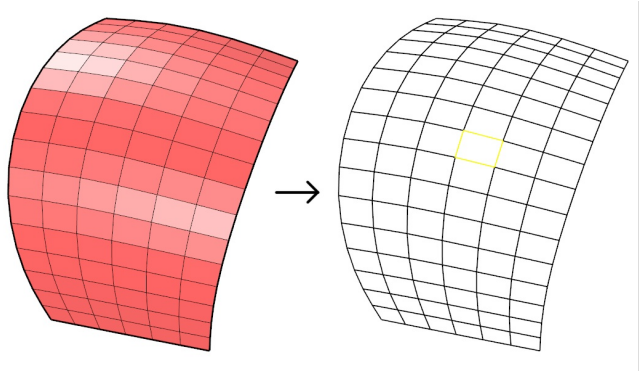
To install the polygon clustering plugin, copy the content of the delivered archive to your preferred location and drag the *EvoluteClusteringPlugin.rhp* and *EvoluteCoutHook.rhp* to the Rhino window. The *etPolygonsCluster* command should be immediately available after inserting and validating your license key. The *EvoluteCoutHook.rhp* is not necessarily required, however, for larger input sets the command might run for a longer period and the *EvoluteCoutHook.rhp* will provide a handy progress bar to monitor the job duration.

---

**Typical usage:**

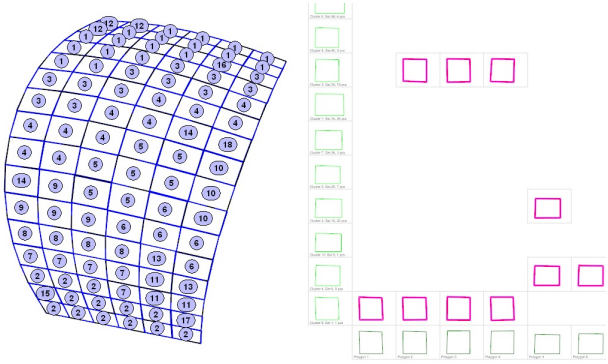
Before attempting any polygon clustering, there are a few prerequisite steps to follow, in order to ensure you have a good input set for the clustering command. Only **convex** polygons can be used as inputs.

**Preparing your inputs**: typically, your input polylines (triangles, quads, n-gons) result from an optimized mesh obtained with EvoluteTools PRO. Once you have your mesh ready, you must extract the boundary polylines of the mesh faces in order to use them as inputs for the *etPolygonsCluster* command. It is important to note that, unless your mesh is made of triangular faces, the constituent mesh faces should be as close to planar as possible, otherwise the clustering command will project the unplanar polygons on their best fit plane and use that as an input instead. To extract the face boundaries as polylines, you must use *etMeshExtractPolylines* with the *Mode* toggle set to *Faces* (Mode=*Faces*). You can hide the input mesh to better visualize the polyline extraction (below, optimized mesh on the left, extracted face polylines to the right).



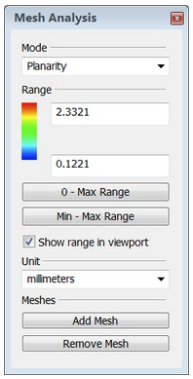
It is advisable to optimize the mesh for goals that will make it more suitable for polygon clustering, some of these optimization goals could be: *IdealEdgeLength*, *IdealPanel*, *Conical*, *Circular*, *Ballpacking*. Although mesh optimization can be carried out with EvoluteTools PRO, it is not necessary to create the input polygons with EvoluteTools, as any collection of convex polygons can be used as an input, whether they lie in 3D space or the XY Plane. Once the inputs are prepared, the next step is to attempt the polygon clustering.

**The polygon clustering**: after running *etPolygonsCluster* and selecting the desired inputs, the clustering parameters must be set at the command line prompt. The *Tolerance* parameters dictate how close (Inside and Outside) the best fit polygons must be to each input polygon. These tolerances are usually dictated by manufacturing/construction techniques, fixing systems and visual appearance. If the facade panels are supposed to have gaps between them, the gaps have to be taken care of before running the polygon clustering, by offsetting the panel boundaries to the required distances. Once the tolerances are set and the rest of the parameters checked (see parameter description above), pressing *Enter* or the *Space Bar* will initiate the polygon clustering and fitting. The results will be displayed in the Rhino window, they consist of a 2D matrix of polygons and the best fit polygons fitted in 3D space over the collection of input polylines, along with the numbered dots and grouped objects to better identify the clusters of repeatable polygons (below, best fit polygons in 3D space -left, 2D matrix -right). The optional 2D matrix of polygons consists of all the polygon inputs (dark green) on the lower row, the polygons clusters (light green) on the right column, and the corresponding best fit (blue) and cluster polygons leaders (red) infield, along with the tolerance offsets (magenta). The 2D matrix will be drawn with the lower right corner anchored at the origin.



# Mesh Analysis

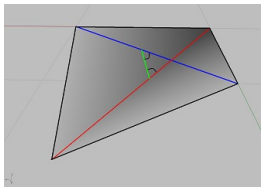
Mesh Analysis provides functionality to display geometrical properties of meshes using color coding. It is started for selected meshes using the commands etAnalyzeCloseness or etAnalyzePlanarity.




**Options**

**Mode**

- Closeness: Visualizes closeness of mesh vertices to the reference surface(s), see also etSetReference.
- Edgelengths: Visualizes the panel edge length distribution across the mesh. Faces are colored according to their maximum edge length.
- Planarity: Visualizes planarity of individual mesh faces. Planarity for quad faces is measured using the so-called diagonal distance. This corresponds to the closest distance between the two diagonals of a quad. Planarity for n-gons (arbitrary polygons) is measured by the sum of the diagonal distances for each inscribed quad consisting of 4 consecutive vertices (i.e. not all inscribed quads are considered), divided by the face valence (the number of vertices).



- Planarity Scale Invariant: Like Planarity, but the diagonal distances are divided by the mean length of diagonals. This results in a scale invariant planarity measurement. For n-gons (arbitrary polygons), each diagonal distance is individually divided by the mean diagonal length of the corresponding quad before summation and division by the face valence.

**Range**
Lets you specify the minimum (blue) and maximum (red) values for color coding. All measurements are clipped using these values.

**0 - Max Range**
Sets the minimum clipping value (blue) to 0 and the maximum clipping value (red) to the maximum measurement occuring in all meshes for which Mesh Analysis is activated.

**Min - Max Range**
Sets the minimum clipping value (blue) to the minimum measurement and the maximum clipping value (red) to the maximum measurement occuring in all meshes for which Mesh Analysis is activated.

**Units**
Drop down menu allows selection of prefered displayed units, from metric to imperial and even molecular.

**Add Mesh**
Activates Mesh Analysis for currently selected meshes.

**Remove Mesh**
Deactivates Mesh Analysis for currently selected meshes.

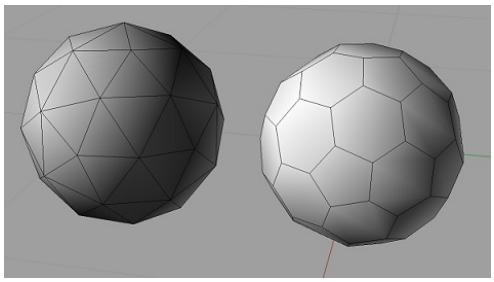
**Show range in viewport**
If checked, it will display the range, units and color bar in the top left side of the Rhino viewport for easy capturing.

**Hints**

Color coding is updated as you update the meshes for which Mesh Analysis is activated. Mesh analysis can be deactivated using etAnalyzeOff or by closing the Mesh Analysis dialog.

## Polymeshes (Experimental)

*A preliminary generalization of Rhino Mesh Objects.*

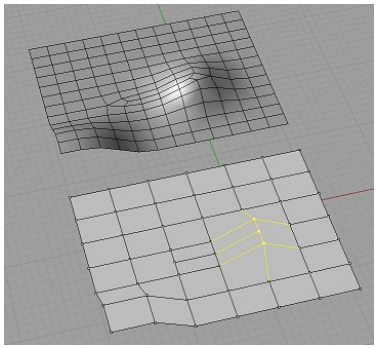


Polygonal mesh objects in Rhino are currently limited to allow only triangular and quadrilateral faces. This represents a significant limitation to working with subdivision algorithms and especially to the creation of dual meshes. The dual mesh of a regular triangular mesh, to give an example, is a hexagonal mesh (see image above), which can not be displayed as a standard mesh object in Rhino.

Satisfactorily solving this problem will involve the development of an entirely new type of mesh object for Rhino. This is a long-term project of our development team which we hope to introduce in a future release of EvoluteTools.

The solution we have included into this release allows you to create general polygonal meshes using a number of our commands like etMeshCut or etSubdivide. These objects look very similar to standard Rhino mesh objects, however they will not exhibit the same behaviour. The main limitations to these objects are

- Editability:
  - Not working with polymeshes:
    - Advanced mesh editing tools included in Rhino (like welding, trimming or repairing meshes) will generally not work with our polymeshes.
    - Selecting mesh edges.
    - Selecting mesh faces.
  - Working:
    - Dragging vertices
    - Dragging and otherwise moving the mesh
    - The EvoluteTools commands etMeshAddDiagonal, etMeshAddDiagonalLine and etMeshAddDiagonalLinesParallel as well as etSelectMeshBoundary, etSelectMeshPolyline and etMeshExtractPolylines.
    - etSubdivide. This is a main advantage of having polymeshes in EvoluteTools for Rhino: in some cases, subdividing a polymesh even results in a standard mesh (for example when using the CatmullClark algorithm). The following image shows such a situation, where inserting edges by using etMeshCut resulted in a polymesh but one subdivision step leads back to a standard quad mesh.



- Storage:
  - Working:
    - Import & Export as .obj file format via the Import and Export tools introduced in EvoluteTools PRO 2.0.
    - Saving in Rhino document files. If opened by an instance of Rhino with EvoluteTools installed, documents including polymeshes will display them correctly. If EvoluteTools are not present, polymeshes will appear without shading and proper behaviour.
    - Another way to export polymeshes into 3rd-party software is currently in the form of polylines. You can use etMeshExtractPolylines to create those and use Rhino to export them into many different storage formats.

Rhino will treat polymesh objects as invalid geometries and display incorrect object properties. Generally, while we encourage you to use this feature to explore the possibilities of working with general polygonal meshes, we have to explicitly state that this part of EvoluteTools is, as of now, totally experimental and will not work properly in conjunction with most other features of Rhino.

### Release notes for EvoluteTools for Rhino

***v2.3, 2013/11/06***

- New modules are available:
  - Developable lofting
  - Panel fitting
  - Panel clustering
- Helpfile update.

***v2.2.4, 2013/09/25***

- Helpfile update.
- etExtractConjugateLines now also works for flat surfaces.
- Added scripting toolbar function etsNurbsFromQuadMesh.
- Bugfixes:
  - Fixed a crash bug in etOptimize which occurred randomly for the 64bit version.

***v2.2.3, 2013/09/03***

- Helpfile update.
- Added scripting toolbar function etsAddPolyFace.
- Added options for curvature estimation to etExtractCurvatureLines and etExtractConjugateLines.
- etSetReference and et.SetReference now allow to specify points and point clouds as reference objects.
- Bugfixes:
  - et.MeshGetMaxPlanarityError sometimes returned outdated values.
  - etSetReference and et.SetReference did not work for certain curves.
  - et.EmAddPolyMesh did not do proper input data checking.
  - et.EmAddVertex did not do proper input data checking.

***v2.2.2, 2013/08/14***

- Optimization speed improvement
- New RhinoScript interface functions et.OptionsReset, et.OptionsImportance, et.ReferenceMaximumSamplingDistance.
- Additional parameters for RhinoScript interface function et.MeshGetMaxClosenessError.

***v2.2.1, 2013/08/05***

- Bugfixes:
  - etSetVerticesFairing and etSetVerticesCoplanar caused incompatibility between files saved with 32bit and 64bit versions
  - Scripting function etMeshGetMaxPlanarityError now gives approximate error values for polymeshes
  - Scripting function etMeshGetMaxClosenessError now returns NULL when no reference surface is set

***v2.2.0, 2013/06/18***

- Improved edge-length analysis display
- Added Rhino 5.0 64bit compatibility
- New toolbar with scripting functionality
- etAnalyzeCloseness now also works for reference curves
- Bugfixes:
  - etExtractCurvatureLines did not take care about transformations
  - Fixed some memory leaks in the RhinoScript interface
  - Fixed possibility of a crash, if a polyline with duplicate vertices is set as reference curve
  - etSetReference did not take care about transformations in all cases
  - Transforming a reference object did not update background reference data in all cases
  - Importer for obj files could not handle line breaks
  - Fixed rare planarity optimization crash

***v2.1.0, 2012/10/02***

- Added RhinoScript commands for setting coplanarity constraints: et.SetVerticesCoplanar, et.DeleteCoplanarityConstraint
- Added RhinoScript commands for setting fairing constraints: et.SetVerticesFairing, et.DeleteFairingConstraint
- Added documentation for RhinoScript command et.SetVertexCloseToNormal
- Added RhinoScript command for setting and querying ball packing radii: et.VertexBallpackingRadius
- Improved sampling of reference curves
- etSetReference: Added a warning message in case of low absolute document tolerance compared to reference object size, which could cause Rhino to freeze during meshing. The warning message is displayed if the command is run in interactive mode, and the bounding box diagonal of the object is longer than 100000 times the absolute document tolerance.
- Bugfixes:
  - Analysis modes: Planarity analysis and edge length analysis were displaying wrong values in case the mesh had been scaled previously
  - fixes to documentation
  - Bugfix in et.EmAddVertex which caused a problem in with fixing vertices using et.SetVertexFixed in the scripting example CreateRuledStripsFromQuadDominantMesh
  - Fixed sampling of curves in etSetReference, which sometimes caused crashes due to an excessive number of sampling points
  - SelectByVertices now works correctly in etMeshDeletePolyline
  - Edge flags which have for example been set using etSetEdgeFeature or etSetEdgeLengthOptimization are now preserved during mesh editing operations which cut edges, e.g. etMeshCut and etMeshLoopCut
  - Fairing constraints which have been configured using etSetVerticesFairing will now be preserved properly during mesh edit operations. Formerly this was only the case for coplanarity constraints.
  - Fixed unexpected behavior of vertex selection / deselection in all commands for flagging vertices.
  - Fixed strange behavior in et.PickVertex, et.PickFace and et.PickEdge if called several times in a row.
  - Fixed a bug which could cause et.EmGargabeCollection to drop flags and constraints.

***v2.0.5, 2012/08/06***

- EvoluteTools for Rhino now working with Rhino 5 32bit.

***v2.0.4, 2012/06/04***

- Added command etExtractConjugateLines.
- Enhanced command etExtractCurvatureLines.
- Bugfixes:
  - Fixed problem with loading locally stored licenses if internet connection is down.

