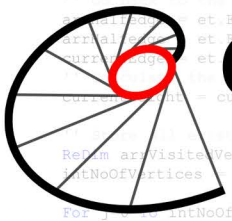


evolute tools PRO

Scripting Primer

Revision 1.0

Marko Tomičić

Introduction

Welcome! You have just opened the first edition of the EvoluteTools scripting Primer.
The introduction of parametric modeling and scripting to architecture is comparable to the discovery of perspective in the Renaissance according to Patrik Schumacher[1]. Indeed, learning to script changes the way you look at problems and how you solve them. The aim of this primer is to introduce you to the new possibilities within the Evolute Tools scripting interface and to show you how to take advantage of them. Scripting with the EvoluteTools scripting interface in RhinoScript opens new possibilities for the designer to edit, optimize and manufacture meshes/building envelopes. While EvoluteTools' obvious application field is the building skin, this work will hopefully inspire research of this software's applicability in other fields such as boat building or product design.
The primer features twenty example scripts which are explained in detail in their respective chapters. You can download the scripts and some example files together with this primer. We will go through the scripts step by step and explain all techniques that are used. Don't worry if you don't understand something at first, some things might just take a while to sink in, but after a while everything will feel natural.

I'd like to thank Alexander Schiftner, Florin Isvoranu, Mathias Höbinger and Michael Eigensatz for their help and patience during the last few months. This primer would not have been possible without them.

I hope this tutorial will teach you everything you need to know to successfully use the EvoluteTools scripting interface and that the learning process will not be too painful.

Good luck!

Marko Tomičić

[1] Patrik Schumacher, *The Autopoiesis of Architecture*, John Wiley & Sons Ltd., 2011, p 148.

Table of Contents

1. Warm Up

1.1 About EvoluteTools for Rhino

EvoluteTools for Rhino is a very powerful Rhino plugin which allows you to design beautiful and buildable panel layouts for any reference geometry you choose. It provides one of the most powerful paneling and planarization tools for freeform structures on the market. Basically, EvoluteTools works with a reference surface (NURBS surface, mesh etc.) and a coarse mesh which you will have to provide. The coarse mesh is subdivided and optimized according to specified constraints to match the reference surface as close as possible while maintaining a visually pleasing panel layout. It allows the user to specify any amount of constraints, optimize the mesh, add or remove constraints, change their importance and optimize again.

EvoluteTools for Rhino is available as a LITE (free) version and as a PRO version.

This Primer is targeted at the EvoluteTools Pro users for two reasons. One is that we will explore EvoluteTools' hidden functionality which is accessed via the EvoluteTools scripting interface and is probably something beginners will not engage into willingly. However, this Primer will show you how to take your workflow with EvoluteTools to the next level, especially when it comes to handling a huge amount of constraints and speeding up your work. The second reason is simply that the EvoluteTools Scripting interface is only supported by the Pro version of EvoluteTools for Rhino.

1.2 Overview

This Primer is divided into five parts. The first part is an introduction which will show you how to get started. In the second part you will learn about the halfedge data structure which is used in EvoluteTools. This is the way EvoluteTools stores and accesses meshes. Mastering the halfedge data structure will enable you to take advantage of the EvoluteTools mesh navigation functions which help you to perform targeted operations on the mesh. These first two parts are the inevitable theoretical parts of the Primer.

After learning about the halfedge data structure we go on to the three main workflow steps in EvoluteTools (Subdivision, Optimization and Post processing).

1. Create the reference surface
2. Create the coarse mesh
3. Subdivision
4. Optimization
5. Post processing

In the third part of the Primer "Mesh Editing" we will explore how a mesh is subdivided and try to automate and customize this process. The fourth part "Mesh Optimization" showcases example scripts which automate the optimization process with a focus on scripting the thousands of possible constraints[2] prior to optimization. The last part explores some possibilities of "Post processing". We will use the EvoluteTools scripting interface to take the building envelope from concept (digital mesh) to manufacturing.

After completing this Primer you should be able to control and automate both the mesh subdivision and the optimization process. Further, you should be able to build substructures for your mesh and to lay out the panels for production.

[2] See chapter 4.3 Constraints

1.3 Before we start…

… there are some things that you should familiarize yourself with.

Since EvoluteTools works as a plugin within Rhinoceros it wouldn't hurt to be acquainted with the basic functionality of Rhino. I assume that you are an experienced Rhino user since you're reading this Primer, but in case you are really new to Rhino you should probably take a look at some online tutorials first.

Please take a look at the video tutorials on Evolute's website because that's from where we are going to carry on in this Primer.

http://evolute.at/software/evolutetools-for-rhino/learning-center.html

We will be working exclusively on meshes so the theoretical knowledge about meshes will be of great help as well.[3]

Further, we will use the EvoluteTools Scripting interface and RhinoScript to unleash the enormous hidden powers of EvoluteTools, so it is necessary to have a basic knowledge of RhinoScript, algorithms, conditional statements, flow control, loops, arrays etc.

If you need to refresh your memory about the previously mentioned topics I heartily recommend David Rutten's **RhinoScript 101** Primer which can be downloaded from the Rhino website.

Please take a look at the EvoluteTools Help file as well, especially the part where the EvoluteTools Handles and the Garbage Collections are explained (page 67 – 73).

It is useful to have the EvoluteTools Help and the RhinoScript Help running in the background while you are working on a script.

1.4 Getting started

1.4.1 Installation

You can install EvoluteTools for Rhino by either dragging and dropping the *EvoluteToolsForRhino.rhp* file into the Rhino viewport or you could also install it via the Rhino Plugin Manager (command "*_PluginManager*" or Tools > Options > Plug-ins). You can install the EvoluteTools Pro for Rhino Toolbar by dragging and dropping as well.

Upon first use EvoluteTools will ask you to enter the serial number which you should have received during the purchasing process. If a file named *serial.txt* exists in the same directory as the plugin, its content will be filled automatically into the serial number dialog. You may use the "*etLicenseManager*" command in Rhino to manage your license(s) later.

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

[3] See chapter 2 – The Halfedge Data Structure

1.4.2 Monkey

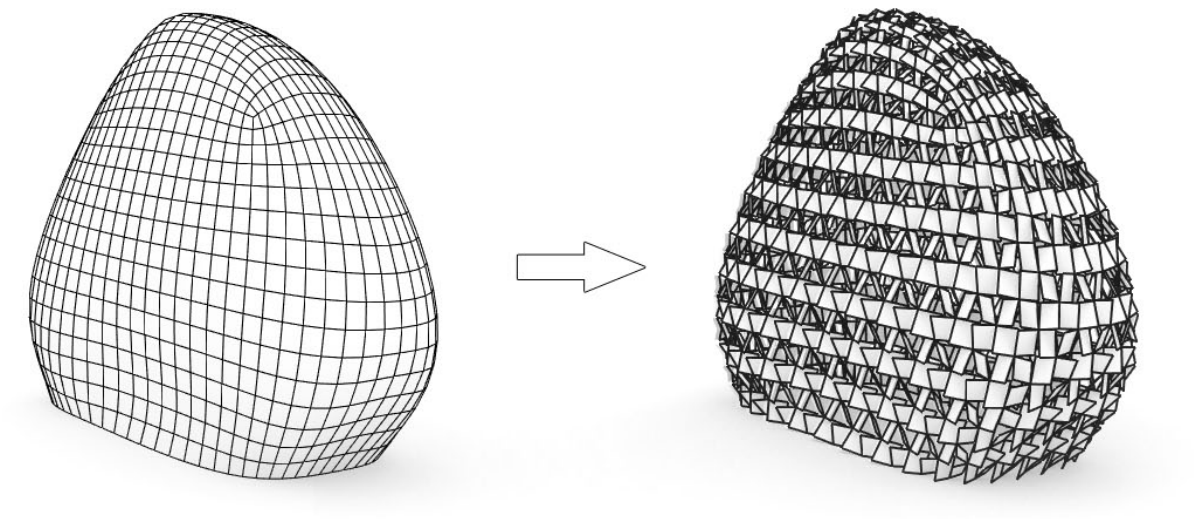
You will spend a lot of time writing scripts in your work so it makes sense to install the *Monkey RhinoScript editor* if you work with Rhino 4. EvoluteTools supports code auto completion which will make your life a lot easier. To enable the code auto completion you need first to install Monkey RhinoScript Editor (download from Rhino website) and then to copy the *EvoluteToolsForRhino.syntaxml* from your EvoluteTools folder to the subfolder Plug-ins/Monkey/Resources of your Rhino installation.

If you work with Rhino 5 you don't have to download the *Monkey RhinoScript editor* since it is integrated in Rhino 5 as "EditScript". You can install the code auto completion by running *InstallRhinoScriptSyntaxDescription.bat* from your EvoluteTools folder. This will copy the *EvoluteToolsForRhino.syntaxml* file to the correct location in Rhino4 and Rhino5.

If you experience any other trouble during the installation of the software please refer to the EvoluteTools Help file or contact the Evolute team at support@evolute.at

1.5 Run your first script

This is a small preview of what we will be doing in this Primer. It should show you where to find the example scripts and how to run them. Together with this Primer you can download the example files. Please navigate to the folder with the current chapter's number (01 Introduction) and open the Rhino file with the same number as the current subchapter (1_2_YourFirstExample). If there is no example file with some of the scripts this means that they should work on any mesh. Type *_Monkey* or *_EditScript* in Rhino's command line to access your script editor. In the script editor open the script with the name of this chapter (1_2_YourFirstScript). Run the Script.



The script will ask you to select your mesh and to specify a rotation angle. All mesh faces will be rotated around their diagonal. This is of course no valid mesh anymore. You will end up with a lot of individual surface objects representing the object's panels.

The following might come in handy in other scripts as well:
On line 23 of the script there is a function which has been commented out:

```
'Call Rhino.EnableRedraw(False)
```

The apostrophe in the beginning of the line tells the script editor that this is a comment and thus should be skipped when the script is executed. Right now, since the function is not active, all changes that happen during the execution of the script will be shown in the Rhino viewport. This is nice if you want to see what happens while your script runs. Later you can disable the Rhino viewport redraw by deleting the apostrophe in the beginning of this line of code.

```
Call Rhino.EnableRedraw (False)
```

This will make the script complete a lot faster. Don't forget to enable the viewport redrawing again after your script finishes like it is done on line 44:

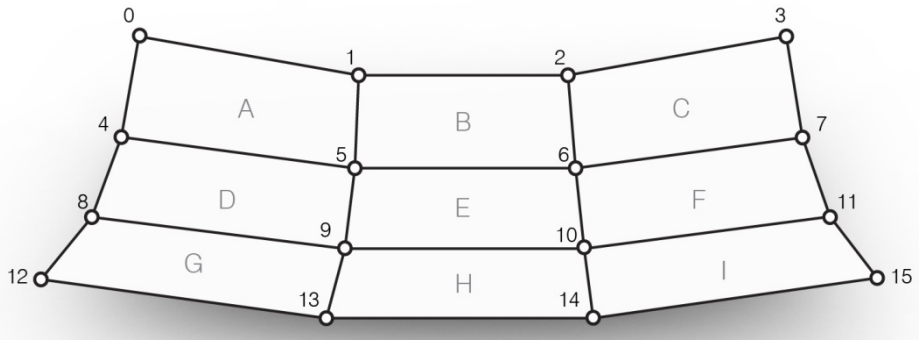
```
Call Rhino.EnableRedraw (True)
```

I'm sure it is fun to execute scripts that are working well already. In order for you to be able to write your own, we need to start with some theory first…

2. The Halfedge Data Structure

2.1 Rhino's mesh storage

In order to be able to use the EvoluteTools scripting interface it is very important to understand the halfedge data structure. There is no way around this since everything you do with the EvoluteTools scripting interface will be based on the halfedge data structure. I guess you are familiar with the basic concepts of how meshes work since you are reading this, but it is important to know the difference between the way Rhino stores meshes and the way EvoluteTools does it (Halfedge Data Structure!). Now, in Rhino the mesh is stored in two lists. The vertices with their coordinates and a unique index for every vertex are stored in the first list. The second list contains the face identifiers and the identifiers of their respective vertices. For the mesh in the image below we would have an array containing sixteen vertices with their x, y and z coordinates, and another array of nine faces where every face would store the indices of its four vertices.