***v2.0.3, 2012/05/04***

- Added RhinoScript command et.Optimize, which allows calling optimization and getting back detailed optimization results.
- Bugfixes:
  - Fixed timezone-related problem with EvoluteTools Lite registration.
  - Fixed bug which caused floating licenses to be given back before last instance of plugin was unloaded.

***v2.0.2, 2012/03/24***

- Bugfixes:
  - Fixed crash which could occur in etSetRulingDirection or etSubdivision with Rule=Strips.
  - Fixed minimum / maximum value checking in analysis widget, which could trap cursor in widget.

***v2.0.1, 2012/02/20***

- etSetEdgeLengthOptimization allows localized control of the IdealEdgelength optimization, the user can pick which edges will be optimized.
- Bugfixes:
  - AWindows 7 security feature prevented the license to be transfered to our server after closing Rhino, this is fixed.
  - The IdealEdgeLength optimization did not initiate correctly, this is fixed.

***v2.0, 2012/01/30***

- etMeshLoopCut now also cuts triangle faces if they are both at the end of a loop and on a mesh boundary.

- Added new subdivision rules: Trihex (creates a hybrid polymesh consisting of triangles and hexagons) and DualEdge (the result is a hybrid mesh with special properties).
- *Added a RhinoScript interface, including most of the core plugin functionality as well as very powerful and efficient access to meshes and their elements using the concept of halfedge structures.*
- The etMeshAddDiagonalLine and etMeshAddDiagonalLinesParallel commands have been improved to work on hybrid meshes. One can therefore now apply them to a quad mesh in both directions consecutively.
- The plugin will now load the complete Toolbar when loaded for the first time.
- Polymeshes will now be shaded in appropriate Colors.
- Added import and export tools for N-Gon meshes (only available in the PRO version).
- Coplanarity now accepts as fixed planes any Rhino surface flat within document tolerances, e.g. planes created by extruding a line segment.
- Optimization parameter importance settings are now saved in the Rhino file, they can be loaded for new worksessions.
- Joining meshes which contain EvoluteTools userdata will no longer unselect them. Therefore etWeld will now recognize the joined object if called directly after "Join"
- In the mesh analysis modes, one can now choose the unit of the analysis range.
- Analysis color coded legend and range can now be baked into the Rhino viewport via a checkbox in the Analysis widget.
- New commands:
  - etPolymeshToNURB convert Polymeshes to triangulated NURBS polysurfaces.
  - etWeldVertices welds two user picked vertices.
  - etMeshTriangulateNGons splits all NGons found in a mesh into triangular faces.
  - eMeshToPlanarPanels creates NURBS planar panels with gaps from a watertight non-planar quad mesh or polymesh.
  - etMeshSplitFace splits quadrangular or triangular faces in smaller triangles starting from user picked points.
  - etMeshRemoveNGons deletes all N-gons from a hybrid polymesh.
  - eSetVerticesFairing allows localized fairing control.
  - etMeshOffset offsets any mesh with contant spacing and adds connection elements.
  - etExtractCurvatureLines allows analysis and extraction of principle curvature lines of any mesh or NURBS polysurface.
  - etSetVertexRestrictedToNormal is a new command that enables restricted vertex movement to its normal, restriction importance is controlled by the NormalCloseness parameter.
  - etMeshPatternMapper maps points, polylines or meshes from a domain mesh to a target mesh.
  - etBallPackingMeshExtractBalls is a new command for extracting the spheres of a ball packing mesh.
  - etSetEdgeFeature a new command enables users to define creases, this turns off fairing across the crease and allows sharp features in the panel layout.
- Bugfixes:
  - fixed unexpected behaviour of some vertices created by the subdivision rule "DualWithBoundary". Attributes like "fixed", "corner" and others will now be kept where possible.
  - in some cases it was impossible to select a set of vertices to become coplanar, this is fixed.
  - tri-hex meshes didn't inherit flags properly, this is fixed.
  - ruling directions were not preserved after a Catmull-Clark subdivision, this is fixed.
  - extracting polylines from a tri-hex mesh could potentially crash Rhino, this is fixed.
  - attempting to join a polymesh with a standard mesh could potentially crash Rhino, this is fixed.

***v1.1.2, 2011/08/05***

- etMeshExtractPolylines: split boundary curve at vertices flagged as corner.
- Bugfixes:
  - pressing ESC while selecting new reference surfaces using etSetReference no longer removes all current reference surfaces
  - using the LOOP subdivision rule after the CATMULL-CLARK rule produced unexpected results and potentially crashed Rhino.
  - etSubdivide with rule Strip: keep fixed vertices fixed.
  - etMeshExtractPolylines: fixed crash for polylines consisting of only 1 edge.

***v1.1.1, 2011/05/09***

- Display detailed version information when loading plugin.
- Documentation for *FairnessSprings* was missing.
- Improved usability of floating license configuration (KeepLocal=Yes or KeepLocal=No).
- Bugfixes:
  - Plugin caused Rhino to crash in certain configurations, when joining surfaces or curves
  - etSetVerticesCoplanar: setting a fixed plane or showing a fitted plane did not take care of global mesh transformations exactly
  - etLicenseManager: Transfer of a floating license acted like GiveBackFloating and did not delete the serial number from the local list.
  - relation between subdivision parent and child was not taken care of in some cases (when using a mesh editing operation before the first subdivision)
  - etSetReference may fail if Rhino's analysis mesh generation produces isolated vertices. Added a workaround.
  - Using a relatively high document absolute tolerance could crash etWeld.
  - Fixed a crash after transfer of license.

***v1.1, 2011/04/06***

- Added support for floating licenses via an online license server, please see etLicenseManager for details. The floating license feature requires Internet access and correctly configured network / proxy settings for Internet Explorer. Please contact support@evolute.at if you would like your node-locked license to be changed to a floating license.
- Added command etWeld for simplifying the creation of coarse meshes for use with etSubdivide.
- Mesh analysis commands now support PolyMeshes.
- etSetVerticesCoplanar can now be used to constrain vertices to a given plane surface, or to planes parallel to a given plane surface.
- etMeshCut and etMeshLoopCut now support vertex selection, enabling their use for PolyMeshes.
- New commands etMeshDeleteEdge, etMeshDeleteFace, etMeshDeletePolyline, etMeshDeleteVertex.
- New subdivision rules for creating strip models from quad meshes, see etSubdivide and etSetRulingDirection.
- Bugfixes:
  - correctly apply subdivision history after child has been recreated
  - etSelectMeshPolyline: could not select boundary of triangle mesh
  - etAnalyzePlanarity, etAnalyzeCloseness: corrected unwanted behavior if maximum value was set to 0
  - etSetVerticesCoplanar: wrong plane was shown after optimization
  - etOptimize: fixed crash if coarse mesh was hidden
  - all commands: complain if mesh is inconsistent (causing crashes during later calls to etOptimize or etSubdivide)
  - etMeshExtractPolylines: take care of transformation
  - etLoopCut: new vertices were not updated after change of coarse mesh
  - etMeshCut: fixed crash when used with single quad
  - etLicenseManager: bug in reading proxy settings
  - etMeshExtractPolylines: transformation was not taken care of

***v1.0, 2010/11/17***

- initial release

# EvoluteTools for Rhino Scripting

***Only available in EvoluteTools PRO***

**Syntax:**

```
Dim et
Set et = Rhino.GetPlugInObject( "EvoluteTools for Rhino" )
```

The EvoluteTools for Rhino Scripting Interface is an effort to make as much as possible of our core functionality accessible for automation using RhinoScript. As many of EvoluteTools' features involve handling individual vertices, edges and faces of a mesh or groups of such elements, there is an apparent need for an efficient way of identifying, accessing and navigating between those elements. EvoluteTools uses a data structure called **Half-edge Structure** for these tasks in its mesh handling operations, and the scripting interface will allow direct access to its powerful features.

Therefore, this interface consists of two different types of RhinoScript commands: Those allowing low-level access to functions of the half-edge structure like finding, adding or deleting edges, faces and vertices have names including the prefix **"Em"** (for Evolute mesh), like *EmNextHalfedge*. The other group consists of those commands which mirror the functionality of higher level Rhino commands provided by EvoluteTools. These are named identical to their counterparts wherever possible.

# Half-edge Structures

## Basics

The halfe-edge structure is a data structure designed to efficiently store and manipulate, among other objects, all meshes of the type used within Rhino. The building blocks of a half-edge structure are half-edges, vertices, edges and faces.

The half-edges are directed edges, 2 for each edge, which are oriented opposedly. Each halfedge has a start vertex and an end vertex. If halfe-edge H has vertices Aand B as its start and end vertices, respectively, then its opposite halfedge has B as its start vertex and Aas its end vertex. Additionally, each half-edge does have a predecessor (or previous halfedge) and a successor (the next halfedge). These links do always form a loop, so each halfedge is associated to some loop. Every loop belongs to one of two groups: it either encloses a mesh face or a hole. Loops in the latter group are called boundaries of the mesh, and all half-edges within such a loop are referred to as boundary half-edges.

## Fast Queries

Within a half-edge structure, certain information about neighboring elements is saved with each half-edge, vertex, edge and face. This allows the following operations to be performed very efficiently:

- Given an edge, find one of its half-edges
- Given a vertex, find a half-edge connected to it (having the given vertex as start vertex)
- Given a face, find a half-edge connected to it
- Given a half-edge, find the next halfedge, its end vertex or the face associated with it.

All other querying tasks can be implemented using these very basic operations. Alarge number of common ones are already included as commands in the EvoluteTools Scripting interface, more specialized ones can easily be scripted. Two short examples are motivated in the following.

## Examples

- Given a face, find all edges associated with it:
    1. Find one associated half-edge (see above)
    2. Find next halfedge
    3. Find its associated edge
    4. Repeat step 2 and 3 until you reach the first half-edge again
- Given an edge of a quad mesh, find all edges in its grid polyline:
    1. Find one of the associated half-edges
    2. If the end vertex of this halfedge has 4 associated edges (to check that, see example 1), find the next halfedge in the polyline by
        1. finding the next half-edge,
        2. then finding that halfedges opposite halfedge
        3. and again proceeding to that ones next halfedge.
    3. Repeat step 2 until you hit a dead end or get back to the start (in that case you found a closed polyline)
    4. If the polyline is not closed: Find the opposite halfedge of the original one found in step 1.
    5. Repeat step 2 until you hit another dead end.

Much more detailed information on half-edge structures and their theory can easily be found in the web. Two scientific papers dealing with the subject are:

*S. Campagna, L. Kobbelt, H.-P. Seidel, Directed Edges - A Scalable Representation For Triangle Meshes , ACM Journal of Graphics Tools 3 (4), 1998.*

*Lutz Kettner, Using Generic Programming for Designing a Data Structure for Polyhedral Surfaces, in Proc. 14th Annual ACM Symp. on Computational Geometry, 1998.*

## Syntax 101

Every function provided by the EvoluteTools Scripting Interface evidently has it's own syntax, but the following general rules on how to use mesh elements like vertices and faces as parameters apply to a large majority of them, so they are summarized here. Mesh elements are encoded as so called EvoluteTools Handles, which are strings comprised of a letter designating the handle type and a number for its index. To identify a mesh element within a Rhino document, however, we do also need to identify the mesh object itself. For enabling efficient scripting, the EvoluteTools Scripting Interface provides two general ways of handing the combination of EvoluteTools Handles and mesh UUIDs to its functions:

1. **Array of *unique mesh element handles*:**
   This is the standard input format, and also the only way functions will return mesh element data. What needs to be provided is an array of arbitrary length, each element consisting of a string array of length two. Within these string arrays, the first element is the UUID of a valid mesh object, the second an EvoluteTools Handle of a valid vertex, edge, halfedge or face within that mesh. Such an array of strings is called a ***Unique EvoluteTools Handle*** in this documentation ( if the type is specified, this is changed to *Unique Vertex Handle* etc. ). If the parameter is only a single handle, a single Unique EvoluteTools Handle can be used (instead of an array with one unique handle as its only element).

   ***Example Code:***

```
Option Explicit
Call Main()
Sub Main()
  Dim et, mesh, input
  Set et = Rhino.GetPlugInObject( "EvoluteTools for Rhino" )
  mesh = Rhino.GetObject( "Select Mesh", 32 )
  If ( et.EmNumberOfVertices( mesh ) > 3 ) Then
    input = Array(Array( mesh, "v0"), Array( mesh, "v1"), Array( mesh, "v2" ) )
    et.EmDeleteVertex( input )
  End If
End Sub
```

2. **Mesh id and array of handles:**
   In situations where operations with a large number of handles within the same mesh are processed, it might be more efficient not to hand over the same mesh id for every single handle. Alternatively it is possible to provide a single string holding the mesh UUID and, as a second parameter, an array of strings holding the handles. Again, in the case of a single handle the second parameter can be a string instead of an array with one element.

   ***Example Code:***

```
Option Explicit
Call Main()
Sub Main()
  Dim et, mesh, input
  Set et = Rhino.GetPlugInObject( "EvoluteTools for Rhino" )
  mesh = Rhino.GetObject( "Select Mesh", 32 )
  If ( et.EmNumberOfVertices( mesh ) > 3 ) Then
    input = Array( "v0", "v1", "v2" )
    et.EmDeleteVertex mesh, input
  End If
End Sub
```

Several commands need either a minimum or a specific number of input handles to work. These will generally ignore any redundant handles. Furthermore, many commands have additional input parameters of different types which are described in their individual documentation entries.

Commands like EmNormal accept input handles of several different types, in this case \Vertex and Face Handles. For others, like Em\VertexValence the type of necessary input handle is unambiguous. Most of the commands of the second type will also accept integers instead of proper Handles for ease of use. Please see the individual documentation entries for details.

# EvoluteTools Handles

Vertices, edges, half-edges and faces of a mesh are each numbered and indentified by their indices starting at 0. The number of, e.g. vertices of a mesh can easily be found out using the command EmNumberOfVertices. To avoid ambiguous function calls and confusion of mesh elements of different type, the standard format for mesh elements when working with the EvoluteTools Scripting Interface is that of a string consisting of a lower case letter designating the type of the element followed by the respective index.

The prefix letters are

| Element | Prefix | Example |
|---|---|---|
| vertex | v | v143 |
| halfedge | h | h0 |
| edge | e | e9 |
| face | f | f9342 |

In the documentation, these strings will be referred to as ***Vertex Handles***, ***Face Handles*** etc.

## Unique Handles

These handles can not be properly used without identifying the Rhino mesh object which the described element is a part of. An array of strings consisting of one unique mesh id and one EvoluteTools Handle is called ***Unique EvoluteTools Handle*** in this documentation. If the type needs to be specified, the terms ***Unique Vertex Handle*** etc. are used, respectively. Such an array needs to have exactly two elements.