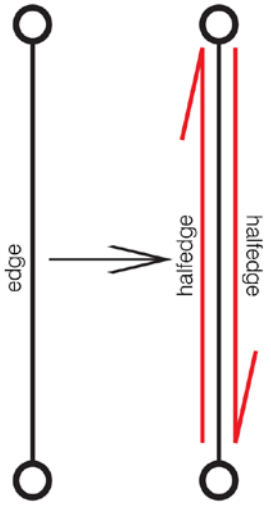


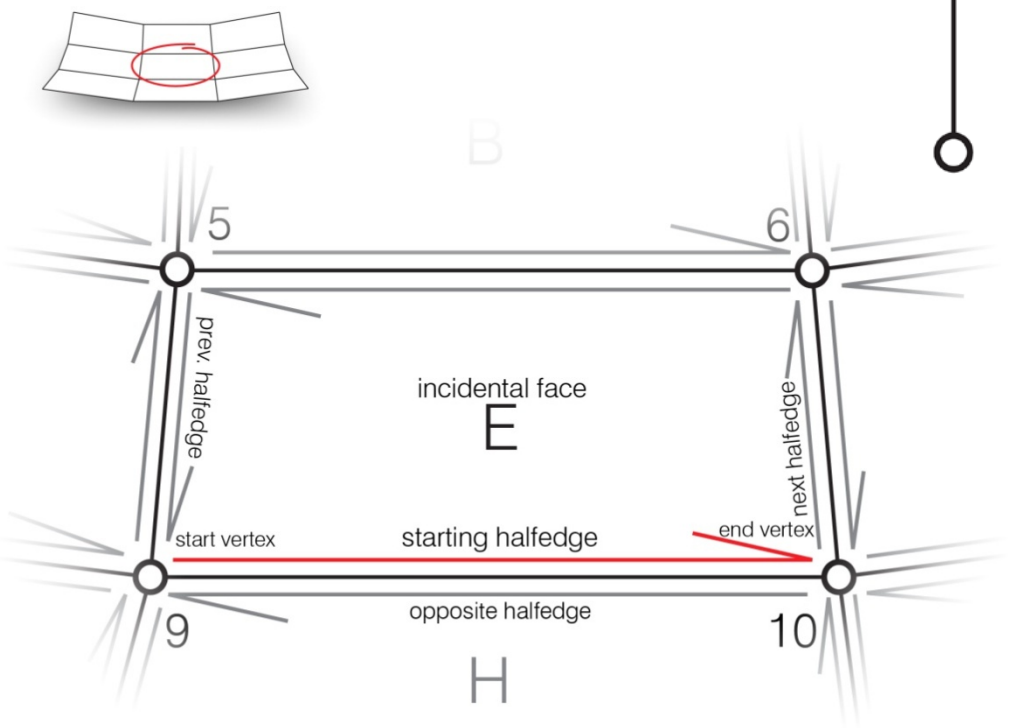
This is an efficient way to store a mesh; however, it is not the optimal method if you need to have full navigational control over the mesh.

Let's say we want to select one vertex and then do something with all its incidental faces. For this task we would have to scan the second list where the vertex combinations are stored and search for all the faces which have stored the vertex we selected. This can be a really time consuming task, especially when you start working on complex meshes with thousands of faces. The halfedge data structure enables us to have full control over the mesh and to perform such operations in a more effective way.

2.2 Halfedges

The halfedge data structure is an edge-based structure in which each edge is split up into two contrarily directed edges (see image on the right). We will use those halfedges to navigate the mesh or parts of it, independent of the position of the vertices in space. Each halfedge has one start and one end vertex. The end vertex of one halfedge is the start vertex of its next halfedge and of its opposite halfedge.



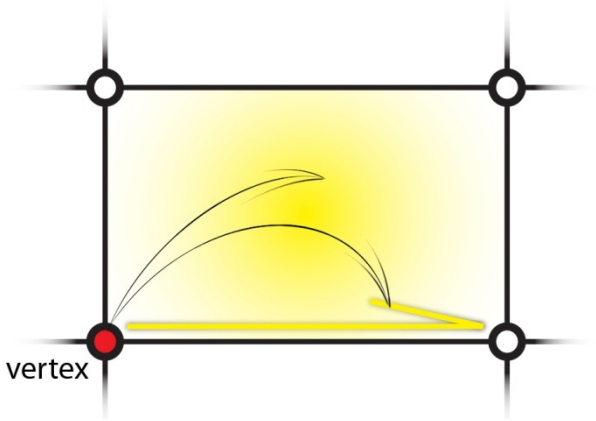


Let's assume we somehow selected the starting halfedge shown in red (you will never actually select it because it is not a physical object in Rhino but rather a concept). When we got hold of the starting halfedge we automatically know which edge it belongs to (the edge between vertices 9 and 10), which opposite halfedge it belongs to, we know the start and end vertex (9 and 10 respectively in this case) and we know which face it belongs to (face E). This is information that is stored in the halfedge and as you will learn it is easy to extract this data.

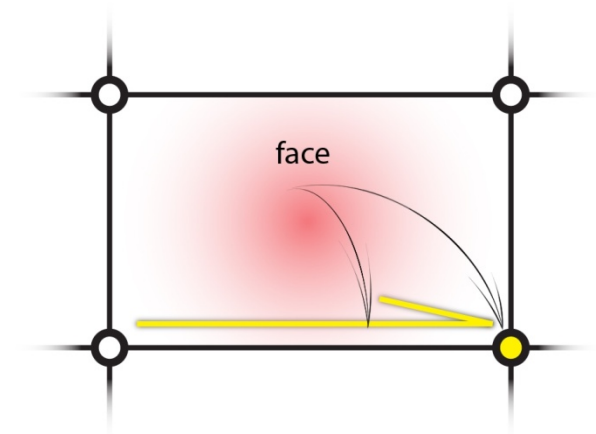
Each edge points to two opposing halfedges, even if it is a boundary edge. Those are two opposite halfedges. Notice how the halfedges of one face form a loop. This is because every halfedge has a predecessor (previous halfedge) and a successor (next halfedge). All halfedges are part of some loop which either encloses a mesh face (like on the image above) or loops around a hole. Those latter loops are called mesh boundaries, and the halfedges within this loop are called boundary halfedges. These halfedges are special in that they do not point to a valid face. It might be useful to know that the looping of the halfedges usually follows the right hand convention, i.e. loops rotate around the direction of mesh normals according to the right hand rule.

2.3 Interconnections

The navigation in the halfedge data structure is based on the interconnection of the mesh elements and the referencing between them. Every element stores information about another one and vice versa.



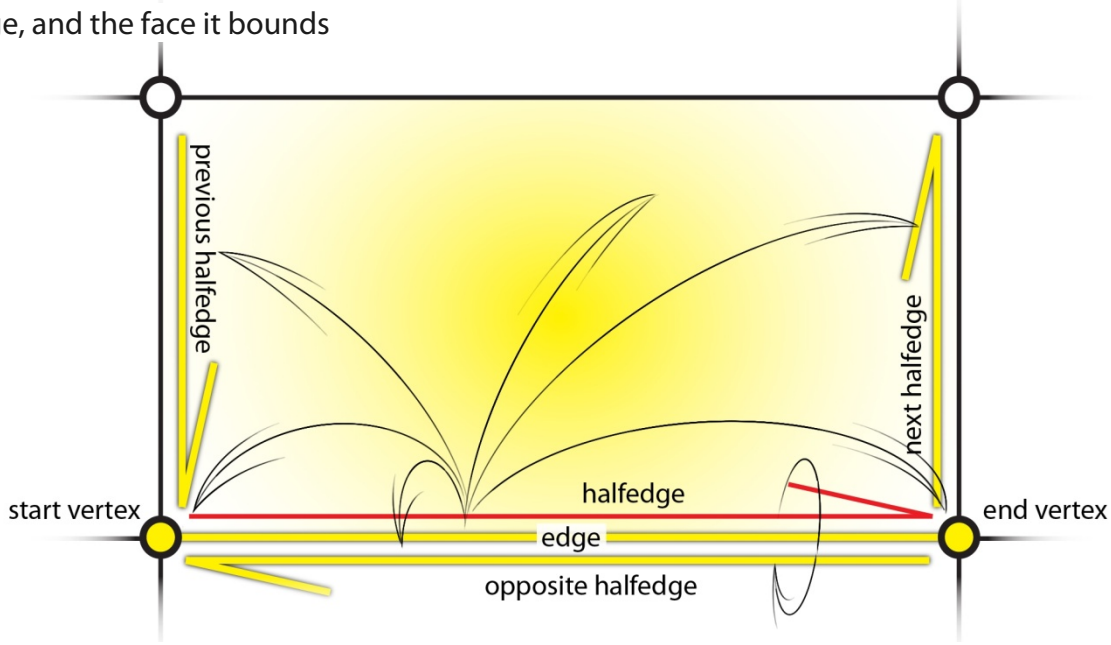
Each vertex references to one of the halfedges emanating from it, i.e. one of the halfedges starting at this vertex. There is no specific rule defining which of the outgoing halfedges is referenced. We will see later on that this information suffices to determine all the mesh elements adjacent to the vertex.



Each face stores a reference to one of its bounding halfedges. Again, this is sufficient information to determine all the mesh elements adjacent to the face, and there is no specific rule defining which of the bounding halfedges is referenced.

Each halfedge stores its start and end vertex, its previous and next halfedge, its edge, its opposite halfedge, and the face it bounds

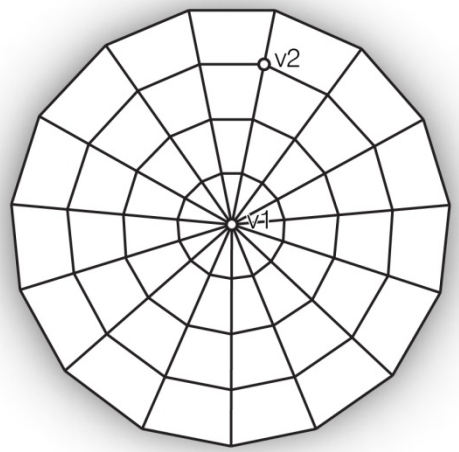
.



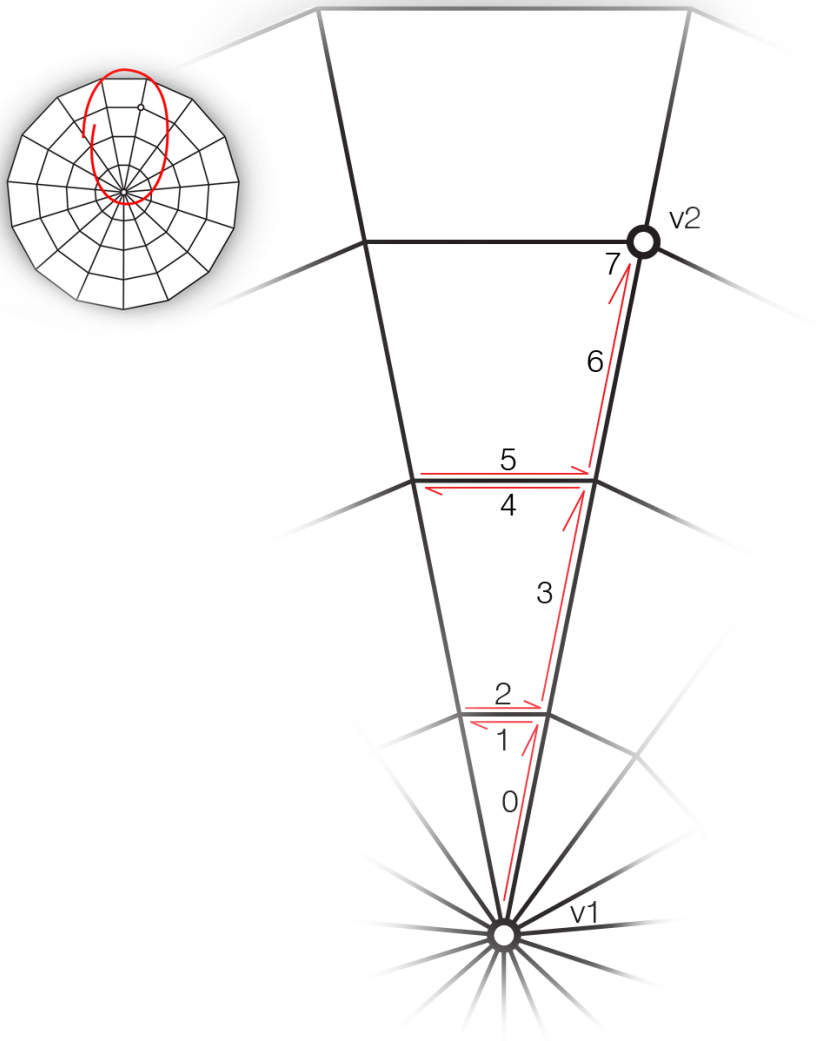
You recognize from the above, that the major part of information about the mesh connectivity is stored with the halfedges. In most cases we will use the vertex handle or the face handle just to get hold of a halfedge in order to complete a task. The important thing for us is, as you may have noticed, that there is a link from vertices, faces and edges to the halfedges and back.

2.4 Navigating the Halfedge Data Structure

Since you now know the theory behind the halfedge data structure it's time to teach you how to use it properly. The task is to find your way from point A to point B by moving from halfedge to halfedge. Take a look at the image below and try to sketch the shortest way from vertex v1 to vertex v2 using the halfedges to move along.



This is what we need to do: We need to move along the halfedges until we hit the halfedge which has stored the vertex v2 either as its start point or as its end point. We start at the halfedge 0 and move to its successor (1). From there we change to the opposite halfedge (2) and move two times to the next halfedge (3 and 4). We switch again to the opposite halfedge (5) and once more we move on to the next halfedge (6). Now that we are at the halfedge which has the vertex v2 (7) in this case as its end point it is easy for us to extract it.



2.5 Accessing the Halfedge Data Structure with EvoluteTools

The first thing that we need to do in order to get access to the halfedge data structure is to load the EvoluteTools scripting interface object in RhinoScript. For this we need to define a variable (et) and call the *Rhino.GetPluginObject* function. Those two lines are mandatory in the beginning of every script in which you want to use the EvoluteTools functions.

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

The EvoluteTools object has a lot of functions which will be used throughout the entire Primer. They will be discussed later as we stumble upon them. You can find detailed information about all EvoluteTools functions in the help file. The EvoluteTools functions use *handles* to modify the mesh. We will first take a look at the handles and then at some basic ET functions.

2.5.1 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:

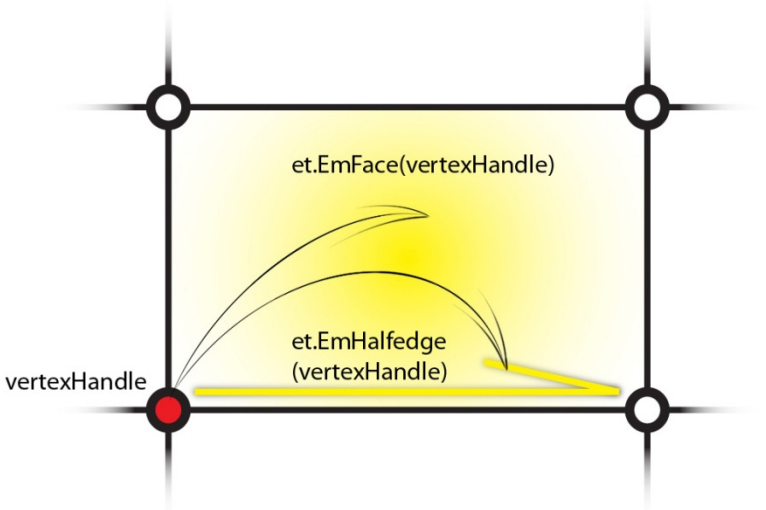
Element	Prefix	Example
vertex	v	v143
halfedge	h	h0
edge	e	e9
face	f	f9342

Those strings are vertex handles, halfedge handles…

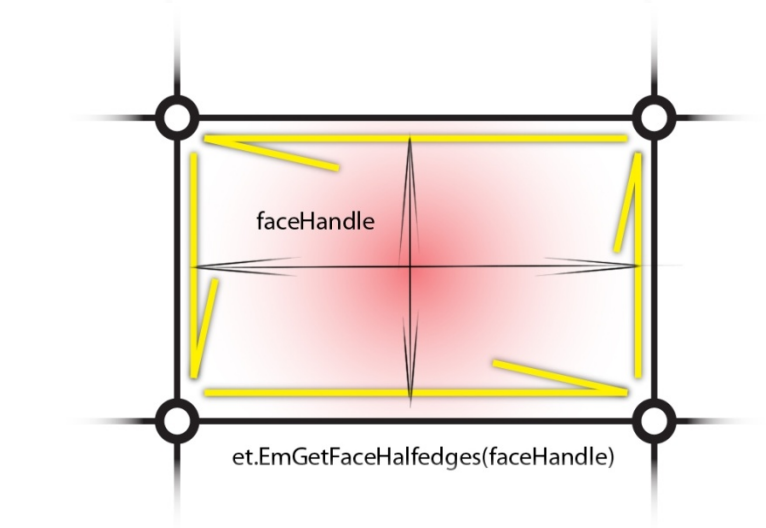
These handles cannot be properly used without identifying the Rhino mesh object which the described element is a part of. An array of strings consisting of one unique mesh ID and one EvoluteTools Handle is called **Unique EvoluteTools Handle**. If the type needs to be specified, the terms **Unique Vertex Handle** etc. are used, respectively. Such an array needs to have exactly two elements. The first element (0) is always the mesh ID and the second element (1) is the EvoluteTools handle.

2.5.2 EvoluteTools Functions

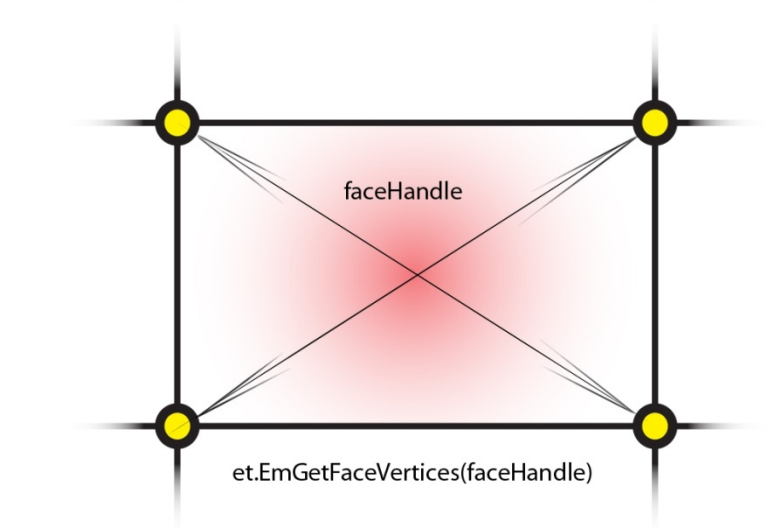
In chapter *2.3 Interconnections* we discussed how the mesh connectivity is stored by means of references between the mesh elements. This chapter will show you the most important functions for mesh navigation. Please refer to the help file for a detailed explanation of all EvoluteTools functions. You will remember the images below from chapter 2.3.



The incidental halfedge of a vertex is found with the et.EmHalfedge function. For reasons of convenience, the face bounded by this halfedge can be found using et.EmFace.



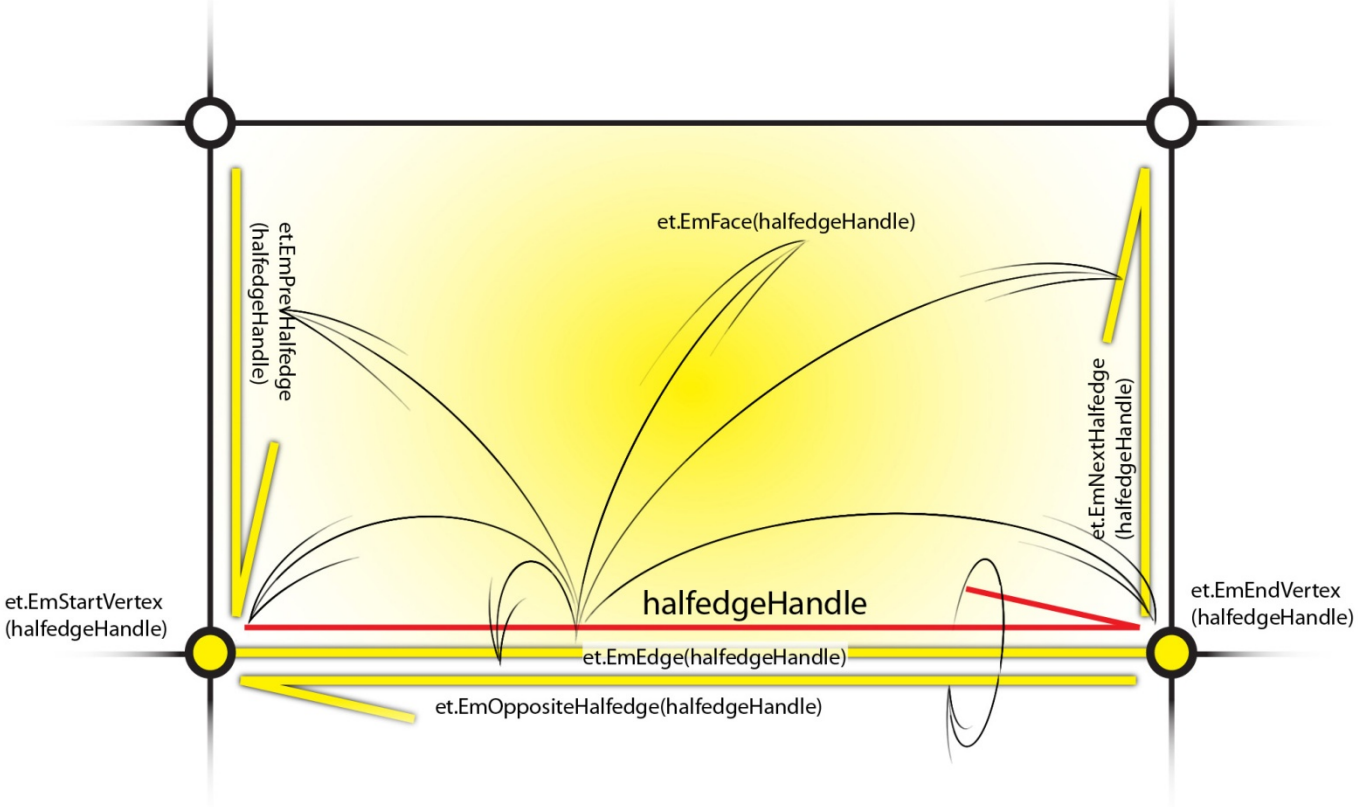
Although the face stores only one halfedge which can be retrieved by et.EmHalfedge, there are functions in EvoluteTools to extract all of them at once. The face's halfedges are accessed with the *et.EmGetFaceHalfedges* function



A face's vertices are accessed with the *et.EmGetFaceVertices* function.

The *et.EmGetFaceHalfedges* function and the *et.EmGetFaceVertices* function return arrays ordered according to the halfedge loop.

As mentioned above, the halfedge stores the majority of connectivity information. We can use it to access all other elements in the mesh. The following image shows some functions to do that.



The following examples will teach you to use this and other ET functions for mesh navigation and to access any part of the mesh that you want.