# Deleting Mesh Components

## Garbage Collection

The sets of vertices, edges and faces of a mesh are each stored as arrays, providing fast and efficient access to any single mesh component via its type and index - this is done using EvoluteTools Handles. In general, the size of the array is identical to the number of valid elements, meaning that by knowing the number of, e.g. vertices in a mesh ( see EmNumberOfVertices ) one does also knows if an EvoluteTools Handle is valid for a mesh or not. However, this is no longer true as soon as an element is deleted from the mesh as after such an operation there the former index of this element becomes invalid. There are two ways of resolving this situation:

- Accept the existence of such invalid handles and take precautions by either remembering all invalid indices or catching possible errors by always testing each handles' validity before using it. While this method is possible in the EvoluteTools Scripting Engine, it is strongly recommended to only use it in a very limited set of circumstances ( see examples below ), as it potentially complicates scripts and increases the probability of running into errors.

- Recreate the array of mesh elements, potentially changing the indices of all vertices, edges and faces in the mesh. This process will keep the mesh definition "clean" of leftover invalid Handles and is aptly named *Garbage Collection*. However, any set of Handles saved before throwing out the garbage is then useless, as these might now point to different or non-existent vertices, edges or faces.

While every user is obviously free to handle this issue as he or she prefers, it is again strongly recommended to clean up "dirty" meshes at the earliest possible opportunity. The following paragraphs will explain how to use EvoluteTools Scripting functions to handle the issue of *Garbage Collection* and give some typical examples of proper use.

### Use of EvoluteTools Scripting Functions

First of all it is important to note that all functions provided by EvoluteTools for Rhino Scripting Interface are designed to catch invalid handles. Generally, they will simply return Null if they are handed one. All functions which potentially delete mesh elements have an optional parameter called "ByRef *blnGarbageCollection* As Bool" which is set to "*true*" by default, which means that all such functions will perform a *Garbage Collection* if they are not specifically prevented from doing so by setting blnGarbageCollection to *false*. In addition, there is the EmDirty command used to test if a mesh has invalid handles and the EmGarbageCollection command to manually perform a cleanup. The validity of a specific handle can be tested using EmHandleIsValid. In case you want to call a standard Rhino command provided by EvoluteTools from your script, be aware that all of them will always clean up after themselves - and there is no way of preventing it. Calling external or native Rhino commands which alter the connectivity of a mesh will (with very few exceptions) automatically invalidate all handles as well, because it forces EvoluteTools to recreate the meshes' handle index vectors.

### Examples

1. **Deleting Vertices one by one:**
   In general, the most efficient way of deleting a set of vertices is to use EmDeleteVertices. However, there may be situations where you want to do this vertex by vertex, e.g. if you want to check something after each one and abort if some condition is met. Here, it would be inefficient to have each call to EmDeleteVertex perform a *GarbageCollection* - apart from the computation time used for repeating this operation for each vertex, it would also invalidate the rest of your handles. In the following example script, we delete three vertices and check after each one if a certain face has been deleted with it.

   ***Example Code:***

```
Option Explicit
Call Main()
Sub Main()

Dim et, vertices, face, i, meshID
Set et = Rhino.GetPlugInObject( "EvoluteTools for Rhino" )

vertices = et.EmPickVertex( 3, 3, "Pick vertices you want to delete" )
If IsNull( vertices ) = True Then
   Exit Sub
End If

meshID = vertices(0)(0)
If meshID <> vertices(1)(0) Or meshID <> vertices(2)(0) Then
   Rhino.Print "Vertices must be on same mesh. Aborting."
   Exit Sub
End If

face = et.EmPickFace( 1, 1, "Pick a face to watch" )

For i = 0 To 2
   'Here we call EmDeleteVertex with deactivated Garbage Collection
   If IsNull( et.EmDeleteVertex( vertices(i), False ) ) = True Then
      et.EmGarbageCollection( meshID )
      Exit Sub
   End If
   If et.EmHandleIsValid( face ) = False Then
      Rhino.Print "Watched face has been deleted. Aborting."
      Exit Sub
   End If
Next
et.EmGarbageCollection meshID

Rhino.Print "All vertices were deleted without touching the watched face."
End Sub
```

2. **Using higher-level deleting functions:**
   Several functions in the EvoluteTools Scripting Engine can potentially delete a whole number of mesh elements. Preventing these commands from immediately performing a *Garbage Collection* is a good way to check what exactly has been deleted. The following script deletes a polyline from a mesh and inserts red line objects in place of former edges.

   ***Example Code:***

```
Option Explicit
Call Main()
Sub Main()

  Dim et, edge, i, meshID
  Set et = Rhino.GetPlugInObject( "EvoluteTools for Rhino" )

  edge = et.EmPickEdge( 1, 1, "Pick an edge to delete its polyline" )
  If IsNull( edge ) Then
    Exit Sub
  End If

  meshID = edge(0)
  'Copy object to find deleted edges later
  Dim meshCopy
  meshCopy = Rhino.CopyObject( meshID )

  'Here we call EmDeletePolyline with deactivated Garbage Collection
  If IsNull( et.EmDeletePolyline( edge, False ) ) Then
    et.EmGarbageCollection( meshID )
    Exit Sub
  End If

  'Here we count from 0 to the number of vertices, including deleted ones
  For i = 0 To et.EmNumberOfEdges( meshID, True ) - 1
    If ( et.EmHandleIsValid( meshID, et.EmEdge( i ) ) ) = False Then
      Dim vertices, startPoint, endPoint, line
      vertices = et.EmGetEdgeVertices( meshCopy, et.EmEdge( i ) )
      startPoint = et.EmPoint( vertices(0) )
      endPoint = et.EmPoint( vertices(1) )
      line = Rhino.AddLine( startPoint, endPoint )
      Rhino.ObjectColor line, VBRed
    End If
  Next

  et.EmGarbageCollection meshID
  Rhino.DeleteObject meshCopy
End Sub
```

*Note:* The EmDeletePolyline command does not only delete those edges belonging to the actual polyline, but also merges edges of polylines crossing it. For each pair of merged edges, one of them is deleted, too - this leads to additional red lines created by the previous script. This can easily be prevented by designing the script differently.

# EvoluteTools Scripting Examples

The subfolder *ScriptingExamples* of the plugin package contains a few simple examples demonstrating the capabilities of the EvoluteTools Scripting Interface.

| | |
|---|---|
| Incircles.rvb | Extracts incircles of the triangles in a mesh. |
| JoinStripFaces.rvb | Lets the user pick an edge, traces a strip of quads from that edge, and removes all the edges until it hits the boundary. |
| LoftQuadStrip.rvb | Lets the user pick an edge, trace a strip of quads from that edge, and creates a ruled strip surface by lofting all the edges it meets. Useful for generating ruled or developable strips from a quad mesh. |
| SubStructure.rvb | Creates beams, nodes and panels for a mesh for rendering purposes. |
| JoinStripFaces.rvb | Joins the faces of a quad strip to a single face. |
| KdtreeTest.rvb | Demonstrates the usage of the k-d tree functionality in the EvoluteTools RhinoScript interface. |
| CreateRuledStripsFromQuadDominantMesh.rvb | Creates ruled strips from a quad dominant mesh, and optimizes them. |

# Integration with Monkey RhinoScript Editor

EvoluteTools Pro supports code autocompletion in Rhino's script editor (aka Monkey editor). If you are using Rhino 4.0, please copy the file *EvoluteToolsForRhino.syntaxml* to the subfolder *Plug-ins\Monkey\Resources* of your Rhino installation to activate this feature, or run the following script. If you are using Rhino 5.0, please run "InstallRhinoScriptSyntaxDescription.bat" and restart Rhino. Please watch out for errors, you might need to run this script using administrator privileges.

## EmAddFace

*Adds a new face to a mesh.*

This function takes an array of Unique Vertex Handles and adds a new face to the mesh, if possible. If necessary, edges will also be added. If successful, the function returns the Unique Face Handle of the newly added face.

**Syntax:**

EmAddFace( *arrHandleData* )

EmAddFace( *strMesh*, *arrVertexHandles* )

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Array of Unique Vertex Handles. |
| *strMesh* | Required. String. The mesh. |
| *arrVertexHandles* | Required. Array. Array of Vertex Handles. |

**Return value:**

| | |
|---|---|
| Array | Unique Face Handle. |
| Null | On error. |

## EmAddPolyMesh

*Adds a polymesh object to the document.*

**Syntax:**

EmAddPolyMesh( *arrVertices*, *arrFaceVertices* [, *arrVertexNormals*] )

**Parameters:**

| | |
|---|---|
| *arrVertices* | Required. Array. An array of 3-D points defining the vertices of the mesh. |
| *arrFaceVertices* | Required. Array. An array containing arrays of integers (arbitrary length) that define the vertex indices for each face of the mesh. |
| *arrVertexNormals* | Optional. Array. An array of 3-D vectors defining the vertex normals of the mesh. If not provided, the normals will be computed automatically. |

**Return value:**

| | |
|---|---|
| String | The identifier of the new object if successful. |
| Null | On error. |

## EmAddVertex

*Adds vertices with given coordinates to a mesh.*

**Syntax:**

EmAddVertex( *strMesh*, *arrArrPoints* )

**Parameters:**

| | |
|---|---|
| *strMesh* | Required. String. A string identifying the mesh object. |
| *arrArrPoints* | Required. Array. An array of 3-D points at whose coordinates new vertices should be added. |

**Return value:**

| | |
|---|---|
| Array | Array of Unique Vertex Handles of the newly added vertices. |
| Null | On error. |

## EmCCWRotatedHalfedge

*Returns the counter-clockwise rotated halfedge for a given halfedge. EmCCWRotatedHalfedge( x ) is analog to EmOppositeHalfedgeHandle( EmPreviousHalfedgeHandle( x ) )*

**Syntax:**

EmCCWRotatedHalfedge( *arrHandle* )

EmCCWRotatedHalfedge( *strMesh*, *strHandle* )

EmCCWRotatedHalfedge( *strMesh*, *intIndex* )

**Parameters:**

| | |
|---|---|
| *arrHandle* | Required. Array. Unique Halfedge Handle. |
| *strMesh* | Required. String. A string identifying the mesh object. |
| *strHandle* | Required. String. Halfedge Handle |
| *intIndex* | Required. Integer. The index of the input Halfedge Handle. |

**Return value:**

| | |
|---|---|
| Array | Unique Halfedge Handle |
| Null | On error. |

## EmCWRotatedHalfedge

*Returns the clockwise rotated halfedge for a given halfedge. EmCWRotatedHalfedge( x ) is analog to EmNextHalfedgeHandle( EmOppositeHalfedgeHandle( x ) )*

**Syntax:**

EmCWRotatedHalfedge( *arrHandle* )

EmCWRotatedHalfedge( *strMesh*, *strHandle* )

EmCWRotatedHalfedge( *strMesh*, *intIndex* )

**Parameters:**

| | |
|---|---|
| *arrHandle* | Required. Array. Unique Halfedge Handle. |
| *strMesh* | Required. String. A string identifying the mesh object. |
| *strHandle* | Required. String. Halfedge Handle |
| *intIndex* | Required. Integer. The index of the input Halfedge Handle. |

**Return value:**

| | |
|---|---|
| Array | Unique Halfedge Handle |
| Null | On error. |

## EmDeleteEdge

*Deletes edges from a mesh. For interior edges, the adjacent faces are joined. For boundary edges, the adjacent face is deleted.*

**Syntax:**

EmDeleteEdge( *arrHandleData* [, *blnDeleteIsolatedVertices* [, *blnGarbageCollection*]] )

EmDeleteEdge( *strMesh*, *arrEdgeHandles* [, *blnDeleteIsolatedVertices* [, *blnGarbageCollection*]] )

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Array of Unique Edge Handles. |
| *strMesh* | Required. String. The mesh. |
| *arrEdgeHandles* | Required. Array (or String). Either a single proper Edge Handle or an array of such Handles. |
| *blnDeleteIsolatedVertices* | Optional. Boolean. If true (default) isolated vertices remaining in the mesh after the deletion process is finished are deleted, too. |
| *blnGarbageCollection* | Optional. Boolean. If true (default) the mesh element indices are reset after this command, making previously saved handles invalid. See a general explanation of the subject and the documentation to the EmGarbageCollection command for more information. |

**Return value:**

| | |
|---|---|
| Integer | Number of deleted vertices. |
| Null | On error. |

## EmDeleteFace

*Deletes faces from a mesh.*

**Syntax:**

EmDeleteFace( *arrHandleData* [, *blnDeleteIsolatedVertices* [, *blnGarbageCollection*]] )

EmDeleteFace( *strMesh*, *arrFaceHandles* [, *blnDeleteIsolatedVertices* [, *blnGarbageCollection* ] ])

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Array of Unique Face Handles. |
| *strMesh* | Required. String. The mesh. |
| *arrFaceHandles* | Required. Array (or String). Either a single proper Face Handle or an array of such Handles. |
| *blnDeleteIsolatedVertices* | Optional. Boolean. If true (default) isolated vertices remaining in the mesh after the deletion process is finished are deleted, too. |
| *blnGarbageCollection* | Optional. Boolean. If true (default) the mesh element indices are reset after this command, making previously saved handles invalid. See a general explanation of the subject and the documentation to the EmGarbageCollection command for more information. |

**Return value:**

| | |
|---|---|
| Integer | Number of deleted faces. |
| Null | On error. |

## EmDeleteIsolatedVertices

*Deletes all isolated vertices from a mesh.*

**Syntax:**

EmDeleteIsolatedVertices( *arrObjects* [, *blnGarbageCollection*] )

**Parameters:**

| | |
|---|---|
| *arrObjects* | Required. Array. An array of strings identifying the meshes to delete isolated vertices from. |
| *blnGarbageCollection* | Optional. Boolean. If true (default) the mesh element indices are reset after this command, making previously saved handles invalid. See a general explanation of the subject and the documentation to the EmGarbageCollection command for more information. |

**Return value:**

| | |
|---|---|
| Integer | Number of deleted vertices. |
| Null | On error. |

## EmDeleteVertex

*Deletes vertices from a mesh. All connected faces and edges are deleted as well.*

**Syntax:**

EmDeleteVertex( *arrHandleData* [, *blnDeleteIsolatedVertices* [, *blnGarbageCollection*]] )

EmDeleteVertex( *strMesh*, *arrVertexHandles* [, *blnDeleteIsolatedVertices* [, *blnGarbageCollection*]] )

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Array of Unique Vertex Handles. |
| *strMesh* | Required. String. The mesh. |
| *arrVertexHandles* | Required. Array (or String). Either a single proper Vertex Handle or an array of such Handles. |
| *blnDeleteIsolatedVertices* | Optional. Boolean. If true (default) isolated vertices remaining in the mesh after the deletion process is finished are deleted, too. |
| *blnGarbageCollection* | Optional. Boolean. If true (default) the mesh element indices are reset after this command, making previously saved handles invalid. See a general explanation of the subject and the documentation to the EmGarbageCollection command for more information. |

**Return value:**

| | |
|---|---|
| Integer | Number of deleted vertices. |
| Null | On error. |

## EmDihedralAngleAtEdge

*Returns the dihedral angle between the planes defined by the face normals of a given edges' two adjacent faces.*

**Syntax:**

EmDihedralAngleAtEdge( *arrHandleData* )

EmDihedralAngleAtEdge( *strMesh*, *strEdgeHandle* )

EmDihedralAngleAtEdge( *strMesh*, *intIndex* )

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Array of Unique Edge Handle. |
| *strMesh* | Required. String. The mesh. |
| *strEdgeHandle* | Required. String. Edge Handle |
| *intIndex* | Required. Integer. The index of the input Edge Handle. |

**Return value:**

| | |
|---|---|
| Double | Dihedral angle at input edge. |
| Null | On error. |

## EmDisplaceVertex

*A given vertex is displaced along a given vector.*

**Syntax:**

EmDisplaceVertex( *arrHandleData*, *arrVector* )

EmDisplaceVertex( *strMesh*, *strVertexHandle*, *arrVector* )

EmDisplaceVertex( *strMesh*, *intIndex*, *arrVector* )

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Unique Vertex Handle. |
| *strMesh* | Required. String. The mesh. |
| *strVertexHandle* | Required. String. Vertex Handle |
| *arrVector* | Required. Array. The 3-D vector along which the vertex should be displaced. |
| *intIndex* | Required. Integer. The index of the input Vertex Handle. |

**Return value:**

| | |
|---|---|
| Bool | If true, the vertex has been properly moved. |
| Null | On error. |

## EmEdge

*Creates an Edge Handle from an integer, returns the corresponding Unique Edge Handle for a given Unique Halfedge Handle.*

**Syntax:**

EmEdge( *intIndex* ) (returns Edge Handle)

EmEdge( *strMesh*, *intIndex* ) (returns Unique Edge Handle)

EmEdge( *arrHandle* ) (returns Unique Edge Handle)

EmEdge( *strMesh*, *strHandle* ) (returns Unique Edge Handle)

This function has two separate uses. Its first version takes an integer and simply returns an Edge Handle with the input as its index. So an input of "17" will create a the string "e17". This is analog to the functionality of the EmVertex function.
The second version takes a Unique Halfedge Handle and returns the Unique Edge Handle of the edge to which it belongs.

**Parameters:**

| | |
|---|---|
| *intIndex* | Required. Integer. The index of the Edge Handle to be created. |
| *arrHandle* | Required. Array. Unique Halfedge Handle. |
| *strMesh* | Required. String. A string identifying the mesh object. |
| *strHandle* | Required. String. Halfedge Handle |

**Return value:**

| | |
|---|---|
| String or Array | Edge Handle or Unique Edge Handle |
| Null | On error. |

## EmEndVertex

*Returns the end vertex of a given halfedge.*

**Syntax:**

EmEndVertex( *arrHandle* )

EmEndVertex( *strMesh*, *strHandle* )

EmEndVertex( *strMesh*, *intIndex* )

**Parameters:**

| | |
|---|---|
| *arrHandle* | Required. Array. Unique Halfedge Handle. |
| *strMesh* | Required. String. A string identifying the mesh object. |
| *strHandle* | Required. String. Halfedge Handle |
| *intIndex* | Required. Integer. The index of the input Halfedge Handle. |

**Return value:**

| | |
|---|---|
| Array | Unique Vertex Handle |
| Null | On error. |

## EmFace

*Creates a Face Handle from an integer, returns an adjacent Unique Face Handle for a given Unique Halfedge Handle.*

**Syntax:**

EmFace( *intIndex* ) (returns Face Handle)

EmFace( *strMesh*, *intIndex* ) (returns Unique Face Handle)

EmFace( *arrHandle* ) (returns Unique Face Handle)

EmFace( *strMesh*, *strHandle* ) (returns Unique Face Handle)

This function has two separate uses. Its first version takes an integer and simply returns a Face Handle with the input as its index. So an input of "17" will create a the string "f17". This is analog to the functionality of the EmVertex function.
The second version takes a Unique Halfedge Handle and returns the Unique Face Handle of the adjacent face. If the input halfedge is a boundary halfedge, this command returns Null.

**Parameters:**

| | |
|---|---|
| *intIndex* | Required. Integer. The index of the Face Handle to be created. |
| *arrHandle* | Required. Array. Unique Halfedge Handle. |
| *strMesh* | Required. String. A string identifying the mesh object. |
| *strHandle* | Required. String. Halfedge Handle |

**Return value:**

| | |
|---|---|
| String or Array | Face Handle or Unique Face Handle |
| Null | On error. |

## EmFaceValence

*Returns the valence of a mesh face.*

**Syntax:**

EmFaceValence( *arrHandleData* )

EmFaceValence( *strMesh*, *strFaceHandle* )

EmFaceValence( *strMesh*, *intIndex* )

This function returns the number of vertices adjacent to a mesh face.

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Unique Face Handle. |
| *strMesh* | Required. String. The mesh. |
| *strFaceHandle* | Required. String. Face Handle |
| *intIndex* | Required. Integer. The index of the input Face Handle. |

**Return value:**

| | |
|---|---|
| Integer | Number of vertices adjacent to the input face. |
| Null | On error. |

## EmGarbageCollection

*Removes mesh elements marked as deleted and recreates the arrays of mesh vertices, edges, halfedges and faces. All existing EvoluteTools Handles to this mesh are invalidated.*

**Syntax:**

EmGarbageCollection( *strMesh* )

This function is used to clean up a mesh from elements which have been marked as deleted. The arrays defining the indices of EvoluteTools Handles are recreated, invalidating all existing Handles as they might point to a different or nonexistent element. See the help entry on deleting mesh elements for more information.

**Parameters:**

| | |
|---|---|
| *strMesh* | Required. String. A string identifying the mesh object. |

**Return value:**

| | |
|---|---|
| Boolean | True if the garbage collection was successful. |
| Null | On error. |

## EmGetEdgeFaces

*Returns the two adjacent faces for a given edge.*

**Syntax:**

EmGetEdgeFaces( *arrHandleData* )

EmGetEdgeFaces( *strMesh*, *strEdgeHandle* )

EmGetEdgeFaces( *strMesh*, *intIndex* )

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Unique Edge Handle. |
| *strMesh* | Required. String. The mesh. |
| *strEdgeHandle* | Required. String. Edge Handle |
| *intIndex* | Required. Integer. The index of the input Edge Handle. |

**Return value:**

| | |
|---|---|
| Array | Array of Unique Face Handles |
| Null | On error. |

## EmGetEdgePoints

*Returns the 3-D points belonging to the two adjacent vertices of a given edge.*

**Syntax:**

EmGetEdgePoints( *arrHandleData* )

EmGetEdgePoints( *strMesh*, *strEdgeHandle* )

EmGetEdgePoints( *strMesh*, *intIndex* )

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Unique Edge Handle. |
| *strMesh* | Required. String. The mesh. |
| *strEdgeHandle* | Required. String. Edge Handle |
| *intIndex* | Required. Integer. The index of the input Edge Handle. |

**Return value:**

| | |
|---|---|
| Array | Array of 3-D Points. |
| Null | On error. |

## EmGetEdgeVertices

*Returns the two adjacent vertices for a given edge.*

**Syntax:**

EmGetEdgeVertices( *arrHandleData* )

EmGetEdgeVertices( *strMesh*, *strEdgeHandle* )

EmGetEdgeVertices( *strMesh*, *intIndex* )

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Unique Edge Handle. |
| *strMesh* | Required. String. The mesh. |
| *strEdgeHandle* | Required. String. Edge Handle |
| *intIndex* | Required. Integer. The index of the input Edge Handle. |

**Return value:**

| | |
|---|---|
| Array | Array of Unique Vertex Handles |
| Null | On error. |

## EmGetFaceEdges

*Returns all adjacent edges for a given face.*

**Syntax:**

EmGetFaceEdges( *arrHandleData* )

EmGetFaceEdges( *strMesh*, *strFaceHandle* )

EmGetFaceEdges( *strMesh*, *intIndex* )

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Unique Face Handle. |
| *strMesh* | Required. String. The mesh. |
| *strFaceHandle* | Required. String. Face Handle |
| *intIndex* | Required. Integer. The index of the input Face Handle. |

**Return value:**

| | |
|---|---|
| Array | Array of Unique Edge Handles |
| Null | On error. |

## EmGetFaceHalfedges

*Returns all adjacent halfedges for a given face.*

**Syntax:**

EmGetFaceHalfedges( *arrHandleData* )

EmGetFaceHalfedges( *strMesh*, *strFaceHandle* )

EmGetFaceHalfedges( *strMesh*, *intIndex* )

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Unique Face Handle. |
| *strMesh* | Required. String. The mesh. |
| *strFaceHandle* | Required. String. Face Handle |
| *intIndex* | Required. Integer. The index of the input Face Handle. |

**Return value:**

| | |
|---|---|
| Array | Array of Unique Halfedge Handles |
| Null | On error. |

## EmGetFacePoints

*Returns the 3-D points belonging to all adjacent vertices of a given face.*

**Syntax:**

EmGetFacePoints( *arrHandleData* )

EmGetFacePoints( *strMesh*, *strFaceHandle* )

EmGetFacePoints( *strMesh*, *intIndex* )

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Unique Face Handle. |
| *strMesh* | Required. String. The mesh. |
| *strFaceHandle* | Required. String. Face Handle |
| *intIndex* | Required. Integer. The index of the input Face Handle. |

**Return value:**

| | |
|---|---|
| Array | Array of 3-D points. |
| Null | On error. |

## EmGetFaceVertices

*Returns all adjacent vertices for a given face.*

**Syntax:**

EmGetFaceVertices( *arrHandleData* )

EmGetFaceVertices( *strMesh*, *strFaceHandle* )

EmGetFaceVertices( *strMesh*, *intIndex* )

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Unique Face Handle. |
| *strMesh* | Required. String. The mesh. |
| *strFaceHandle* | Required. String. Face Handle |
| *intIndex* | Required. Integer. The index of the input Face Handle. |

**Return value:**

| | |
|---|---|
| Array | Array of Unique Vertex Handles |
| Null | On error. |

## EmGetPolylinePoints

*Returns the 3-D points belonging to the vertices of the polyline indicated by the given edge.*

**Syntax:**

EmGetPolylinePoints( *arrHandleData* )

EmGetPolylinePoints( *strMesh*, *strEdgeHandle* )

EmGetPolylinePoints( *strMesh*, *intEdgeIndex* )

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Unique Edge Handle. |
| *strMesh* | Required. String. The mesh. |
| *strEdgeHandle* | Required. String. Edge Handle |
| *intEdgeIndex* | Required. Integer. The index of the input Edge Handle. |

**Return value:**

| | |
|---|---|
| Array | Array of 3-D Points. |
| Null | On error. |

## EmGetPolylineVertices

*Returns the vertices of the mesh polyline indicated by the given edge.*

**Syntax:**

EmGetPolylineVertices( *arrHandleData* )

EmGetPolylineVertices( *strMesh*, *strEdgeHandle* )

EmGetPolylineVertices( *strMesh*, *intEdgeIndex* )

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Unique Edge Handle. |
| *strMesh* | Required. String. The mesh. |
| *strEdgeHandle* | Required. String. Edge Handle |
| *intEdgeIndex* | Required. Integer. The index of the input Edge Handle. |

**Return value:**

| | |
|---|---|
| Array | Array of Unique Vertex Handles |
| Null | On error. |

## EmHalfedge

*Creates a Halfedge Handle from an integer, or returns an adjacent Unique Halfedge Handle*

- *for a given Unique Vertex Handle, or*
- *for a Unique Face Handle, or*
- *one of the corresponding Unique Halfedge Handles for a given Unique Edge Handle.*

**Syntax:**

EmHalfedge( *intIndex* ) (returns Halfedge Handle)

EmHalfEdge( *strMesh*, *intIndex* ) (returns Unique Halfedge Handle)

EmHalfedge( *arrHandle* [, *intWhich*]) (returns Unique Halfedge Handle)

EmHalfedge( *strMesh*, *strHandle* [, *intWhich*] ) (returns Unique Halfedge Handle)

This function has two separate uses. Its first version takes an integer and simply returns a Halfedge Handle with the input as its index. So an input of "17" will create a the string "h17". This is analog to the functionality of the EmVertex function. Optionally, a string designating a mesh object can be specified together with the integer to produce a Unique Edge Handle.
The second version takes a Unique EvoluteTools Handle of type vertex, face, or edge. If a valid Unique Vertex Handle is provided, the return will be the Unique EvoluteTools Handle of an adjacent halfedge. If a Unique Face handle is provided, this function will return the associated Unique Halfedge Handle. If a Unique Edge Handle is provided, this function will return one of the two Unique Halfedge Handles belonging to the input handle. In this case the parameter *intWhich* can be used to distinguish between the two, accepts 0 or 1 as valid input and defaults to 0.