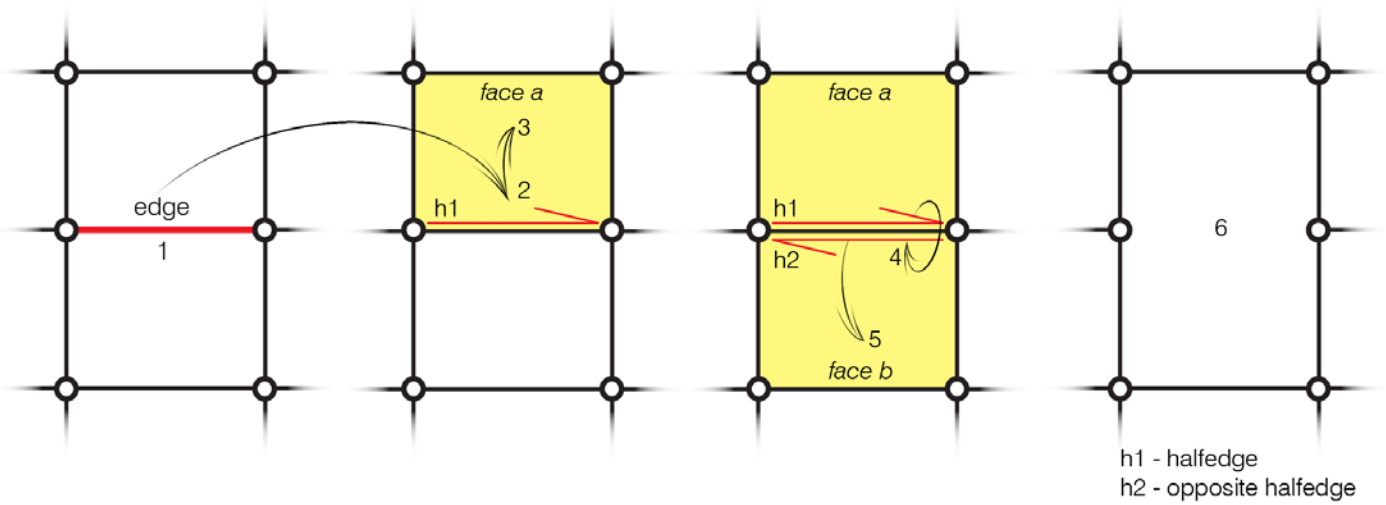
.

2.6 Examples

On the following pages there will be six example scripts which will help you to understand the navigation techniques used in the EvoluteTools scripting interface. We will start with a short and simple script and work or way up to a little bit more complicated examples.

2.6.1 Delete Edge's Faces

The first script asks you to select an edge, finds the adjacent faces and deletes them. It does so by finding first one halfedge of the selected edge. It then remembers the adjacent face of that edge by storing it into an array. The script advances to the opposite halfedge and finds its adjacent face. Finally the two faces are deleted.



1. Select edge
2. Find halfedge
3. Store adjacent face
4. Go to opposite halfedge
5. Store adjacent face
6. Delete

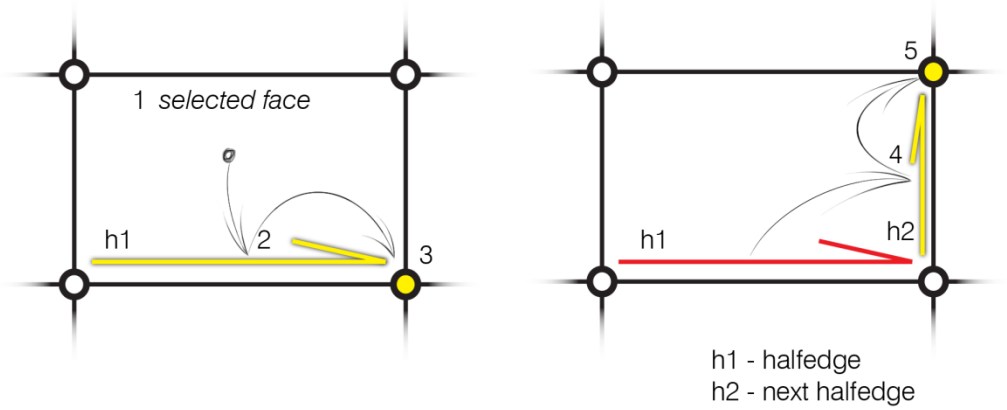
```
1 Option Explicit
2 'Script written by Marko Tomicic
3 'Script copyrighted by Evolute GmbH
4
5 Call EdgeDeleteAdjacentFaces ()
6 ' This script will ask the user to select an edge, it then finds
7 ' the adjacent faces of that edge and deletes them
8 Sub EdgeDeleteAdjacentFaces ()
9     ' Declaration of variables
10    Dim et, edge, halfedge, face
11
12    ' Get EvoluteTools plugin object
13    Set et = Rhino.GetPlugInObject ("EvoluteTools for Rhino")
14    If IsNull(et) Then Exit Sub
15
16    ' Get an edge, and one of the corresponding halfedges
17    edge = et.PickEdge(1, 1, "Select an edge")
18    If IsNull(edge) Then Exit Sub
19    halfedge = et.EmHalfedge (edge)
20
21    ' Find adjacent faces and delete them
22    face = et.EmFace(halfedge)
23    et.EmDeleteFace (face)
24    face = et.EmFace(et.EmOppositeHalfedge (halfedge))
25    et.EmDeleteFace (face)
26 End Sub
```

We declare our variables on line 10. After that we have to load the EvoluteTools scripting interface object in each and every script when we want to use EvoluteTools. This is done as shown on line 13. With the *et.PickEdge* function we ask the user to select an edge by clicking it in the viewport (line 17). Now we need to find the two adjacent faces of the selected edge. For this we will use the halfedge data structure, therefore the halfedge of the selected edge is extracted on line 19. The first face is found with the *et.EmFace* function on line 22 and deleted on the following line. For the second face we will redefine the *face* variable. This time we will extract the face belonging to the opposite halfedge of the halfedge that we found on line 19. The second face is deleted on line 25. If a boundary edge is selected there will only be one face to delete. The empty face belonging to one of the two halfedges of the selected boundary edge will be skipped.

2.6.2 Get Face's Vertices

Now you know how to get a halfedge from another handle e.g. an edge, and how to find another handle which is stored in that halfedge (in the previous example it was the faces belonging to the selected edge).

You also saw an example of how to get from one halfedge to its opposite halfedge (Line 21). In the next example there is a script which asks the user to select a mesh face and returns that face's vertices. We will use the *et.EmNextHalfedge* function to navigate our way around the border of the face while selecting the end vertex of every new halfedge we visit.



1. Get face
2. Get its halfedge
3. Find end vertex of the halfedge
4. Go to next halfedge
5. Find end vertex of the halfedge
6. Repeat this until you hit the first halfedge again

```
1 Option Explicit
2 'Script written by Marko Tomicic
3 'Script copyrighted by Evolute GmbH
4
5 Call FaceVerticesUi ()
6  ' This script works with triangular and quadrilateral meshes
7 ' This script will return an array of vertex handles for a given face
   handle.
8 ' This function should be used for demonstration purposes only, et.
   EmGetFaceVertices is
9 ' a convenience function which provides exactly its functionality.
10 Function FaceVertices (ByVal face)
11     FaceVertices = Null
12
13     ' Declaration of variables
14     Dim et, halfedge, startedge, i
15
16     ' Get EvoluteTools plugin object
17     Set et = Rhino.GetPlugInObject ("EvoluteTools for Rhino")
18     If IsNull(et) Then Exit Function
19
20     ' Get halfedge, remember starting edge
21     arrHalfedge = et.EmHalfedge (face)
22     arrStartEdge = et.EmEdge(arrHalfedge)
23
24     ' The arrVertices array is defined according to the face's valence
25     ReDim arrVertices ( et.EmFaceValence (face) - 1 )
26
27     ' Loop through all halfedges bounding the selected face
28     ' and add their end vertices to the array
29     i = 0
30     Do
```

```
31          arrVertices (i) = et.EmEndVertex (arrHalfedge)
32          '' Advance
33          arrHalfedge  = et.EmNextHalfedge (arrHalfedge)
34          '' Loop until we hit the starting edge
35          i = i+1
36      Loop While et.emEdge(arrHalfedge)(1)  <> arrStartEdge(1)
37
38      FaceVertices  = arrVertices
39  End Function
40
41  'This function serves as a user interface for demonstrating VertexFaces.
42  Sub FaceVerticesUi
43      ' Declaration of variables
44      Dim et, face, arrVertices
45
46      ' Get EvoluteTools plugin object
47      Set et = Rhino.GetPlugInObject ("EvoluteTools for Rhino")
48      If IsNull(et) Then Exit Sub
49
50      ' Ask user to select a face
51      face = et.PickFace(1, 1)
52      If IsNull(face) Then Exit Sub
53
54      ' Determine the adjacent vertices
55      arrVertices = FaceVertices (face)
56      If IsNull(arrVertices ) Then
57          Rhino.Print "FaceVertices failed."
58      Else
59          Rhino.Print "Face " & face(1) & " is adjacent to " & UBound(
      arrVertices ) + 1 & " vertices."
60      End If
61
62  End Sub
```

After declaring some variables and loading the EvoluteTools scripting interface object we ask the user to select one face from a mesh (line 51). The next thing that we need to do in the main subroutine is to call the function *FaceVertices* which will return an array containing the selected face's adjacent vertices (line 55). The function starts again by defining its own variables and calling once more the EvoluteTools plugin object (line 14-18). After that we find one halfedge which belongs to the selected face (line 21). The edge belonging to this halfedge is marked *arrStartEdge*(line 22). This is because we will move in a loop from one halfedge to the next. We want to know when we arrive at the same edge where we started our journey because then we will have visited all the face's halfedges. We will start a loop from our starting halfedge in which we will move to the next halfedge in every iteration and add the end vertex of the current halfedge to an array. Before entering the loop we need to define the *arrVertices* variable as an array with one empty space for each vertex (line 25).

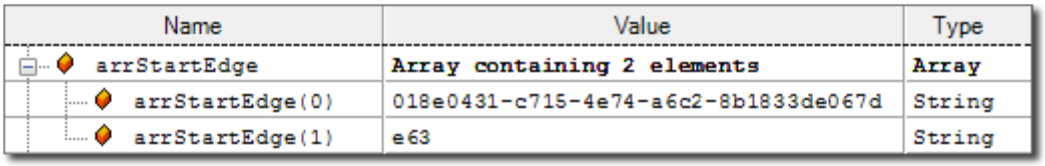
In each new iteration of the loop we will fill the current free space indicated by the variable "i" in the array with the end vertex of the current halfedge (line 31). Before the loop finishes we need to prepare our *arrHalfedge* variable for the next round. The variable needs to store the next halfedge in the circle of face halfedges that we are moving along (line 33). The loop will repeat until a full circle through all halfedges belonging to the face is completed. The loop ends when the *arrHalfedge's* edge becomes the same as our *arrStartEdge* (line 36).

You will have noticed that there is a number one in parenthesis next to our two edges on line 36:

```
Loop While et.emEdge(arrHalfedge )(1) <> arrStartEdge(1)
```

This is because edges are stored as arrays which consist of two elements. The first is the ID of the mesh this element belongs to and the second is the edge handle. Since both elements that we want to

compare belong to the same mesh the mesh ID is obviously the same for both. If we would use (0) instead of (1) the loop would end after the first repetition.

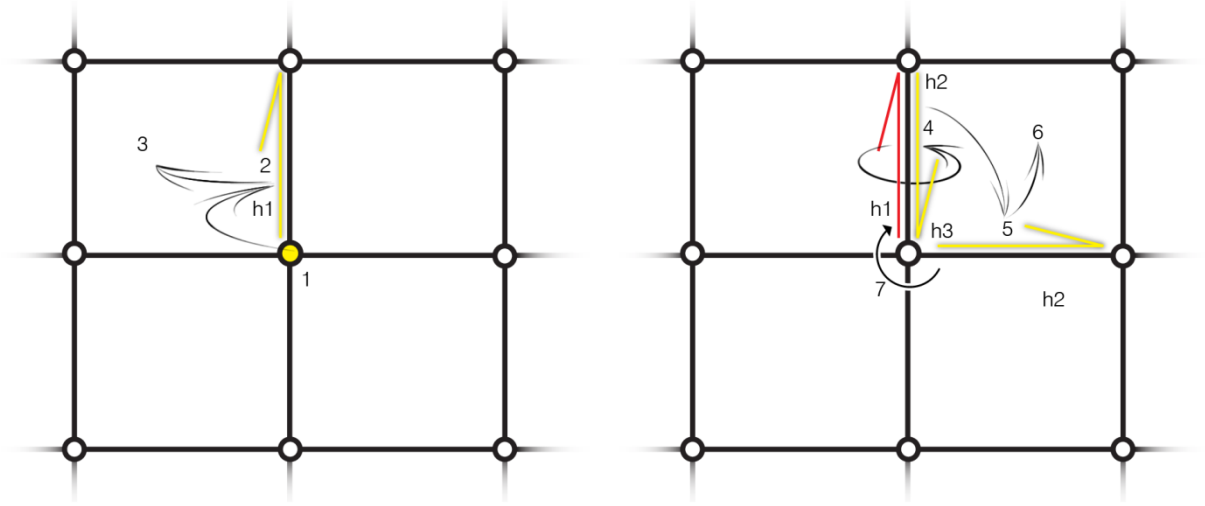


Name	Value	Type
arrStartEdge	**Array containing 2 elements**	**Array**
arrStartEdge(0)	018e0431-c715-4e74-a6c2-8b1833de067d	String
arrStartEdge(1)	e63	String

In the image above you can see that *arrStartEdge (1)* stores the edge handle. This information can be compared to other edge handles in order to know when we hit the edge where we started from.