***Short code example motivating the** intWhich **parameter:***

```
Dim edge, et
Set et = Rhino.GetPlugInObject( "EvoluteTools for Rhino" )
edge = et.EmPickEdge( 1,1,"Pick some edge" )

Dim halfedge, distanceZ, startPoint, endPoint
halfedge = et.EmHalfedge( edge, 0 )
startPoint = et.EmPoint( et.EmStartVertex( halfedge ) )
endPoint = et.EmPoint( et.EmEndVertex( halfedge ) )
distanceZ = endPoint(2) - startPoint(2)
If ( distanceZ <= 0.0 ) Then
   halfedge = et.EmHalfedge( edge, 1 )
End If
End Sub
```

**Parameters:**

| | |
|---|---|
| *intIndex* | Required. Integer. The index of the Halfedge Handle to be created. |
| *arrHandle* | Required. Array. Unique Vertex Handle or Unique Face Handle or Unique Edge Handle. |
| *strMesh* | Required. String. A string identifying the mesh object. |
| *strHandle* | Required. String. Vertex Handle or Edge Handle |
| *intWhich* | Optional. Integer. If the input handle is an Edge Handle, this parameter can be used to distinguish between the two possible return halfedges by specifying 0 or 1. Default is 0. |

**Return value:**

| | |
|---|---|
| String or Array | Halfedge Handle or Unique Halfedge Handle |
| Null | On error. |

## EmHalfedgeVector

*Returns the 3-D vector defined by a given mesh halfedge.*

**Syntax:**

EmHalfedgeVector( *arrHandleData* )

EmHalfedgeVector( *strMesh*, *strHalfedgeHandle* )

EmHalfedgeVector( *strMesh*, *intIndex* )

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Unique Halfedge Handle. |
| *strMesh* | Required. String. The mesh. |
| *strHalfedgeHandle* | Required. String. Halfedge Handle |
| *intIndex* | Required. Integer. The index of the input Halfedge Handle. |

**Return value:**

| | |
|---|---|
| Array | The 3-D vector defined by the input halfedge. |
| Null | On error. |

## EmHandleIsValid

*Checks if a provided handle point to an existing valid mesh element.*

**Syntax:**

EmHandleIsValid( *arrUniqueHandle* )

EmHandleIsValid( *strMesh*, *strHandle* )

**Parameters:**

| | |
|---|---|
| *arrUniqueHandle* | Required. Array. Unique EvoluteTools Handle. |
| *strMesh* | Required. String. The mesh. |
| *strHandle* | Required. String. EvoluteTools Handle |

**Return value:**

| | |
|---|---|
| Boolean | True if mesh element identified by the input handle exists and is not marked as deleted. |
| Null | On error, here usually in case the input is not a properly formatted EvoluteTools Handle. |

## EmIndex

*Returns the index of an EvoluteTools Handle or Unique EvoluteTools Handle.*

**Syntax:**

EmIndex( *arrUniqueHandle* )

EmIndex( *strHandle* )

**Parameters:**

| | |
|---|---|
| *arrUniqueHandle* | Required. Array. Unique EvoluteTools Handle. |
| *strHandle* | Required. String. EvoluteTools Handle. |

**Return value:**

| | |
|---|---|
| Integer | Index of input handle. |
| Null | On error. |

## EmlsBoundary

*Checks if a given vertex, edge or halfedge lies on a boundary.*

**Syntax:**

EmlsBoundary( *arrHandleData* )

EmlsBoundary( *strMesh*, *strHandle* )

Note that only one of the two halfedges belonging to a boundary edge is also a boundary halfedge. One of the halfedges corresponds to an adjacent face, the other is a boundary halfedge.

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Unique Vertex Handle, Unique Edge Handle or Unique Halfedge Handle. |
| *strMesh* | Required. String. The mesh. |
| *arrVertexHandles* | Required. String. Vertex Handle, Edge Handle or Halfedge Handle. |

**Return value:**

| | |
|---|---|
| Boolean | True if the vertex, edge or halfedge lies on a boundary. |
| Null | On error. |

## EmIsIsolated

*Checks if a given vertex has any adjacent edges and faces.*

**Syntax:**

EmIsIsolated( *arrHandleData* )

EmIsIsolated( *strMesh*, *strHandle* )

EmIsIsolated( *strMesh*, *intIndex* )

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Unique Vertex Handle. |
| *strMesh* | Required. String. The mesh. |
| *arrVertexHandles* | Required. String. Vertex Handle. |
| *intIndex* | Required. Integer. Index of the input Vertex Handle. |

**Return value:**

| | |
|---|---|
| Boolean | True if the vertex is isolated. |
| Null | On error. |

## EmNextHalfedge

*Returns the next halfedge for a given halfedge.*

**Syntax:**

EmNextHalfedge( *arrHandle* )

EmNextHalfedge( *strMesh*, *strHandle* )

EmNextHalfedge( *strMesh*, *intIndex* )

**Parameters:**

| | |
|---|---|
| *arrHandle* | Required. Array. Unique Halfedge Handle. |
| *strMesh* | Required. String. A string identifying the mesh object. |
| *strHandle* | Required. String. Halfedge Handle |
| *intIndex* | Required. Integer. The index of the input Halfedge Handle. |

**Return value:**

| | |
|---|---|
| Array | Unique Halfedge Handle |
| Null | On error. |

## EmNormal

*Returns the normal vector of a given vertex or face.*

**Syntax:**

EmNormal( *arrHandle* )

EmNormal( *strMesh*, *strHandle* )

**Parameters:**

| | |
|---|---|
| *arrHandle* | Required. Array. Unique Vertex Handle or Unique Face Handle. |
| *strMesh* | Required. String. A string identifying the mesh object. |
| *strHandle* | Required. String. Vertex Handle or Face Handle. |

**Return value:**

| | |
|---|---|
| Array | The 3D normal vector. |
| Null | On error. |

## EmNumberOfEdges

*Returns number of edges of a mesh object.*

**Syntax:**

EmNumberOfEdges( *strMesh* [, *blnCountDeleted*] )

By default, this function counts the valid edges of a mesh object, meaning that if edges have been deleted by functions of the Scripting Interface without performing a Garbage Collection afterwards, these edges are omitted. If the deleted edges should be counted as well, the optional boolean parameter *blnCountDeleted* must be set to True.

**Parameters:**

| | |
|---|---|
| *strMesh* | Required. String. A string identifying the mesh object. |
| *blnCountDeleted* | Optional. Boolean. If true, edges marked as deleted will be counted. See the documentation on Garbage Collection for more information. If omitted, defaults to False. |

**Return value:**

| | |
|---|---|
| Integer | Number of valid edges or total edges on the mesh. |
| Null | On error. |

## EmNumberOfFaces

*Returns number of faces of a mesh object.*

**Syntax:**

EmNumberOfFaces( *strMesh* [, *blnCountDeleted*] )

By default, this function counts the valid faces of a mesh object, meaning that if faces have been deleted by functions of the Scripting Interface without performing a Garbage Collection afterwards, these faces are omitted. If the deleted faces should be counted as well, the optional boolean parameter *blnCountDeleted* must be set to True.

**Parameters:**

| | |
|---|---|
| *strMesh* | Required. String. A string identifying the mesh object. |
| *blnCountDeleted* | Optional. Boolean. If true, faces marked as deleted will be counted. See the documentation on Garbage Collection for more information. If omitted, defaults to False. |

**Return value:**

| | |
|---|---|
| Integer | Number of valid faces or total faces on the mesh. |
| Null | On error. |

## EmNumberOfVertices

*Returns number of vertices of a mesh object.*

**Syntax:**

EmNumberOfVertices( *strMesh* [, *blnCountDeleted*] )

By default, this function counts the valid vertices of a mesh object, meaning that if vertices have been deleted by functions of the Scripting Interface without performing a Garbage Collection afterwards, these vertices are omitted. If the deleted vertices should be counted as well, the optional boolean parameter *blnCountDeleted* must be set to True.

**Parameters:**

| | |
|---|---|
| *strMesh* | Required. String. A string identifying the mesh object. |
| *blnCountDeleted* | Optional. Boolean. If true, vertices marked as deleted will be counted. See the documentation on Garbage Collection for more information. If omitted, defaults to False. |

**Return value:**

| | |
|---|---|
| Integer | Number of valid vertices or total vertices on the mesh. |
| Null | On error. |

## EmOppositeHalfedge

*Returns the opposite halfedge for a given halfedge.*

**Syntax:**

EmOppositeHalfedge( *arrHandle* )

EmOppositeHalfedge( *strMesh*, *strHandle* )

EmOppositeHalfedge( *strMesh*, *intIndex* )

**Parameters:**

| | |
|---|---|
| *arrHandle* | Required. Array. Unique Halfedge Handle. |
| *strMesh* | Required. String. A string identifying the mesh object. |
| *strHandle* | Required. String. Halfedge Handle |
| *intIndex* | Required. Integer. The index of the input Halfedge Handle. |

**Return value:**

| | |
|---|---|
| Array | Unique Halfedge Handle |
| Null | On error. |

## EmPoint

*Returns the coordinates of a given mesh vertex.*

**Syntax:**

EmPoint( *arrHandle* )

EmPoint( *strMesh*, *strHandle* )

EmPoint( *strMesh*, *intIndex* )

**Parameters:**

| | |
|---|---|
| *arrHandle* | Required. Array. Unique Vertex Handle. |
| *strMesh* | Required. String. A string identifying the mesh object. |
| *strHandle* | Required. String. Vertex Handle. |
| *intIndex* | Required. Integer. The index of the input Vertex Handle. |

**Return value:**

| | |
|---|---|
| Array | The 3D point coordinates of the vertex. |
| Null | On error. |

## EmPrevHalfedge

*Returns the previous halfedge for a given halfedge.*

**Syntax:**

EmPrevHalfedge( *arrHandle* )

EmPrevHalfedge( *strMesh*, *strHandle* )

EmPrevHalfedge( *strMesh*, *intIndex* )

**Parameters:**

| | |
|---|---|
| *arrHandle* | Required. Array. Unique Halfedge Handle. |
| *strMesh* | Required. String. A string identifying the mesh object. |
| *strHandle* | Required. String. Halfedge Handle |
| *intIndex* | Required. Integer. The index of the input Halfedge Handle. |

**Return value:**

| | |
|---|---|
| Array | Unique Halfedge Handle |
| Null | On error. |

## EmSetVertex

*A given vertex is moved to the given coordinates.*

**Syntax:**

EmSetVertex( *arrHandleData*, *arrPoint* )

EmSetVertex( *strMesh*, *strVertexHandle*, *arrPoint* )

EmSetVertex( *strMesh*, *intIndex*, *arrPoint* )

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Unique Vertex Handle. |
| *strMesh* | Required. String. The mesh. |
| *strVertexHandle* | Required. String. Vertex Handle |
| *intIndex* | Required. Integer. The index of the input Vertex Handle. |
| *arrPoint* | Required. Array. The 3-D point to whose coordinates the vertex should be moved. |

**Return value:**

| | |
|---|---|
| Bool | If true, the vertex has been properly moved. |
| Null | On error. |

## EmSetNormal

*The normal of a given vertex or face is set to a given vector.*

**Syntax:**

EmSetNormal( *arrHandleData*, *arrVector* )

EmSetNormal( *strMesh*, *strHandle*, *arrVector* )

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Unique Vertex or Face Handle. |
| *strMesh* | Required. String. The mesh. |
| *strHandle* | Required. String. Vertex or Face Handle |
| *arrVector* | Required. Array. The 3-D vector to whose coordinates the normal should be set. |

**Return value:**

| | |
|---|---|
| Bool | If true, the normal has been properly set. |
| Null | On error. |

## EmStartVertex

*Returns the start vertex of a given halfedge.*

**Syntax:**

EmStartVertex( *arrHandle* )

EmStartVertex( *strMesh*, *strHandle* )

EmStartVertex( *strMesh*, *intIndex* )

**Parameters:**

| | |
|---|---|
| *arrHandle* | Required. Array. Unique Halfedge Handle. |
| *strMesh* | Required. String. A string identifying the mesh object. |
| *strHandle* | Required. String. Halfedge Handle |
| *intIndex* | Required. Integer. The index of the input Halfedge Handle. |

**Return value:**

| | |
|---|---|
| Array | Unique Vertex Handle |
| Null | On error. |

## EmVectorProperty

*A vector is assigned to or retrieved from a vertex, face, edge or halfedge as a named property. Properties are not saved to 3dm files and will be lost upon quitting Rhino.*

**Syntax:**

EmVectorProperty( *arrHandleData*, *strName* [, *arrVector*] )

EmVectorProperty( *strMesh*, *strHandle*, *strName* [, *arrVector*] )

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Unique Vertex, Face, Edge or Halfedge Handle. |
| *strMesh* | Required. String. The mesh. |
| *strHandle* | Required. String. Vertex, Face, Edge or Halfedge Handle |
| *strName* | Required. String. The name of the property. Assigning a vector valued property with a new name to a mesh element will create this property for all other elements of the same type and initialize it with the vector (0,0,0). An attempt to retrieve a property with a name that doesn't exist will result in Null being returned. |
| *arrVector* | Optional. Array. If provided, this vector will be set as a property. If skipped, the current value of the property is returned. |

**Return value:**

| | |
|---|---|
| Bool | If a vector was provided, and the property has been properly set. |
| Array | If no vector was provided, the current value of the property for the given mesh element. |
| Null | On error. |

## EmVertex

*Creates a Vertex Handle from an integer .*

**Syntax:**

EmVertex( *intIndex* ) (returns Vertex Handle)

EmVertex( *strMesh*, *intIndex* ) (returns Unique Vertex Handle)

**Parameters:**

| | |
|---|---|
| *intIndex* | Required. Integer. The index of the Vertex Handle to be created. |
| *strMesh* | Required. String. A string identifying the mesh object. |

**Return value:**

| | |
|---|---|
| String or Array | Vertex Handle or Unique Vertex Handle |
| Null | On error. |

## EmVertexValence

*Returns the number of edges emanating from a mesh vertex.*

**Syntax:**

EmVertexValence( *arrHandleData* )

EmVertexValence( *strMesh*, *strVertexHandle* )

EmVertexValence( *strMesh*, *intIndex* )

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Unique Vertex Handle. |
| *strMesh* | Required. String. The mesh. |
| *strVertexHandle* | Required. String. Vertex Handle |
| *intIndex* | Required. Integer. The index of the input Vertex Handle. |

**Return value:**

| | |
|---|---|
| Integer | The number of edges emanating from the input vertex. |
| Null | On error. |

## SetReference

*All current reference objects are unset.*

**Syntax:**

ClearReference()

**Return value:**

| | |
|---|---|
| Boolean | True if set of reference objects could be cleared. |
| Null | On error. |

## DecoupleSubdivision

*Decouples objects from their parent and child meshes.*

**Syntax:**

DecoupleSubdivision( *arrObjects* )

**Parameters:**

| | |
|---|---|
| *arrObjects* | Required. Array. The mesh objects to decouple from their parent and child meshes. |

**Return value:**

| | |
|---|---|
| Boolean | If true, all objects could be decoupled. |
| Null | On error. |

## DeleteCoplanarityConstraint

*Used for deleting coplanarity constraints.*

**Syntax:**

DeleteCoplanarityConstraint( *ByVal strMesh [, ByVal intIndex ]* ) As Boolean

DeleteCoplanarityConstraint( *ByVal strMesh, ByVal arrIndices* ) As Boolean

See the documentation of the corresponding EvoluteTools Rhino command.

**Parameters:**

| | |
|---|---|
| *strMesh* | Required. String. The mesh. |
| *intIndex* | Integer. Index of coplanarity constraint to be deleted. Coplanarity constraints are sorted as returned by SetVerticesCoplanar. Be aware that indices of constraints might change when deleting a constraint. If omitted, all coplanarity constraints will be deleted. |
| *arrIndices* | Array of integers. Indices of coplanarity constraints to be deleted. Coplanarity constraints are sorted as returned by SetVerticesCoplanar. Be aware that indices of constraints might change when deleting a constraint. |

**Return value:**

| | |
|---|---|
| Boolean | True if coplanarity constraint(s) could be deleted. |
| Null | On error. |

## DeleteFairingConstraint

*Used for deleting fairing constraints.*

**Syntax:**

DeleteFairingConstraint( *ByVal strMesh [, ByVal intIndex ]* ) As Boolean

DeleteFairingConstraint( *ByVal strMesh, ByVal arrIndices* ) As Boolean

See the documentation of the corresponding EvoluteTools Rhino command.

**Parameters:**

| | |
|---|---|
| *strMesh* | Required. String. The mesh. |
| *intIndex* | Optional. Integer. Index of fairing constraint to be deleted. Fairing constraints are sorted as returned by SetVerticesFairing. Be aware that indices of constraints might change when deleting a constraint. If omitted, all fairing constraints will be deleted. |
| *arrIndices* | Required. Array of integers. Indices of fairing constraints to be deleted. Fairing constraints are sorted as returned by SetVerticesFairing. Be aware that indices of constraints might change when deleting a constraint. |

**Return value:**

| | |
|---|---|
| Boolean | True if fairing constraint(s) could be deleted. |
| Null | On error. |

## DistanceFromReference

*Returns the distance of a given vertex from the nearest reference geometry.*

**Syntax:**

DistanceFromReference( *arrHandleData* )

DistanceFromReference( *strMesh*, *strVertexHandle* )

DistanceFromReference( *strMesh*, *intIndex* )

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Unique Vertex Handle. |
| *strMesh* | Required. String. The mesh. |
| *strVertexHandle* | Required. String. Vertex Handle |
| *intIndex* | Required. Integer. The index of the input Vertex Handle. |

**Return value:**

| | |
|---|---|
| Double | The distance from the input vertex to the nearest reference geometry, if there is one. |
| Null | On error. For example if no reference geometry is set. |

## FacePlanarity

*Returns the planarity measure (diagonal distance) of a mesh face.*

**Syntax:**

FacePlanarity( *arrHandleData* )

FacePlanarity( *strMesh*, *strFaceHandle* )

FacePlanarity( *strMesh*, *intIndex* )

This function returns the diagonal distance value for a given mesh face, which is a measure indicative of the "degree of planarity" of a polygon. If a face has zero diagonal distance, it is precisely planar. See the help entry on mesh analysis for more information. If the face is an n-gon with n > 4, the returned value will be the sum of diagonal distances for all inscribed quads given by four consecutive vertices of the face.

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Unique Face Handle. |
| *strMesh* | Required. String. The mesh. |
| *strFaceHandle* | Required. String. Face Handle |
| *intIndex* | Required. Integer. The index of the input Face Handle. |

**Return value:**

| | |
|---|---|
| Double | The diagonal distance value for the input face. |
| Null | On error. |

## FacePlanarityScaleInvariant

*Returns a scale-invariant planarity measure of a mesh face.*

**Syntax:**

FacePlanarityScaleInvariant( *arrHandleData* )

FacePlanarityScaleInvariant( *strMesh*, *strFaceHandle* )

FacePlanarityScaleInvariant( *strMesh*, *intIndex* )

This function returns the diagonal distance value for a given mesh face divided by the mean length of its diagonals, which is a measure indicative of the "degree of planarity" of a polygon. If a face has zero diagonal distance, it is precisely planar. See the help entry on mesh analysis for more information. If the face is an n-gon with n > 4, the returned value will be the sum of scale invariant diagonal distances for all inscribed quads given by four consecutive vertices of the face.

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Unique Face Handle. |
| *strMesh* | Required. String. The mesh. |
| *strFaceHandle* | Required. String. Face Handle |
| *intIndex* | Required. Integer. The index of the input Face Handle. |

**Return value:**

| | |
|---|---|
| Double | The scale invariant diagonal distance value for the input face. |
| Null | On error. |

## MeshAddDiagonal

*Adds an edge between two vertices adjacent to the same face.*

**Syntax:**

MeshAddDiagonal( *arrHandleData* )

MeshAddDiagonal( *strMesh*, *arrVertexHandles* )

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Two unique vertex handles adjacent to the same face, but not already connected by an edge. |
| *strMesh* | Required. String. The mesh. |
| *arrVertexHandles* | Required. Array. An array of exactly two proper Vertex Handles. |

**Return value:**

| | |
|---|---|
| Boolean | True if edge could be added successfully. |
| Null | On error. |

## MeshAddDiagonalLine

*Adds a polyline diagonal to existing quad faces to a mesh.*

**Syntax:**

MeshAddDiagonalLine( *arrHandleData* )

MeshAddDiagonalLine( *strMesh*, *arrVertexHandles* )

Starting with a diagonal to a given quad face, this function adds diagonals to additional faces until a non-quad face, the mesh boundary or a vertex with uneven valence is encountered. See also the documentation of the corresponding Rhino command.

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Two unique vertex handles adjacent to the same quad face, but not already connected by an edge. |
| *strMesh* | Required. String. The mesh. |
| *arrVertexHandles* | Required. Array. An array of exactly two proper vertex handles. |

**Return value:**

| | |
|---|---|
| Boolean | True if polyline could be added successfully. |
| Null | On error. |

## MeshAddDiagonalLinesParallel

*Adds polylines diagonal to existing quad faces to a mesh.*

**Syntax:**

MeshAddDiagonalLinesParallel( *arrHandleData* )

MeshAddDiagonalLinesParallel( *strMesh*, *arrVertexHandles* )

Starting with a diagonal to a given quad face, this function adds diagonals to additional faces until a non-quad face, the mesh boundary or a vertex with uneven valence is encountered. This is repeated in a parallel way using a given interval to prescribe the gap between lines. See also the documentation of the corresponding Rhino command.

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Two unique vertex handles adjacent to the same quad face, but not already connected by an edge. |
| *strMesh* | Required. String. The mesh. |
| *arrVertexHandles* | Required. Array. An array of exactly two proper vertex handles. |

**Return value:**

| | |
|---|---|
| Boolean | True if polylines could be added successfully. |
| Null | On error. |

## MeshCut

*Divides a row of quadrilateral mesh faces into two by cutting each face in half.*

**Syntax:**

MeshCut( *arrHandleData* [, *blnAcceptNGons* ] )

MeshCut( *strMesh*, *arrEdgeHandles* [, *blnAcceptNGons* ] )

This command takes two mesh edges as input. If a row of quadrilateral mesh faces connects these edges, the faces along the row are cut in half by inserting new edges starting from the midpoints of the input edges. The command has two modes of operation:

- By default, the option blnAcceptNgons is set to false. This causes EvoluteTools to split the quadrilateral mesh faces at both ends of the newly inserted cut into triangles. This is done because otherwise these quad faces would contain 5 mesh vertices and therefore, technically, become pentagons. As Rhino does not support mesh faces with more than 4 vertices, this is the only way to facilitate such a mesh cut and still create a standard Rhino mesh object.
- If the option blnAcceptNgons is set to true, the triangle split is not performed. This usually leads to the creation of a polymesh object containing pentagons. These objects are not supported by Rhino and not fully functional in this version of EvoluteTools for Rhino. See the specific documentation topic about this issue for more information.

See also the documtentation of the corresponding Rhino command.

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Two unique edge handles adjacent to the same quad face but not to each other. |
| *strMesh* | Required. String. The mesh. |
| *arrEdgeHandles* | Required. Array. An array of exactly two proper edge handles. |
| *blnAcceptNGons* | Optional. Boolean. If true, the creation of faces with more than four edges will be tolerated, leading to the creation of a polymesh object. Defaults to false. |

**Return value:**

| | |
|---|---|
| Boolean | True if cut could be added successfully. |
| Null | On error. |

## MeshDeletePolyline

*Deletes a polyline of mesh edges, joining the respective pairs of adjacent faces.*

**Syntax:**

MeshDeletePolyline( *arrHandleData* [, *blnDeleteIsolatedVertices* [, *blnGarbageCollection* ] ] )

MeshDeletePolyline( *strMesh*, *arrEdgeHandles* [, *blnDeleteIsolatedVertices* [, *blnGarbageCollection* ] ] )

This command takes mesh edges as input. The polylines on which the input edges lie will be removed from the mesh. For each removed edge, the two adjacent faces are joined.

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Array of Unique Edge Handles. |
| *strMesh* | Required. String. The mesh. |
| *arrEdgeHandles* | Required. Array. Array of Edge Handles. |
| *blnDeleteIsolatedVertices* | Optional. Boolean. If true (default) isolated vertices remaining in the mesh after the deletion process is finished are deleted, too. |
| *blnGarbageCollection* | Optional. Boolean. If true (default) the mesh element indices are reset after this command, making previously saved handles invalid. See a general explanation of the subject and the documentation to the EmGarbageCollection command for more information. |

**Return value:**

| | |
|---|---|
| Integer | Number of deleted edges. |
| Null | On error. |

## MeshExtractPolylines

*Extracts all mesh polylines as Rhino polyline objects.*

**Syntax:**

MeshExtractPolylines( *arrObjects* )

**Parameters:**

| | |
|---|---|
| *arrObjects* | Required. Array. An array of strings identifying the mesh objects to extract polylines from. |

**Return value:**

| | |
|---|---|
| Array | Array of strings holding the polyline objects added to the document. |
| Null | On error. |

## MeshFlip

*Reverses the direction of a mesh (flips normals).*

**Syntax:**

MeshFlip( *strMesh* )

This function reverses the direction of a given mesh, it takes only one input, namely the mesh ID string.

**Parameters:**

| | |
|---|---|
| *strMesh* | Required. String. The mesh. |

**Return value:**

| | |
|---|---|
| Boolean | True if direction reversal is successful. |
| Null | On error. |

## MeshGetMaxClosenessError

*Returns the maximum distance of any vertex of a given mesh to the nearest reference object.*

**Syntax:**

MeshGetMaxClosenessError( *strMesh*, *blnIgnoreFarOff*, *dblSamplingDistance*, *dblCutOffDistance* )

**Parameters:**

| | |
|---|---|
| *strMesh* | Required. String. The identifier of the mesh to analyze. |
| *blnIgnoreFarOff* | Optional. Boolean. Ignore distances for vertices which are not in a cone with 90 degree opening angle from foot point. |
| *dblSamplingDistance* | Optional. Double. If *blgnoreFarOff* is provided, sampling distance which should be assumed for ignoring far-off vertices. Use ReferenceMaximumSamplingDistance for a safe estimation. |
| *dblCutOffDistance* | Optional. Double. Ignore vertices which are farther away to the closest point of the reference point cloud. |

**Return value:**

| | |
|---|---|
| Double | Maximum distance from any vertex to nearest reference. |
| Null | On error. |

## MeshGetMaxPlanarityError

*Returns a mesh object's maximum diagonal distance or scaled diagonal distance for any of its faces.*

**Syntax:**

MeshGetMaxPlanarityError( *strMesh* [, *blnScaled* ] )

**Parameters:**

| | |
|---|---|
| *strMesh* | Required. String. The identifier of the mesh to analyze. |
| *blnScaled* | Optional. Boolean. If true, the scaled planarity measure is used. If omitted, defaults to *false*. |

**Return value:**

| | |
|---|---|
| Double | Maximum scaled or absolute diagonal distance. |
| Null | On error. |

## MeshLoopCut

*Divides a row of quadrilateral mesh faces into two by cutting each face in half.*

**Syntax:**

MeshLoopCut( *arrHandleData* )

MeshLoopCut( *strMesh*, *strEdgeHandle* )

This command takes a mesh edge as input. The adjacent faces are cut in half by inserting a new edge starting from the midpoint of the input edge. The cut is then continued in both directions until a non-quad face or the mesh boundary is reached. See also the documtentation of the corresponding Rhino command.

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. One unique edge handle adjacent to at least one quad face. |
| *strMesh* | Required. String. The mesh. |
| *strEdgeHandle* | Required. String. A proper edge handle. |

**Return value:**

| | |
|---|---|
| Boolean | True if loop cut could be added successfully. |
| Null | On error. |

## MeshRemoveNGons

*Deletes all mesh faces with valence greater than 4 from a mesh.*

**Syntax:**

MeshRemoveNGons( *arrObjects* [, *blnGarbageCollection* ] )

**Parameters:**

| | |
|---|---|
| *arrObjects* | Required. Array. An array of strings identifying the meshes to remove NGons from. |
| *blnGarbageCollection* | Optional. Boolean. If true (default) the mesh element indices are reset after this command, making previously saved handles invalid. See a general explanation of the subject and the documentation to the EmGarbageCollection command for more information. |

**Return value:**

| | |
|---|---|
| Integer | Number of deleted faces. |
| Null | On error. |

## MeshTriangulateNGons

*Triangulates all faces with a valence greater than 4 by inserting a new vertex at the barycenter of the face and connecting it with all vertices.*

**Syntax:**

MeshTriangulateNGons( *arrObjects* )

**Parameters:**

| | |
|---|---|
| *arrObjects* | Required. Array. An array of strings identifying the mesh objects to triangulate NGons on. |

**Return value:**

| | |
|---|---|
| Integer | Number of triangulated NGons. |
| Null | On error. |

## Optimize

*Optimize the given mesh.*

**Syntax:**

Optimize( *strMesh*, *ByRef arrResults* )

**Parameters:**

| | |
|---|---|
| *strMesh* | Required. String. The mesh. |
| *arrResults* | Required. Two-dimensional array. Return value. Holds detailed optimization results on success. |

**Return value:**

| | |
|---|---|
| Boolean | True if optimization succeeded, False if optimization failed. |

**Example Code:**

```
Option Explicit
Call Main()
Sub Main()
  Dim et, mesh, arrResults, i
  Set et = Rhino.GetPlugInObject( "EvoluteTools for Rhino" )
  mesh = Rhino.GetObject( "Select Mesh", 32 )
  If ( et.Optimize( mesh, arrResults ) ) Then
    For i=0 To UBound( arrResults, 1 )
      Rhino.Print arrResults( i, 0 ) & " = " & arrResults( i, 1 )
    Next
  End If
End Sub
```

## OptionsImportance

*Changes or returns values for the importance of optimization goals.*

**Syntax:**

OptionsImportance( *strImportanceName, dblImportanceValue* )

**Parameters:**

| | |
|---|---|
| *strImportanceName* | Required. Name of the optimization goal to change or return. Example: "Planarity" |
| *dblImportanceValue* | Optional. Must be greater or equal to 0. If missing, the function will return the current value. |

**Return value:**

| | |
|---|---|
| Double | If *dblImportanceValue* is provided, the new value will be set and the function will return the previous value, otherwise the current value of the importance is returned. |
| Null | On error. |

## OptionsReset

*Sets all values changed by etOptionsImportance and etOptionsToggles back to their default values.*

**Syntax:**

OptionsReset()

**Return value:**

| | |
|---|---|
| Boolean | True if reset succeeded, False if reset failed. |
| Null | On error. |

## PickEdge

*Prompts the user to pick mesh edges and returns Unique Edge Handles*

**Syntax:**

PickEdge( *intMin*, *intMax* [, *strPrompt* ] )

**Parameters:**

| | |
|---|---|
| *intMin* | Required. integer. Minimum number of edges to be picked. |
| *intMax* | Required. integer. Maximum number of edges to be picked. If 0, the user is prompted to press Enter to once done picking edges. |
| *strPrompt* | Optional. String. The user prompt. |

**Return value:**

| | |
|---|---|
| Array | Array of Unique Edge Handles . |
| Null | On error. |

## PickFace

*Prompts the user to pick mesh faces and returns Unique Face Handles*

**Syntax:**

PickFace( *intMin*, *intMax* [, *strPrompt* ] )

**Parameters:**

| | |
|---|---|
| *intMin* | Required. integer. Minimum number of faces to be picked. |
| *intMax* | Required. integer. Maximum number of faces to be picked. If 0, the user is prompted to press Enter to once done picking faces. |
| *strPrompt* | Optional. String. The user prompt. |

**Return value:**

| | |
|---|---|
| Array | Array of Unique Face Handles . |
| Null | On error. |

## EmPickVertex

*Prompts the user to pick vertices and returns Unique Vertex Handles*

**Syntax:**

PickVertex( *intMin*, *intMax* [, *strPrompt* ] )

**Parameters:**

| | |
|---|---|
| *intMin* | Required. integer. Minimum number of vertices to be picked. |
| *intMax* | Required. integer. Maximum number of vertices to be picked. If 0, the user is prompted to press Enter to once done picking vertices. |
| *strPrompt* | Optional. String. The user prompt. |

**Return value:**

| | |
|---|---|
| Array | Array of Unique Vertex Handles . |
| Null | On error. |

## ReferenceMaximumSamplingDistance

*Returns maximum sampling distance of reference objects.*

**Syntax:**

ReferenceMaximumSamplingDistance()

**Return value:**

| | |
|---|---|
| Double | Maximum sampling distance of reference objects. |
| Null | On error. |

## SetEdgeFeature

*Prohibits fairing across a set of edges.*

**Syntax:**

SetEdgeFeature( *arrHandleData* [, *blnSet* ] )

SetEdgeFeature( *strMesh* [, *arrEdgeHandles*[, *blnSet* ] ] )

See the documentation of the corresponding EvoluteTools Rhino command.