2.6.3 Vertex Faces

In the first script we used the *et.EmOppositeHalfedge* function to find the opposite halfedge of the halfedge we already had a grip on. In the second script we used the *et.EmNextHalfedge* function to move along the border of a face from one halfedge to the other. In the next example we will combine them to move first to the opposite and then to the next halfedge. The goal is to ask the user to select one vertex and find all the incident faces of that vertex.



h1 - halfedge
h2 - opposite halfedge
h3 - next halfedge

1. Get vertex
2. Get halfedge
3. Get face
4. Go to opposite halfedge
5. Go to next halfedge
6. Store halfedges face
7. Repeat this until you hit the first halfedge again

```
Option Explicit
'Script written by Marko Tomicic
'Script copyrighted by Evolute GmbH

Call VertexFacesUi ()
' This script finds the adjacent faces of a given vertex
' and returns them in an array.
Function VertexFaces (ByVal vertex)
   VertexFaces = Null

    ' Declaration of variables
    Dim et, halfedge, face, i, intVertexValence
    ReDim arrFaces(-1)

    ' Get EvoluteTools plugin object
    Set et = Rhino.GetPlugInObject ("EvoluteTools for Rhino")
    If IsNull(et) Then Exit Function

    ' Get the vertices' halfedge, and its valence
    halfedge = et.EmHalfedge (vertex)
    intVertexValence = et.EmVertexValence (vertex)

    ' Loop through all outgoing halfedges of the vertex,
    ' add their adjacent face (if any) to the array.
    For i=0 To intVertexValence - 1
        face = et.EmFace(halfedge)
        If Not et.EmIsBoundary (halfedge) Then
            ReDim Preserve arrFaces( UBound(arrFaces) + 1 )
            arrFaces(UBound(arrFaces)) = face
```

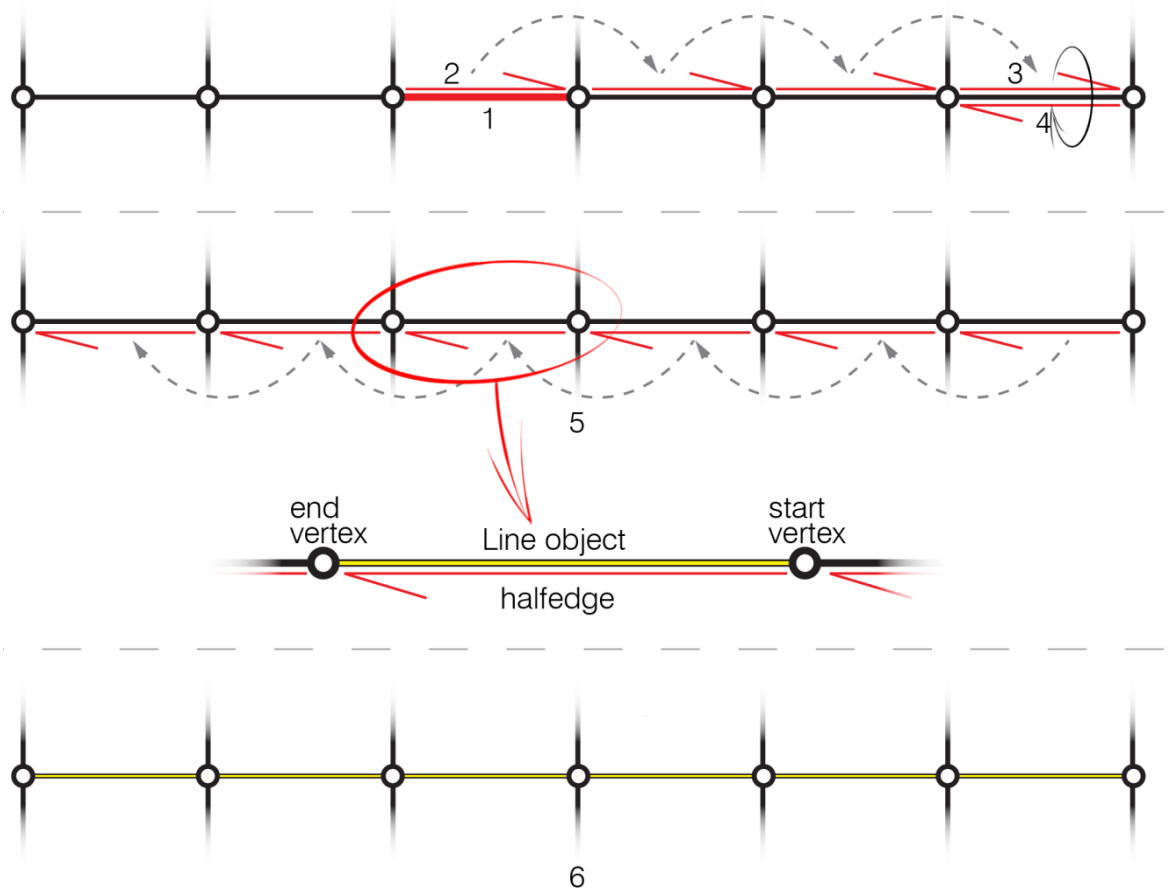
```
30          End If
31          ' Proceed to next halfedge
32          halfedge = et.EmNextHalfedge (et.EmOppositeHalfedge (halfedge))
33      Next
34      VertexFaces = arrFaces
35  End Function
36
37  ' This function serves as a user interface for demonstrating VertexFaces.
38  Sub VertexFacesUi
39      ' Declaration of variables
40      Dim et, vertex, arrFaces
41
42      ' Get EvoluteTools plugin object
43      Set et = Rhino.GetPlugInObject ("EvoluteTools for Rhino")
44      If IsNull(et) Then Exit Sub
45
46      ' Ask user to select a vertex
47      vertex = et.PickVertex (1, 1)
48      If IsNull(vertex) Then Exit Sub
49
50      ' Determine the number of adjacent faces
51      arrFaces = VertexFaces (vertex)
52      If IsNull(arrFaces) Then
53          Rhino.Print "VertexFaces failed."
54      Else
55     Rhino.Print "Vertex " & vertex(1) & " is adjacent to " & UBound(arrFaces) + 1 & "
       faces."
56      End If
57  End Sub
```

We will use a function which will always return the adjacent faces to an input vertex. The vertex is provided by the part of the script between lines 38 and 57. After declaring the variables and calling the EvoluteTools plugin object (line 44-44) the user is asked to pick a vertex from the scene (line 47). On line 51 we call the *VertexFaces* function (line 8-35), which will return the adjacent faces of the selected vertex. When starting a function we have to declare the function's own variables and set the EvoluteTools plugin object again (line 12–17). Note, inside this function the *vertex* variable holds the vertex provided by the user on line 47. The function now finds the vertices' halfedge and its valence, which is equal to the number of adjacent faces (line 20-21). The function enters a loop through the adjacent faces. The face belonging to the current halfedge is extracted and stored in the *face* variable. If the current halfedge is not a boundary halfedge, which would mean that there is no face associated with it, then the *arrFaces* array is redefined and one free space is added on its end (line 25-28). The current face is stored in the newly created spot of the *arrFaces* array (line 29). The function proceeds to the next halfedge. It starts at the current halfedge and moves first to the opposite halfedge and from there to the next halfedge (line 32)[3]. The function's output is defined on line 34. It means that the output of the function VertexFaces is the array arrFaces, so the function returns the adjacent faces of a vertex. When the function is finished the main subroutine continues where it left off before. Now the variable *arrFaces* on line 51 holds the faces which were computed in the function. We could now take those faces and manipulate them however we want but for this example we will print our result to the command line and leave it at that.

[3] See image above

2.6.4 Select Polyline

The three previous examples are the most basic examples for navigating the halfedge data structure. It is time to move on and try something a bit more complicated. The following script asks the user to select a polyline of the mesh. The selected polyline is redrawn as a curve object in the Rhino viewport. The image below explains what needs to be done.
This script is designed to work with quadrilateral meshes.



1. Get Edge
2. Get halfedge
3. Move to the end of the polyline
4. Turn around
5. Move to the opposite end and create line objects from start vertex to end vertex of every visited halfedge.
6. Join lines

```
Option Explicit
'Script written by Marko Tomicic
'Script copyrighted by Evolute GmbH

Call SelectPolyline()
' This script finds the polyline which belongs to a selected edge and draws it as a
' curve object in the Rhino viewport. This script works only with quadrilateral meshes.
' This script works with quadrilateral meshes
Sub SelectPolyline()
  Dim et, arrStartEdge , currentHalfedge , nextHalfedge , blnLast
  Set et= Rhino.GetPlugInObject("EvoluteTools for Rhino")

  ' Get the starting edge and its halfedge
  arrStartEdge  = et.PickEdge(1, 1, "Select the polyline")
  nextHalfedge  = et.EmHalfedge(arrStartEdge )

```

```
17  ' Move to the very end of the polyline
18  Do
19     '' Advance
20     currentHalfedge = nextHalfedge
21     '' Compute next halfedge
22     nextHalfedge = et.EmNextHalfedge( et.EmOppositeHalfedge( et.EmNextHalfedge (
       currentHalfedge )))
23  Loop While Not ( et.EmVertexValence (et.EmEndVertex( currentHalfedge )) <> 4 Or
    et.EmEdge(nextHalfedge )(1) = arrStartEdge(1))
24
25  ' Change movement direction
26  currentHalfedge = et.EmOppositeHalfedge(currentHalfedge )
27  arrStartEdge = et.EmEdge(currentHalfedge )
28  nextHalfedge = currentHalfedge
29  ReDim arrEdges(-1)
30
31  ' Move to the opposite end of the polyline, create line objects on the way
32  Do
33     '' Advance
34     currentHalfedge = nextHalfedge
35     '' Create Line
36     ReDim Preserve arrEdges(Ubound(arrEdges)+1)
37     Dim arrPoint1 : arrPoint1 = et.EmPoint( et.EmStartVertex( currentHalfedge ))
38     Dim arrPoint2 : arrPoint2 = et.EmPoint( et.EmEndVertex( currentHalfedge ))
39     arrEdges(Ubound(arrEdges)) = Rhino.AddLine( arrPoint1, arrPoint2)
40     '' Compute next halfedge
41     nextHalfedge = et.EmNextHalfedge( et.EmOppositeHalfedge( et.EmNextHalfedge (
       currentHalfedge )))
42  Loop While Not ( et.EmVertexValence( et.EmEndVertex( currentHalfedge )) <> 4 Or
    et.emEdge(nextHalfedge )(1) = arrStartEdge(1))
43
44  ' Join the lines into a polyline, move the polyline to "Polyline" layer
45  Dim crvFinalCurve : crvFinalCurve = Rhino.JoinCurves(arrEdges)
46  Call Rhino.DeleteObjects(arrEdges)
47
48  '' Create layer "Polyline" if one doesn't exist already
49  If Not Rhino.IsLayer("Polyline" ) Then
50     Rhino.AddLayer("Polyline" )
51  End If
52
53  Rhino.ObjectLayer crvFinalCurve , "Polyline"
54  Rhino.ObjectColor crvFinalCurve , RGB(255,0,0)
55 End Sub
```

The script asks the user to pick a mesh edge (line 14). It then finds one of its halfedges (line 15) and enters a loop in which it navigates from halfedge to halfedge either to the boundary of the mesh or until it circles back to the starting edge, if the mesh is closed at this point. We need to find the mesh boundary, if it exists, to have a start point from where we will create the polyline.

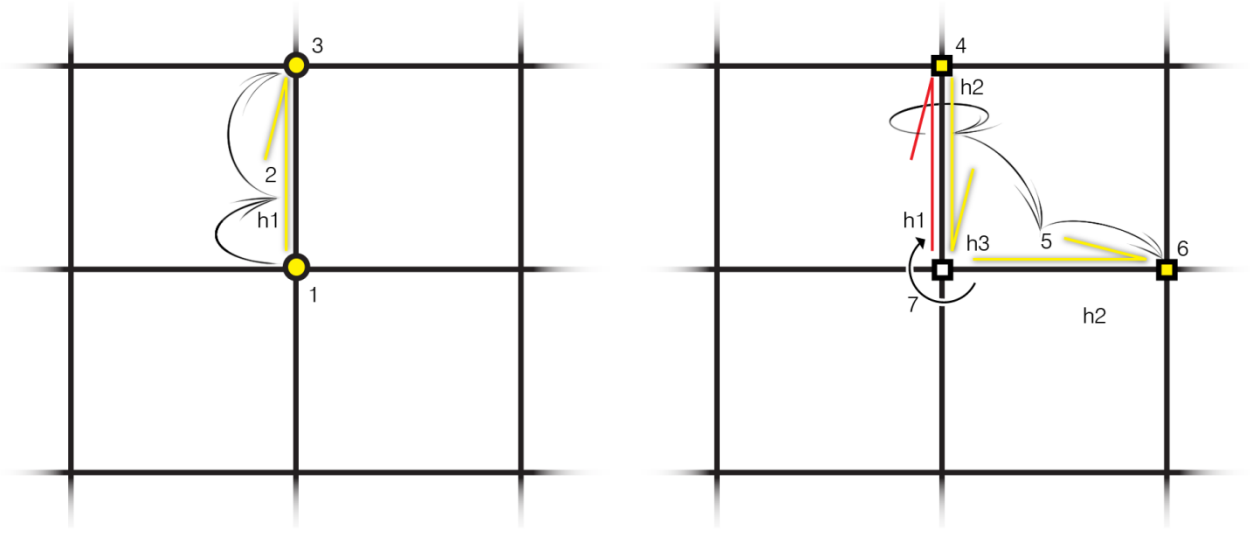
The first halfedge is stored as *currentHalfedge* (line 20). Now comes the tricky part for navigating to the next halfedge in the polyline. We have to move first to the next halfedge, then to the opposite halfedge and from there one more time to the next halfedge (line 22).[4] This step will be repeated until we hit the mesh boundary or circle back to our starting edge *arrStartEdge* (line 23). When we arrive there we can change the movement direction and walk back. We do this simply by returning the opposite halfedge of our *currentHalfedge* which is pointed in the opposite direction. Again we define a starting edge and the *nextHalfedge* variable (line 26-28). In the following loop we will move back along the polyline the same way we did before but now we will create a curve on every edge we visit.

[4] The line 22 is too long to fit on this sheet of paper in one piece so it has been divided into two lines. If you open the example script in your script editor you will notice that it is still in one piece there.

We find the start point and the end point of each edge (line 37-38), and create a curve between them (line 39). We need to store the generated curves in an array to be able to join them into a polyline later. The array was defined as an empty array on line 29 (*arrEdges* (-1) means the array is empty where *arrEdges* (0) would mean that the array contains one element with the index zero). In the beginning of every loop we redimension the array (line 36) to add one free space on its end which will store the next curve object. After all curves are created we join them into a polyline and delete the original curves (line 45-46). If there is no layer "Polyline" in the scene we create it (line 49-51). We put our polyline on that layer (line 53), and give it a red color to make it distinguishable in the viewport (line 54).

2.6.5 Vertex Ring

In the following example we will discuss a script which takes a vertex as input and finds the first ring of vertices around it. It creates Rhino point objects in the coordinates of those vertices. The halfedge data structure navigation part of this script is similar to the one in the 3rd example. You can regard the following example as a warm-up for our final exercise in this chapter. Here's what we need to do:



1. Get vertex
2. Get halfedge
3. Store halfedge's end vertex in an array
4. Create Rhino Point
5. Go to next outgoing halfedge
6. Store halfedge's end vertex in an array, add rhino point
7. Repeat this until you hit the first halfedge again

```
1 Option Explicit
2 'Script written by Marko Tomicic
3 'Script copyrighted by Evolute GmbH
4
5 Call VertexRing()
6 ' This script asks for one vertex as user input. It finds the first ring of vertices
7 ' around the selected vertex, stores them into the arrVertices array and creates
8 ' Rhino point objects in those vertices.
9 Sub VertexRing()
10   Dim et, arrStartVertex , arrStartPoint , arrPoints(), arrHalfedge , arrVertices(),
     arrStartEdge
11   Set et = Rhino.GetPlugInObject("EvoluteTools for Rhino")
12
13   ' Pick vertex, find its halfedge, store vertex handle and its point coordinates in
14   ' two seperate arrays.
15   arrStartVertex = et.PickVertex(1, 1, "select one vertex")
16   arrHalfedge = et.EmHalfedge(arrStartVertex)
17   ReDim arrVertices(0)
18   ReDim arrPoints(0)
19   arrVertices(0) = arrStartVertex
20   arrStartPoint = et.EmPoint(arrStartVertex)
21   arrPoints(0) = arrStartPoint
22   arrStartEdge = et.EmEdge(arrHalfedge)
23
24   ' Loop through all outgoing halfedges of the vertex and add their end vertices
25   ' to the array. Add vertex coordinates to second array.
26   Do
27      Dim intVerticesInArray  : intVerticesInArray  = Ubound(arrVertices)
28      Dim intPointsInArray  : intPointsInArray  = Ubound(arrPoints)
29      ReDim Preserve arrVertices(intVerticesInArray  + 1)
```

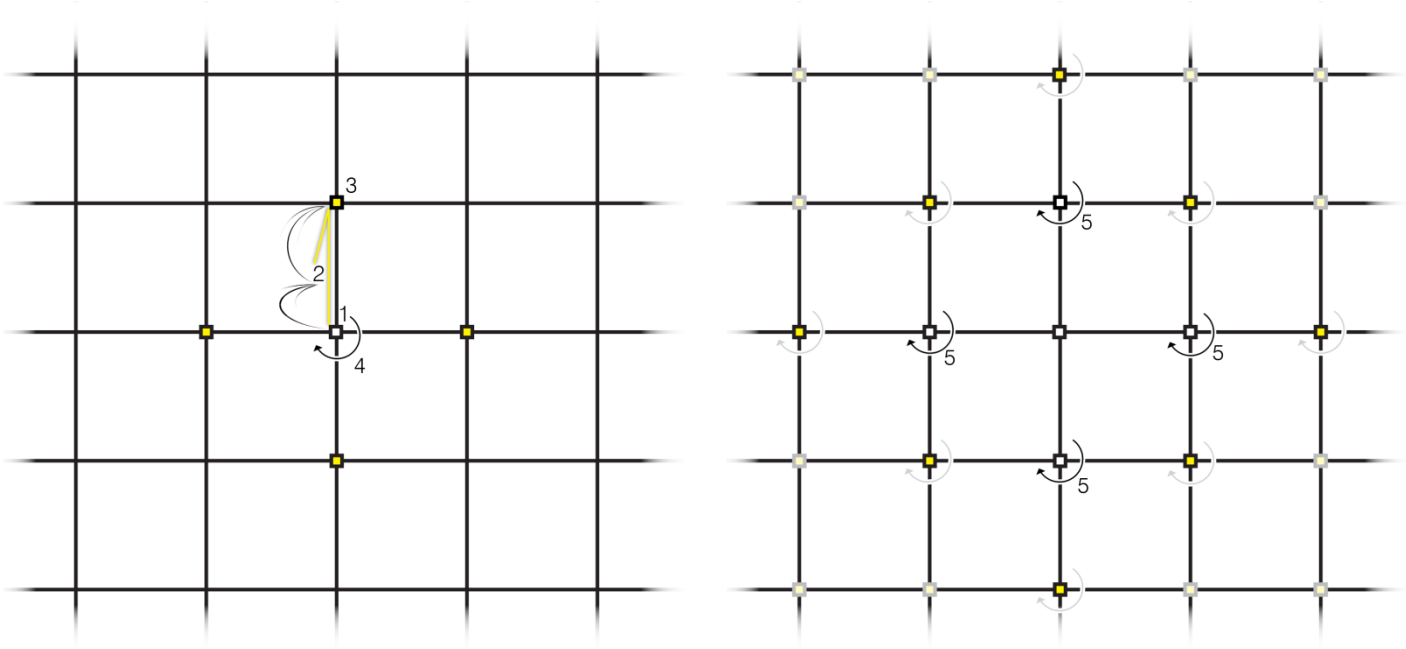
```
30      ReDim Preserve arrPoints(intPointsInArray  + 1)
31      arrVertices(intVerticesInArray  + 1) = et.EmEndVertex(arrHalfedge)
32      arrPoints(intPointsInArray  + 1) = et.EmPoint( arrVertices( intVerticesInArray  + 1))
33      arrHalfedge  = et.EmNextHalfedge( et.EmOppositeHalfedge(arrHalfedge))
34    Loop While et.emEdge(arrHalfedge )(1) <> arrStartEdge(1)
35
36    ' Create point objects
37    Call Rhino.AddPoints(arrPoints)
38  End Sub
```

We start by asking the user to select one mesh vertex from his viewport (line 15). We then find one outgoing halfedge of that vertex (line 16). The script will randomly return one of the possible halfedges. This is not important because we know that the halfedge is outgoing, thus we will loop through all outgoing halfedges starting from there and select their end vertices. The arrays for storing our vertices (*arrVertices*) and their point coordinates (*arrPoints*) are redefined. They now hold each one element so we can store the vertex that the user selected (line 17-18). We will later add all other vertices and their point coordinates to those arrays. First we will remember our first halfedge's edge as *arrStartEdge* (line 22). This way, when we find ourselves again at this edge, we will know that we have completed a full circle through the outgoing halfedges of the selected vertex.

We enter now a loop through all the outgoing halfedges of our selected vertex. The arrays *arrVertices* and *arrPoints* are redefined to feature an empty space in which the next vertex/vertex coordinate will be stored (line 27-30). The end vertex of the current halfedge is stored at the last place (the empty place that we created on line 29) in the arrVertices array (line 31). The respective point coordinates of that vertex are stored in the arrPoints array (line 32). We advance through the halfedge data structure to the next outgoing halfedge by going **first** to the opposite halfedge of our current halfedge and **from there** to its next halfedge. This new halfedge will enter the next loop as *arrHalfedge* and its end vertex will be stored in the array. The loop repeats until it arrives at *arrStartEdge* (line 34). After all vertices are stored in the *arrVertices* array and their respective point coordinates in the *arrPoints* array, we use the latter array to create Rhino point objects in those coordinates (line 37).

2.6.6 Vertex nRing

This will be the last exercise in this chapter. Again, we will create a script which will return the nRing[5] of a selected vertex and add Rhino points to the returned vertices. This time we will spice it up a little by letting the user decide how many rings the script should return. This will implicate some deeper nesting of loops which looks complicated at first, but isn't at all once you take your time to analyze the script. I'll be using the „*MeshNRing*" function to return the array of vertices in this script. You can copy the function to your own scripts and just call it like it is called here on line 19.



1. Get vertex
2. Get halfedge
3. Create point in end vertex
4. Repeat this for all outgoing halfedges
5. Do this once in every newly created point (repeat N times)

First we ask the user to pick one vertex and to tell the script how many rings of vertices should be selected (line 16-17). The number of rings "N" will be used in the function for counting purposes in the first loop. The *MeshNRing* function is called on line 19. It takes the starting vertex which the user has selected and the number of rings he/she wants to select as input. The output of the function is the *arrVertices* array which stores all vertex handles belonging to the nRing. The function will be discussed separately further below. After the function has returned the vertices, we will need to know the number of vertices/elements in the *arrVertices* array. It is found on line 21 and stored in the *intNoOfVertices* variable. This number will be used to limit our next loop to the same number of repetitions. The loop (line 22-25) goes through all vertices in the array one by one and finds the point coordinates of each vertex (line 23). It then creates Rhino point objects in those coordinates (line 24). On the following page is the complete script with the *MeshNRing* function. The function will be discussed on the next page.