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Optional. Array. Array of Unique Edge Handles. If omitted, the command returns an array of current feature edges. |
| *strMesh* | Required. String. The mesh. |
| *arrVertexHandles* | Required. Array (or String). Either a single proper Vertex Handle or an array of such Handles. |
| *blnSet* | Optional. Boolean. If true (default), the input edges are set to be optimized, otherwise they are unset. |

**Return value:**

| | |
|---|---|
| Boolean | True if edges were provided and could be set as features. |
| Array | If a mesh but no edges were provided, an array of Unique Edge Handles currently set as feature edges is returned. |
| Null | On error. |

## SetEdgeLengthOptimization

*Selects edges to be included in edge length optimization.*

**Syntax:**

SetEdgeLengthOptimization( *arrHandleData* [, *blnSet* ] )

SetEdgeLengthOptimization( *strMesh*, [ *arrEdgeHandles*, [ *blnSet* ] ] )

See the documentation of the corresponding EvoluteTools Rhino command.

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Array of Unique Edge Handles. |
| *strMesh* | Required. String. The mesh. |
| *arrVertexHandles* | Optional. Array (or String). Either a single proper Vertex Handle or an array of such Handles. If omitted, the command returns an array of Unique Edge Handles currently set for edge length optimization. |
| *blnSet* | Optional. Boolean. If true (default), the input edges are set to be optimized, otherwise they are unset. |

**Return value:**

| | |
|---|---|
| Boolean | True if edges were provided and could be selected for edge length optimization. |
| Array | If a mesh but no edges were provided, an array of Unique Edge Handles currently set for length optimization is returned. |
| Null | On error. |

## SetReference

*Sets or unsets objects to be optimization references.*

**Syntax:**

SetReference( *arrObjects* [, *blnSet* ] )

**Parameters:**

| | |
|---|---|
| *arrObjects* | Required. Array. An array of strings identifying the objects to be set as reference. |
| *blnSet* | Optional. Boolean. If true, the objects are set, otherwise unset as reference. If omitted, defaults to *true*. |

**Return value:**

| | |
|---|---|
| Integer | Number of successfully set/unset reference objects. |
| Null | On error. |

## SetRulingDirection

*Sets the ruling direction for a quad mesh to used by the Strip subdivision method.*

**Syntax:**

SetRulingDirection( *arrHandleData* )

SetRulingDirection( *strMesh*, *strEdgeHandle* )

Sets the ruling direction for suitable quad meshes by selecting one edge which is to become a ruling edge. See the documentation for the corresponding EvoluteTools Rhino command for more information.

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Unique Edge Handle |
| *strMesh* | Required. String. A string identifying the mesh object. |
| *strEdgeHandle* | Required. String. Edge Handle |

**Return value:**

| | |
|---|---|
| Boolean | True if the Ruling direction could be properly set for the mesh. |
| Null | On error. |

## SetVertexCloseToNormal

*Sets a vertex to stay close to the line defined by its normal during optimization.*

**Syntax:**

SetVertexCloseToNormal( *arrHandleData* [, *blnSet* ] )

SetVertexCloseToNormal( *strMesh [*, *arrVertexHandles* [, *blnSet* ] ] )

See the documentation of the corresponding EvoluteTools Rhino command etSetVertexRestrictedToNormal. The importance *NormalCloseness* must be set using etOptionsImportance.

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Array of Unique Vertex Handles. |
| *strMesh* | Required. String. The mesh. |
| *arrVertexHandles* | Optional. Array (or String). Either a single proper Vertex Handle or an array of such Handles. If omitted, an array of Unique Vertex Handles for which this constraint is currently activated, is returned. |
| *blnSet* | Optional. Boolean. If true (default), the constraint will be activated for the given vertices, otherwise it will be deactivated. |

**Return value:**

| | |
|---|---|
| Boolean | True if vertices were provided and the constraint could be set. |
| Array | If a mesh but no vertices were provided, an array of Unique Vertex Handle, for which this constraint is currently activated, is returned. |
| Null | On error. |

## SetVertexCorner

*Sets a vertex to be treated as a corner during optimization.*

**Syntax:**

SetVertexCorner( *arrHandleData* [, *blnSet* ] )

SetVertexCorner( *strMesh [*, *arrVertexHandles [*, *blnSet* ] ] )

See the documentation of the corresponding EvoluteTools Rhino command.

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Array of Unique Vertex Handles. |
| *strMesh* | Required. String. The mesh. |
| *arrVertexHandles* | Optional. Array (or String). Either a single proper Vertex Handle or an array of such Handles. If ommitted, the command returns an array of Unique Vertex Handles of the currently set corner vertices. |
| *blnSet* | Optional. Boolean. If true (default), the input vertices are set to be corners, otherwise they are unset. |

**Return value:**

| | |
|---|---|
| Boolean | True if vertices were provided and could be set as corners. |
| Array | If a mesh but no vertices were provided, an array of Unique Vertex Handles currently set as corner vertices is returned. |
| Null | On error. |

## SetVertexCurvePoint

*Sets a vertex to be a curve point during optimization.*

**Syntax:**

SetVertexCurvePoint( *arrHandleData [, blnSet ]* )

SetVertexCurvePoint( *strMesh* [, *arrVertexHandles [, blnSet* ] ] )

See the documentation of the corresponding EvoluteTools Rhino command.

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Array of Unique Vertex Handles. |
| *strMesh* | Required. String. The mesh. |
| *arrVertexHandles* | Optional. Array (or String). Either a single proper Vertex Handle or an array of such Handles. If omitted, an array of Unique Vertex Handles currently set as curve points is returned. |
| *blnSet* | Optional. Boolean. If true (default), the input vertices are set to be curve points, otherwise they are unset. |

**Return value:**

| | |
|---|---|
| Boolean | True if vertices were provided and could be set as curve points. |
| Array | If a mesh but no vertices were provided, an array of Unique Vertex Handle currently set as curve point is returned. |
| Null | On error. |

## SetVertexFixed

*Sets a vertex to be fixed during optimization.*

**Syntax:**

SetVertexFixed( *arrHandleData* [, *blnSet* ] )

SetVertexFixed( *strMesh* [, *arrVertexHandles [*, *blnSet* ] ] )

See the documentation of the corresponding EvoluteTools Rhino command etSetVertexFixed.

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Array of Unique Vertex Handles. |
| *strMesh* | Required. String. The mesh. |
| *arrVertexHandles* | Optional. Array (or String). Either a single proper Vertex Handle or an array of such Handles. If omitted, an array of currently fixed Unique Vertex Handles is returned. |
| *blnSet* | Optional. Boolean. If true (default), the input vertices are set to be fixed, otherwise they are unset. |

**Return value:**

| | |
|---|---|
| Boolean | True if vertices were provided and could be fixed. |
| Array | If a mesh but no vertices were provided, an array of currently fixed Unique Vertex Handle is returned. |
| Null | On error. |

## SetVerticesCoplanar

*Designates mesh vertices to be optimized for coplanarity.*

**Syntax:**

SetVerticesCoplanar( *arrHandleData [, intType [, dblImportance [, arrPlane ] ] ]* )

SetVerticesCoplanar( *strMesh [, arrVertexHandles [, intType [, dblImportance [, arrPlane ] ] ] ]* )

See the documentation of the corresponding EvoluteTools Rhino command etSetVerticesCoplanar.

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Array of Unique Vertex Handles. |
| *strMesh* | Required. String. The mesh. |
| *arrVertexHandles* | Optional. Array of Vertex Handles to be used for the coplanarity constraint. If omitted, a nested array of coplanarity constraint definitions is returned. Each coplanarity constraint is returned as an array holding the following entries:<br>(0) an array of Unique Vertex Handles<br>(1) the constraint type index as above<br>(2) the constraint importance<br>(3) the constraint plane |
| *intType* | Optional. Integer. Defines the type of coplanarity constraint. Please see the documentation of etSetVerticesCoplanar for details. Defaults to 6 (*GeneralPlane*). The following types are supported:<br>*ParallelToYZPlane* 0<br>*ParallelToXZPlane* 1<br>*ParallelToXYPlane* 2<br>*NormalToXYPlane* 3<br>*NormalToXZPlane* 4<br>*NormalToYZPlane* 5<br>*GeneralPlane* 6 (default)<br>*FixedPlane* 7<br>*FixedNormal* 8 |
| *dblImportance* | Optional. Double. The importance value for this constraint. Defaults to 1. The optimizer will use this value multiplied by the importance value *Coplanarity*. |
| *arrPlane* | Optional. Array holding a plane definition as described in the RhinoScript documentation. This parameter will only be used for types *FixedPlane* and *FixedNormal*. |

**Return value:**

| | |
|---|---|
| Boolean | True if coplanarity constraint could be added. |
| Array | If only a mesh was provided, a nested array of coplanarity constraint definitions is returned. Each coplanarity constraint is returned as an array holding the following entries:<br>(0) an array of Unique Vertex Handles<br>(1) the constraint type index as above<br>(2) the constraint importance<br>(3) the constraint plane |
| Null | On error. |

## SetVerticesFairing

*Designates mesh vertices to be optimized for additional fairing.*

**Syntax:**

SetVerticesFairing( *arrHandleData[, dblImportance ]* )

SetVerticesFairing(*strMesh [, arrVertexHandles[, dblImportance ] ]* )

See the documentation of the corresponding EvoluteTools Rhino command etSetVerticesFairing.

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Array of Unique Vertex Handles. |
| *strMesh* | Required. String. The mesh. |
| *arrVertexHandles* | Optional. Array of Vertex Handles to be used for the fairing constraint. If omitted, a nested array of fairing constraint definitions is returned. Each fairing constraint is returned as an array holding the following entries:<br>(0) an array of Unique Vertex Handles<br>(1) the constraint importance |
| *dblImportance* | Optional. Double. The importance value for this constraint. Defaults to 1. The optimizer will use this value multiplied by the importance value *FairnessCurvature*. |

**Return value:**

| | |
|---|---|
| Boolean | True if fairing constraint could be added. |
| Array | If only a mesh was provided, a nested array of fairing constraint definitions is returned. Each fairing constraint is returned as an array holding the following entries:<br>(0) an array of Unique Vertex Handles<br>(1) the constraint importance |
| Null | On error. |

## SubdivideCatmullClark

*Subdivides a mesh using the Catmull-Clark rule.*

**Syntax:**

SubdivideCatmullClark( *strMesh* )

**Parameters:**

| | |
|---|---|
| *strMesh* | Required. String. A string identifying the mesh object to subdivide. |

**Return value:**

| | |
|---|---|
| String | String identifying the child object. |
| Null | On error. |

## SubdivideDiagonalize

*Subdivides a mesh using the diagonalization rule.*

**Syntax:**

SubdivideDiagonalize( *strMesh* )

**Parameters:**

| | |
|---|---|
| *strMesh* | Required. String. A string identifying the mesh object to subdivide. |

**Return value:**

| | |
|---|---|
| String | String identifying the child object. |
| Null | On error. |

## SubdivideDual

*Subdivides a mesh using the Dual rule.*

**Syntax:**

SubdivideDual( *strMesh* )

**Parameters:**

| | |
|---|---|
| *strMesh* | Required. String. A string identifying the mesh object to subdivide. |

**Return value:**

| | |
|---|---|
| String | String identifying the child object. |
| Null | On error. |

## SubdivideDualEdge

*Subdivides a mesh using the DualEdge rule.*

**Syntax:**

SubdivideDualEdge( *strMesh* )

**Parameters:**

| | |
|---|---|
| *strMesh* | Required. String. A string identifying the mesh object to subdivide. |

**Return value:**

| | |
|---|---|
| String | String identifying the child object. |
| Null | On error. |

## SubdivideDualWithBoundary

*Subdivides a mesh using the DualWithBoundary rule.*

**Syntax:**

SubdivideDualWithBoundary( *strMesh* )

**Parameters:**

| | |
|---|---|
| *strMesh* | Required. String. A string identifying the mesh object to subdivide. |

**Return value:**

| | |
|---|---|
| String | String identifying the child object. |
| Null | On error. |

## SubdivideEdgeSplit

*Subdivides a mesh using the edge split rule.*

**Syntax:**

SubdivideEdgeSplit( *strMesh* )

**Parameters:**

| | |
|---|---|
| *strMesh* | Required. String. A string identifying the mesh object to subdivide. |

**Return value:**

| | |
|---|---|
| String | String identifying the child object. |
| Null | On error. |

## SubdivideIdentity

*Subdivides a mesh using the identity rule.*

**Syntax:**

SubdivideIdentity( *strMesh* )

**Parameters:**

| | |
|---|---|
| *strMesh* | Required. String. A string identifying the mesh object to subdivide. |

**Return value:**

| | |
|---|---|
| String | String identifying the child object. |
| Null | On error. |

## SubdivideLoop

*Subdivides a mesh using the Loop rule.*

**Syntax:**

SubdivideLoop( *strMesh* )

**Parameters:**

| | |
|---|---|
| *strMesh* | Required. String. A string identifying the mesh object to subdivide. |

**Return value:**

| | |
|---|---|
| String | String identifying the child object. |
| Null | On error. |

## SubdivideSqrt3

*Subdivides a mesh using the Sqrt3 rule.*

**Syntax:**

SubdivideSqrt3( *strMesh* )

**Parameters:**

| | |
|---|---|
| *strMesh* | Required. String. A string identifying the mesh object to subdivide. |

**Return value:**

| | |
|---|---|
| String | String identifying the child object. |
| Null | On error. |

## SubdivideStrips

*Subdivides a mesh using the strip subdivision rule.*

Make sure to set the ruling direction of the strips using *SetRulingDirection* before using this function.

**Syntax:**

SubdivideStrips( *strMesh* )

**Parameters:**

| | |
|---|---|
| *strMesh* | Required. String. A string identifying the mesh object to subdivide. |

**Return value:**

| | |
|---|---|
| String | String identifying the child object. |
| Null | On error. |

## SubdivideTriHex

*Subdivides a mesh using the TriHex rule.*

**Syntax:**

SubdivideTriHex( *strMesh* )

**Parameters:**

| | |
|---|---|
| *strMesh* | Required. String. A string identifying the mesh object to subdivide. |

**Return value:**

| | |
|---|---|
| String | String identifying the child object. |
| Null | On error. |

## VertexBallpackingRadius

*Sets or reads the ball radius for ball packing optimization.*

**Syntax:**

VertexBallpackingRadius( *ByVal arrHandleData [, ByVal dblRadius ]* ) As Integer

VertexBallpackingRadius( *ByVal strMesh, ByVal strVertexHandle [, ByVal dblRadius ]* ) As Integer

VertexBallpackingRadius( *ByVal strMesh, ByVal intIndex [, ByVal dblRadius ]* ) As Integer

See also the documentation of the importance value *BallPacking* in setOptionsImportances, and the documentation of etBallPackingMeshExtractBalls.

**Parameters:**

| | |
|---|---|
| *arrHandleData* | Required. Array. Unique Face Handle. |
| *strMesh* | Required. String. The mesh. |
| *strVertexHandle* | Required. String. Vertex Handle |
| *intIndex* | Required. Integer. The index of the input Vertex Handle. |
| *dblRadius* | Optional. Double. If given, the ball packing radius for the given vertex will be set, otherwise the current ball packing radius will be returned. |

**Return value:**

| | |
|---|---|
| Boolean | True if ball packing radius could be set. |
| Double | If dblRadius was not given, current ball packing radius for the given vertex. |
| Null | On error. |

# EvoluteTools for Rhino Scripting Toolbar

***Only available in EvoluteTools PRO***

The EvoluteTools for Rhino Scripting Toolbar provides access to handy functionality which is implemented by means of the Scripting Interface. When clicking on one of the buttons in the Scripting Toolbar, a corresponding RhinoScript function is called. These RhinoScript functions are stored in the file ScriptingExamples\EvoluteToolsForRhinoToolbarScripts.rvb contained in the EvoluteTools installation package. Feel free to browse this file and adapt the functionality to your own needs, or enhance the Scripting Toolbar by your own functions.

## etsAddPolyFace

*Creates a polygonal (n-gon) mesh face that can be welded together with an existing mesh, or subdivided independently.*
*Only available in EvoluteTools PRO*

Input:

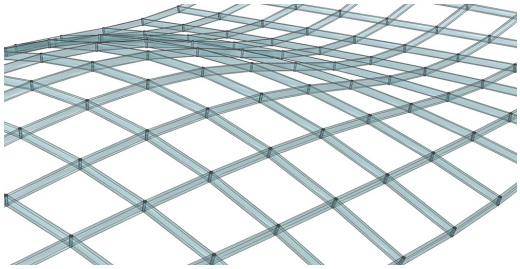
- vertex locations : the location of the new polygonal face vertices (at least 4).

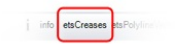

# etsBeamSubstructure

*Creates a beam substructure for rendering and presentation purposes over the mesh edges.*
*Only available in EvoluteTools PRO*

Input:

- mesh: select the mesh on which you are working
- depth of standard bars: specify the depth of the bars which have quadrilateral faces on both sides
- depth of diagonal bars: specify the depth of the bars which have one or two neighboring triangular faces
- width/depth ratio: specify the width/depth ratio for the beams



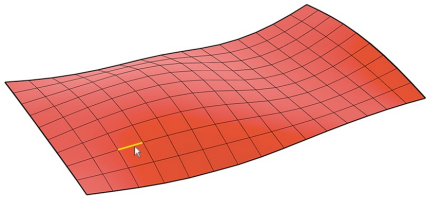



# etsCreases

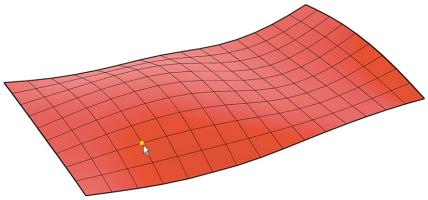
*Sets polyline edges of a quadrilateral mesh, which are located between two selected edges, as creases.*
*Only available in EvoluteTools PRO*

Input:

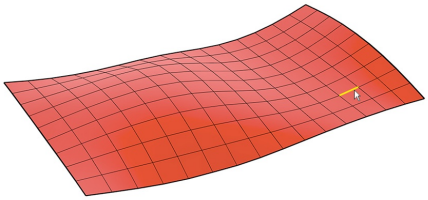
- mesh: select the mesh on which you are working
- start edge: select the first edge in the desired polyline
- direction vertex: select the end vertex of the first edge which points in the selection direction
- end edge: select the end edge of the desired polyline



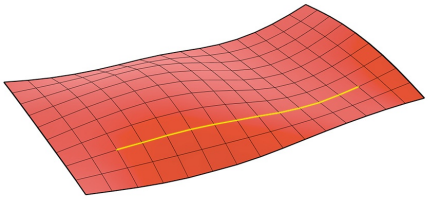
Select the start edge.



Select the direction vertex.



Select the end edge.



The result.

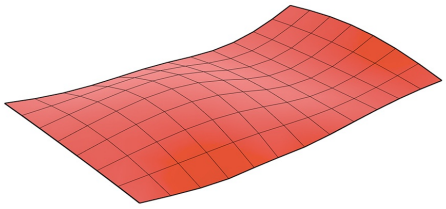

# etsDensify

*Adds density to a quadrilateral mesh by performing loop cuts until no mesh edge is longer than a specified value.*
*Only available in EvoluteTools PRO*
*Only for quadrilateral meshes*

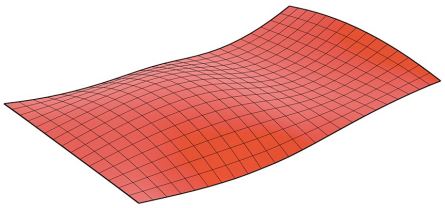
Input:

- mesh: select the mesh on which you are working
- Maximum length: specify the maximum allowed edge length for the mesh

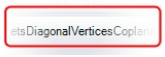
*This command needs a fair amount of Fairness curvature in the ET Optimization Options Importance settings in order to work.*
*The script will optimize the mesh automatically after each loop cut is created, so there has to be a reference surface specified.*



Before



After



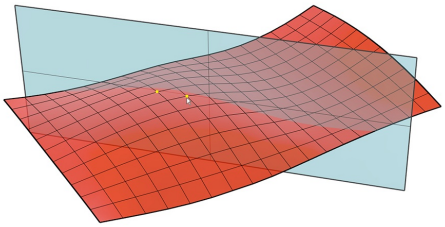

# etsDiagonalVerticesCoplanar

*Sets vertices of a quadrilateral mesh, which belong to a diagonal polyline, to be coplanar to a selected plane (surface object).*
*Only available in EvoluteTools PRO*
*Only for quadrilateral meshes*

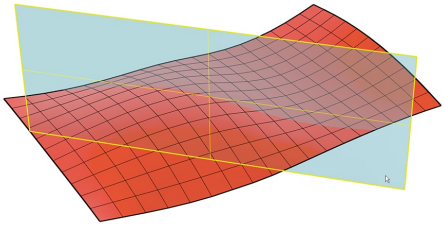
Input:

- mesh: select the mesh on which you are working
- two vertices: select two vertices of the same face which indicate the direction of the desired diagonal.
- plane (surface object): select the plane to which the vertices should become coplanar. The plane should be a Rhino surface object.

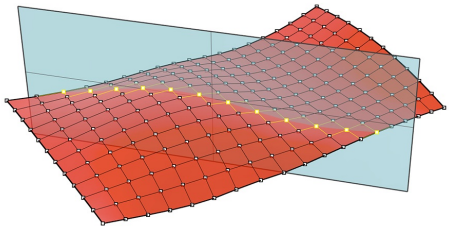
*This command needs a fair amount of Coplanarity in the ET Optimization Options Importance settings in order to work.*



Select the diagonal by clicking on a faces diagonal corners.



Select the target plane.



The script finds all vertices which belong to the desired diagonal and sets them as coplanar to the plane.
After the mesh is optimized, the vertices snap onto the target plane.

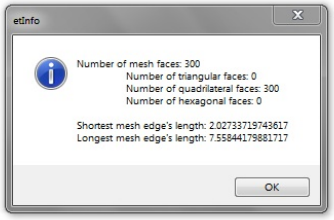
## etsInfo

*Displays information for a selected mesh object.*
*Only available in EvoluteTools PRO*

Input:

- mesh: select the mesh for which the information should be displayed.

This command returns an info screen which displays the total number of mesh faces, the number of faces of each type (triangular, quadrilateral, hexagonal…) and the lengths of the shortest and the longest mesh edge.



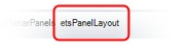
## etsNurbsFromQuadMesh

*Creates a NURBS surface based on a regular quad mesh, by interpolating the vertices of the quad mesh.*
*Only available in EvoluteTools PRO*
*Only for quadrilateral meshes*

Input:

- mesh: select the mesh on which you are working

Before
After




# etsPanelLayout

*Lays all mesh faces out as panels for production.*
*Only available in EvoluteTools PRO*

The panels are labeled with a "f" for face and an unique number. The script displays information regarding the size and planarity of the panels.
The panels are sorted on two layers (planar and curved panels) according to a user specified planarity tolerance.

Input:

- mesh: select the mesh on which you are working
- tolerance: specify the scale invariant planarity tolerance
- number of panels which should be laid out in one row

For a visual check you can look at the original mesh after the script is completed. The descriptions on the mesh faces should be oriented to the normal side of the mesh.
If they are upside down you should go back, flip the mesh, and then run the script again.


## etsPanels

*Creates panels in the mesh faces.*
*Only available in EvoluteTools PRO*

Input:

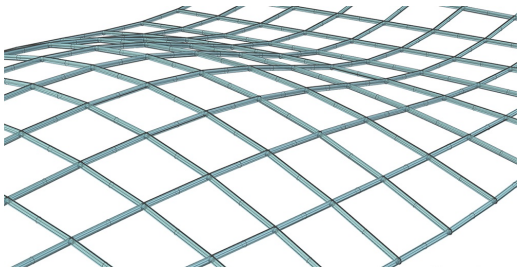
- mesh: select the mesh on which you are working
- distance from face border
- panel thickness

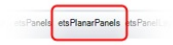

# etsPipeSubstructure

*Creates a pipe substructure for rendering and presentation purposes around the mesh edges.*
*Only available in EvoluteTools PRO*

Input:

- mesh: select the mesh on which you are working
- radius: specify the radius of the bars






## etsPlanarPanels

*Creates planar panels in the mesh faces.*
*This command works well with PQ (Planar Quadrilateral) meshes.*
*Only available in EvoluteTools PRO*

Input:

- mesh: select the mesh on which you are working
- distance from face border
- panel thickness

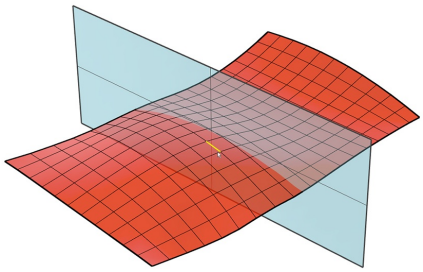

# etsPolylineVerticesCoplanar

*Sets polyline vertices of a quadrilateral mesh, to be coplanar to a selected plane (surface object).*
*Only available in EvoluteTools PRO*

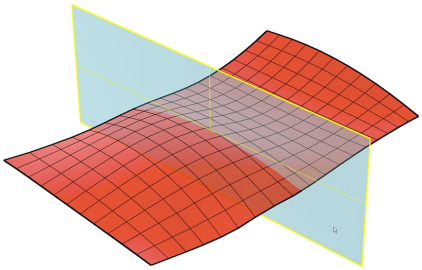
Input:

- mesh: select the mesh on which you are working
- polyline: select the polyline which should become coplanar to a plane by clicking on one edge
- plane (surface object): select the plane to which the vertices should become coplanar. The plane should be a Rhino surface object.

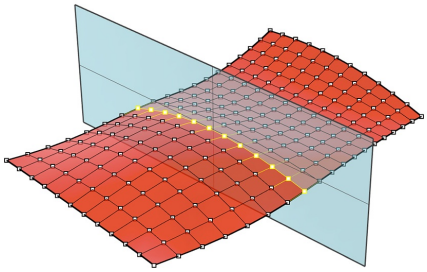
*This command needs a fair amount of Coplanarity in the ET Optimization Options Importance settings in order to work.*



Select the polyline by clicking on one of its edges.



Select the target plane.



The script finds all vertices which belong to the polyline and sets them as coplanar to the plane.
After the mesh is optimized, the vertices snap onto the target plane.

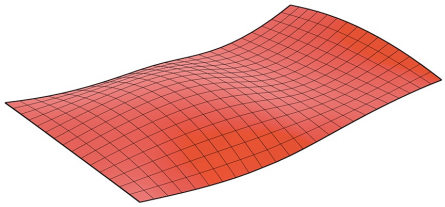

# etsUnDensify

*Makes a quadrilateral mesh less dense by deleting its polylines until no mesh edge is shorter than a specified value.*
*Only available in EvoluteTools PRO*
*Only for quadrilateral meshes*

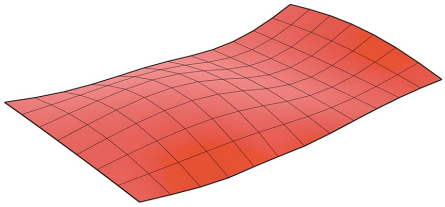
Input:

- mesh: select the mesh on which you are working
- Minimum length: specify the minimum allowed edge length for the mesh

*This command needs a fair amount of Fairness curvature in the ET Optimization Options Importance settings in order to work.*
*The script will optimize the mesh automatically after each loop cut is created, so there has to be a reference surface specified.*



Before



After