[5] The nRing is the ring of vertices which can be reached with "n" steps starting from the starting vertex.

```
Option Explicit
'Script written by Marko Tomicic
'Script copyrighted by Evolute GmbH

Call UniversalVertexNRing()
' This script creates points in all vertices belonging to the nRing*
' around the selected vertex.
' The number of rings is user specified
' *The NRing is the ring of vertices which can be reached with N steps
' from the starting vertex
Sub UniversalVertexNRing()

  Dim et, arrStartVertex , N, arrVertices , intNoOfVertices , i
  Set et = Rhino.GetPlugInObject("EvoluteTools for Rhino")
  ' Get Vertex and number of rings
  arrStartVertex = et.PickVertex(1, 1, "select one vertex")
  N = Rhino.GetInteger("number of rings around the vertex")
  ' Call function
  arrVertices = MeshNRig(arrStartVertex , N)
  ' Add Rhino point objects to the vertices
  intNoOfVertices = Ubound(arrVertices )
  For i=0 To intNoOfVertices
     Dim arrPoint : arrPoint = et.EmPoint(arrVertices(i))
     Call Rhino.AddPoint(arrPoint)
  Next
End Sub

Function MeshNRing(ByVal Vertex, ByVal N)
  ' This function takes a unique vertex handle and a integer "N" as input.
  ' It finds all the vertices in the NRing of the selected vertex.
  Dim et, arrHalfedge , arrVertices(), arrStartEdge , i, j, l
  Set et = Rhino.GetPlugInObject("EvoluteTools for Rhino")
  ' Add the selected vertex as first element to the array
  ReDim arrVertices(0)
  arrVertices(0) = vertex

  arrHalfedge = et.EmHalfedge(Vertex)
  arrStartEdge = et.EmEdge(arrHalfedge )
  Dim intPrevNoInArray : intPrevNoInArray = Ubound(arrVertices )
  Dim intNewNoInArray : intNewNoInArray = 1

  ' N Loops for N rings of vertices
  For i=1 To N
     Dim k : k = intNewNoInArray - intPrevNoInArray
     intPrevNoInArray = Ubound(arrVertices )
     Dim intPrevNoInArrayStatic : intPrevNoInArrayStatic = intPrevNoInArray
     ' Loop through the outgoing halfedges of every vertex in the ring and
     'add end vertex to the array if it isn't already there.
     For j=0 To k-1
        arrHalfedge = et.EmHalfedge( arrVertices( intPrevNoInArrayStatic - j))
        arrStartEdge = et.EmEdge(arrHalfedge )
        Do
           Dim intPrevNoInArray2 : intPrevNoInArray2 = Ubound(arrVertices )
           Dim arrVertex : arrVertex = et.EmEndVertex(arrHalfedge )
           Dim isFound
           isFound = False

           '' Search the array of vertices for the current vertex "arrVertex"
           For l=0 To Ubound(arrVertices )-1
              Dim arrVerticesIndex : arrVerticesIndex = et.EmIndex(arrVertices(l))
              Dim arrVertexIndex : arrVertexIndex = et.EmIndex(arrVertex)
              If arrVerticesIndex = arrVertexIndex Then
                 isFound = True
              End If
           Next
           '' If the current vertex doesn't exist in the array then add it
           If Not isFound Then
              ReDim Preserve arrVertices(intPrevNoInArray2 + 1)
              arrVertices(intPrevNoInArray2 + 1) = et.EmEndVertex(arrHalfedge)
           End If
           '' Advance
           arrHalfedge = et.EmNextHalfedge( et.EmOppositeHalfedge(arrHalfedge))
        Loop While et.emEdge(arrHalfedge)(1) <> arrStartEdge(1)
     Next

     intNewNoInArray = Ubound(arrVertices)
  Next

  intPrevNoInArray = Ubound(arrVertices)
  MeshNRing = arrVertices
End Function
```

The function:

The function starts similar to the main subroutine. The first few lines are used to declare some variables and to call the EvoluteTools scripting interface object.

Before we continue our way through the function line by line I will first explain how this function is built up. You should not have a panic attack if you don't understand this one right away, because it is a little complicated. The function features four nested loops.

The first loop (line 43-77) repeats once for every nRing of vertices which the user wants to have selected. Remember that we asked the user for the number of *nRings* on line 17 in the regular part of the script. We used then that number as input for the *MeshNRing* function. That number determines now how many times the first loop will be repeated.

The second loop (line 49-74) loops through all new vertices in *arrVertices*. The first time it runs just once for the selected vertex, but the second time it will repeat "n" times for "n" newly added vertices to the arrVertices array.

The third loop (line 52-73) repeats once for every outgoing halfedge of the current vertex *(arrHalfedge)* which is fed to it by the previous loop. It finds the end vertex *(arrVertex)* of each outgoing halfedge and compares it to the existing vertices in the *arrVertices* array. If it is not already in the arrVertices array it will be added by this third loop. The loop continues as long as the current edge is different from arrStartEdge which was defined on line 51.

The fourth loop (line 59-65) actually does the comparing of the current vertex, provided by the third loop, and the list of vertices. It tells the third loop whether to add its current vertex to the list or just to move on to the next one.

I will try to clarify it further by going line by line through the function. The user selected vertex is added to the *arrVertices* array on first place (line 35). We find one outgoing halfedge of that vertex and store its edge as *arrStartEdge*. The *intPrevNoInArray* variable is set to the current number of elements in the *arrVertices* array and the *intNewNoInArray* is set to "1" for now (line 39-40). We enter the first loop on line 43 and define the value "k" (line 44). This value tells us how many new vertices are added to the arrVertices array and thus it tells the second loop how many times to repeat (once for every new vertex). After that we have to update the value stored in *intPrevNoInArray* (line 45). This value is also stored in the *intPrevNoInArrayStatic* variable (line 46). The second loop is started on line 49. Here we extract the halfedge (arrHalfedge) of the newly added vertices from the *arrVertices* array one at a time and set its edge to be our *arrStartEdge* (line 50-51). The third loop starts on line 52. In this loop we will find the neighboring vertices of the current vertex. First we will check how many vertices there are in the *arrVertices* array and store that information in *intPrevNoInArray2* (line 53). On line 54 we find the end vertex *(arrVertex)* of *arrHalfedge* - *arrHalfedge* is provided by the previous loop (see line 50 for *arrHalfedge*). The variable *isFound* is defined to be `False` (line 55-56).

The last loop starts on line 59. This loop compares the current vertex to all existing vertices in the *arrVertices* array. For this we need to get the indices of all vertices and compare them (line 60-61). If the index of the current vertex matches with the index of one of the existing vertices in the array, then this is a sign that the current vertex has been visited before and already is stored in the *arrVertices* array. In that case the *isFound* variable is changed to `True` (line 62-64). If no match is

found the *isFound* variable stays `False` . The fourth loop ends on line 65, which brings us back to the third loop. We know now whether the vertex that we defined on line 54 already exist in *arrVertices* or not. If not then we add an empty space at the end of the arrVertices array and add the vertex to that array (line 67-70). We advance to the next outgoing halfedge of the current vertex and loop until we hit the edge from where we started (line 72-73). The third loop is exited on line 73 and the second loop on line 74. After the 2nd, 3rd and 4th loop is exited we prepare the *intPrevNoInArray* variable for a new round in loop number one (line 76). The first loop is exited on line 77. After all our loops are completed the *arrVertices* array should hopefully hold all the vertices we need. We have to define the function output to be that array of vertices (line 80).

The function ends on line 81 and this might be just the right time for you to have a break and enjoy a cold beverage.

3. Mesh Editing

3.1 About this chapter

The EvoluteTools for Rhino workflow consists of 4 basic steps plus one fifth step which will be introduced in the last chapter of this Primer:

1. Create the reference surface
2. Create the coarse mesh
3. Subdivision
4. Optimization
5. Post processing

The **reference geometry** is the geometrical shape which we are trying to match as close as possible with our mesh. The reference can be any Rhino surface or mesh or even justone or more curves in space. It can be created using the broad spectrum of Rhino tools or it can be imported from other CAD systems. It makes no difference so I'll let this topic go.

The mesh creation process starts with a manual creation of the **coarse mesh**. Depending on the complexity of the reference geometry the coarse mesh can be very simple or fairly complicated thus the creation process of the coarse mesh can become a little cumbersome sometimes. This is however one of the most creative parts of the workflow in EvoluteTools for Rhino. The user creates a base object from which the final mesh and thus the built structure will evolve. I would suggest that you take your time when creating the coarse mesh and try to do it right the first time. It is of course no tragedy if you find out later that you could achieve better results in the optimization if you had a different coarse mesh. You can always go back to the step where you create the coarse mesh and redo it, but doing it right the first time will probably save you some time in the future. The evolution from coarse- to the final mesh is an iterative process of **subdivision** and **optimization** in which the mesh is refined and push/pulled into shape[6].

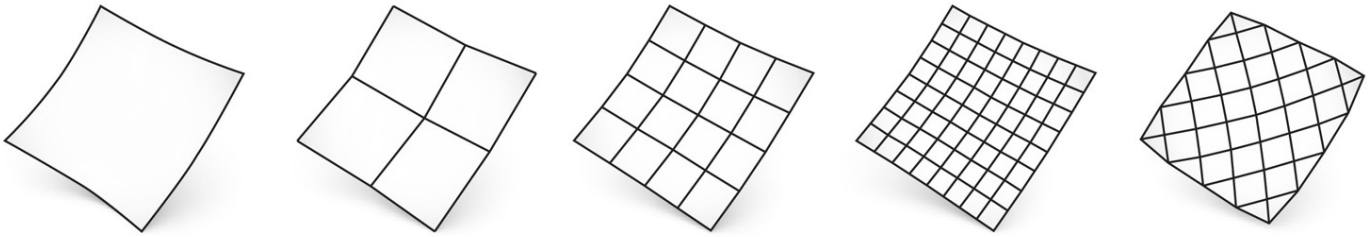
Create the reference surface
Create the coarse mesh
Subdivision
Optimization
Post processing

In this chapter we will concentrate on the subdivision step of the workflow.

[6] For further information on the topics of reference geometry, coarse mesh, subdivision and optimization see Kris Hammerberg's EvoluteTools Primer.

3.2 Divide et Impera

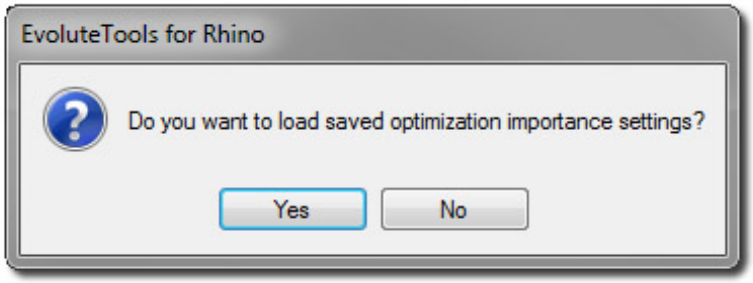
EvoluteTools for Rhino uses a Top Down approach to achieve the optimal mesh for a given reference geometry. It starts with an input geometry and a coarse mesh which is then refined and subdivided via an appropriate procedure[7]. The goal of the mesh subdivision is to achieve a certain layout of the mesh edges and a desired initial density of the mesh. The density of a mesh is measured in the number of faces or edge lengths.



'It is useful to think about refinement as a two-step procedure: change the connectivity (number of vertices and the way they are connected) and then change the geometry (the position of the vertices).'[8] This means that after each subdivision step the vertices change their position. A planar quad face might not subdivide into four smaller planar faces. In some cases it is necessary to make smaller or bigger manual changes to the mesh before or after optimization (the processes of subdivision and optimization are often alternating). The smaller changes like deleting a few edges or adding some loop cuts can be easily done with the excellent EvoluteTools commands in Rhino. There are however cases when those changes can become really time consuming, e.g., if you work on a large scale project featuring a highly complex mesh.

3.3 Examples

On the following pages I will go through some example scripts for mesh editing and explain them step by step. The example files are available as download together with this Primer. Each example file, if there is one for the particular exercise, is named according to the name of the section in this Primer in which it is displayed. The files have been set up so that you can run the script and see the outcome without having to adjust any settings. You should always choose to load the optimization options settings when prompted while opening an example file.



The *Optimization Options Importance* settings are crucial for the successful completion of some of the scripts, so be careful when you experiment with those options in the example files.

[7] See the EvoluteTools Primer
[8] Pottman, H. & Asperl A. & Hofer, M. & Kilian, A., *Architectural Geometry,* Bentley Institute Press, 2007, p 390.

3.3.1 Edge Length Analysis

We will start this chapter's examples with a really simple one. We will write a script which will return the length of the shortest and the longest edge in a mesh. This is something which will be useful in our other scripts. The code will be encapsulated into a Function which we can call later in other scripts. Below is the complete script with the *ExtremeEdges* function. It returns an array holding the shortest and the longest edge's length. In addition to that the script returns the handles of the shortest and longest mesh edge as *ShortestEdge* and *LongestEdge* variables. Those variables have to be provided as function input.

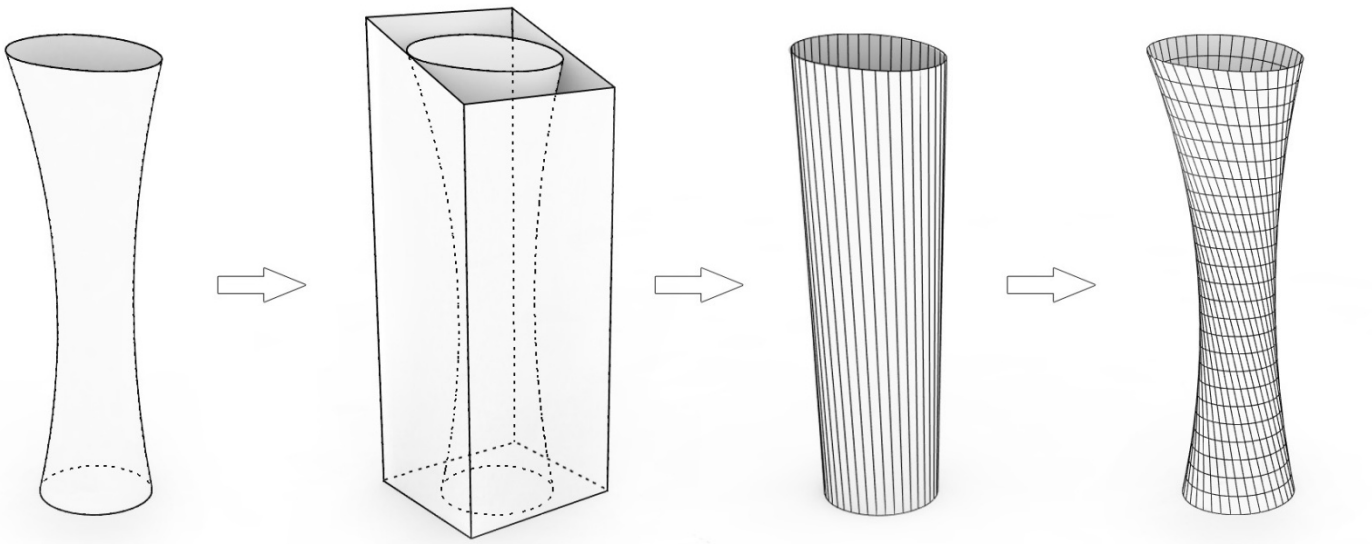
```
Option Explicit
'Script written by Marko Tomicic
'Script copyrighted by Evolute GmbH

Call EdgeLength()
' This script finds the shortest and the longest edge of a mesh
Sub EdgeLength ()
    ' Declaration of variables
   Dim et, objMesh, EdgeLengths , dblShortestEdgeLength , dblLongestEdgeLength ,
   ShortestEdge , LongestEdge
    ' Get EvoluteTools plugin object
    Set et = Rhino.GetPlugInObject ("EvoluteTools for Rhino")
    If IsNull(et) Then Exit Sub

    'Get user input
    objMesh = Rhino.GetObject("Select the mesh", 32)

    ' Call the function to return the longest and shortest edge values
    EdgeLengths = ExtremeEdges (objMesh, ShortestEdge , LongestEdge )
    dblShortestEdgeLength = EdgeLengths (0)
    dblLongestEdgeLength = EdgeLengths (1)

    ' Print the results to the command line
    Call Rhino.Print("Longest edge's length: " & CStr(dblLongestEdgeLength ))
    Call Rhino.Print("Shortest edge's length: " & CStr(dblShortestEdgeLength ))
End Sub


Function ExtremeEdges (ByVal TheMesh, ByRef ShortestEdge , ByRef LongestEdge )
    ' This function returns the handle of the shortest and the longest mesh edge.
    ' The lengths of the longest and the shortest mesh edge are stored in an array.
    ' The shortest edge's length is stored at position 0.
    ' The longest edge's length is stored at position 1.

    ' Declaration of variables
   Dim et, intNoOfEdges , i, currentEdge , dblEdgeLength , arrEdgePoints , EdgesLengths (3)
    Dim dblSmallestLength , dblLargestLength
    ' Get EvoluteTools plugin object
    Set et = Rhino.GetPlugInObject ("EvoluteTools for Rhino")
    If IsNull(et) Then Exit Function

    dblSmallestLength = 99999999
    dblLargestLength = 0
    intNoOfEdges = et.EmNumberOfEdges (TheMesh)


    ' Loop through all edges off the mesh, find the longest
    For i=0 To intNoOfEdges - 1
        currentEdge = et.EmEdge(TheMesh, i)
        '' Extract the endpoints of the edge,
        '' measure the distance between them
        arrEdgePoints = et.EmGetEdgePoints (currentEdge )
        dblEdgeLength = Rhino.Distance(arrEdgePoints (0), arrEdgePoints (1))
        If dblEdgeLength < dblSmallestLength Then
            dblSmallestLength = dblEdgeLength
            ShortestEdge = currentEdge
        End If
        If dblEdgeLength > dblLargestLength Then
            dblLargestLength = dblEdgeLength
```

```
59              LongestEdge  = currentEdge
60          End If
61      Next
62
63      ' Return length of the shortest and the longest mesh edge
64      EdgesLengths (0) = dblSmallestLength
65      EdgesLengths (1) = dblLargestLength
66
67      ' Function output
68      ExtremeEdges  = EdgesLengths
69
70  End Function
```

We start by declaring some variables and loading the EvoluteTools scripting interface object (19-12). After the user is asked to select a mesh in the Rhino viewport it is time to call the function (line 18). The function takes three variables as inputs: the mesh, the shortest edge and the longest edge. In this moment only the mesh needs to be a valid variable. The *ShortestEdge* and *LongestEdge* variables will be assigned in the function itself, thus right now they should be empty. After calling the function the script jumps from line 18 to line 28. Again the variables for the function are defined and the EvoluteTools scripting interface object is set (line 35-39). In the function we will loop through all mesh edges and measure their length. The length will then be compared to the currently longest and shortest edge. If the current edge is longer than the longest edge so far, or is shorter than the shortest edge so far it will replace it. We need some numerical values to compare with the first measured edge. Those values are *dblSmallestLength* and *dblLargestLength* (line 41-42). They are set to "0" and "99999999" to make sure that the length of the first measured edge will replace both values. In order to initiate a loop through all mesh edges we need to know how many there are (line 43). The loop starts on line 47. The edge with the index "i" is picked from the mesh. We extract the two end points of that edge and measure the distance between them in order to find the edge's length (line 48-52). The edge's length is compared to the current smallest and largest edges. As mentioned above, if the current edge is shorter than the shortest edge so far it will replace it as *ShortestEdge* and its length will be saved to the *dblSmallestLength* variable (line 53-55). The same rule is applicable to the case when the current edge is longer as the longest edge so far (line 57-59). The lengths of the shortest and the longest edge are stored to the *EdgesLengths* array which is also the output of this function. The handles of these two edges will be returned to the main subroutine via the *ShortestEdge* and *LongestEdge* variables. Back in the main subroutine we extract the two lengths from the function output to their respective variables (line 19-20) and print them to the command line (line 23-24).

3.3.2 Edge Length Subdivision

In the following example we are going to generate a mesh layout for a tower-like shape. We start with a really simple coarse mesh (a box without top and bottom) and subdivide it into a very good initial mesh for optimization. Instead of just subdividing the coarse mesh with the standard procedures and hoping for the best, we will take the wheel and control the subdivision process with this script. The script uses a target edge length as input and subdivides the edges of the coarse mesh until they meet the target. Note that it is necessary to run the script in two steps to go from coarse to refined mesh. In the first run you should select one horizontal edge, top or bottom, of the mesh. The script will then generate the vertical edges of the mesh. After the script completes you should run it one more time and select one of the vertical edges when asked to get the final shape as shown below.



```
Option Explicit
'Script written by Marko Tomicic
'Script copyrighted by Evolute GmbH

Call EdgeLengthSubdivision ()
' This script subdivides the mesh by loop cutting a selected edge until the edge's length
' is shorter than a specified value. This script works with quadrilateral meshes.
Sub EdgeLengthSubdivision ()

    Dim et, objMesh, dblMaxEdgeLength , currentEdge , arrEndpoints
    Dim dblCurrentEdgeLength , arrResult
    Set et = Rhino.GetPlugInObject ("EvoluteTools for Rhino")
    ' Get user input
    objMesh = Rhino.GetObject("Select the mesh", 32)

    If IsNull(objmesh) Then
        Rhino.Print "Mesh could not be selected!"
        Exit Sub
    End If

    currentEdge = et.PickEdge(1,1, "Select the edge")
    dblMaxEdgeLength = Rhino.GetReal("Maximum edge length? ", 1.5)

    ' Add loop cuts to the edge until it becomes shorter than dblMaxEdgeLength
    Do
        '' Measure the length of the edge
        arrEndpoints = et.EmGetEdgePoints (currentEdge )
        dblCurrentEdgeLength = Rhino.Distance(arrEndpoints (0), arrEndpoints (1))
        '' If the edge is to long add a loop cut
        If dblCurrentEdgeLength > dblMaxEdgeLength Then
            Call et.MeshLoopCut (currentEdge )
```

```
32              Call et.Optimize(objMesh,  arrResult)
33          End If
34
35      Loop While dblCurrentEdgeLength > dblMaxEdgeLength
36  End Sub
```

As you can see this is a very powerful little piece of code. It starts with the declaration of some variables and the mesh as user input. You can always declare all the variables in only one line. I used two lines just to fit the text into this document.

If an error occurs during the selection of the mesh the script will abort and write an error message: `"Mesh could not be selected!"` (line 16-19). You can use the *IsNull(anyVariable)* method after you define any variable to check if everything went according to plan. This can be especially helpful when debugging a script.

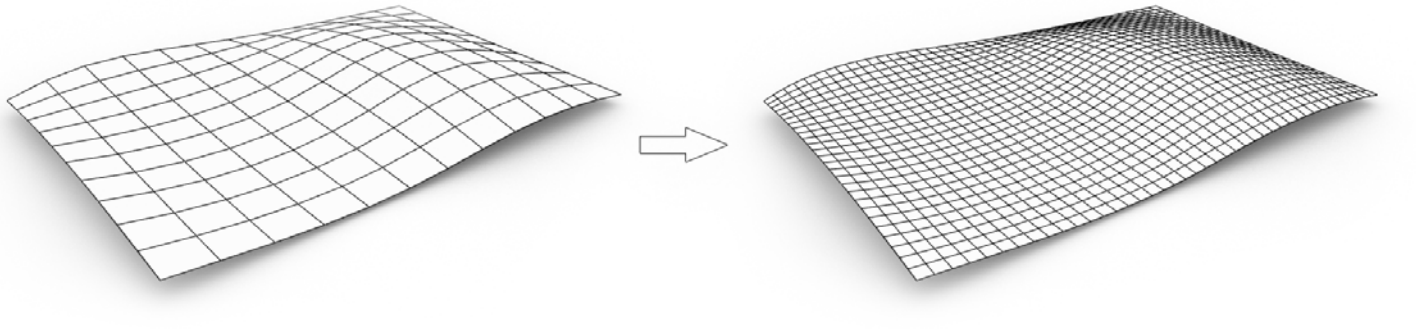
After we made sure that the *objMesh* variable contains the mesh we ask the user to select one edge to subdivide and to specify the maximal allowed edge length (line 21-22).

The loop (line 25-35) measures the length of the selected edge and applies a loop cut to it if it is longer than the maximal allowed edge length. After every loop cut (line 31) the mesh has to be optimized (line 32) in order to get evenly distributed polylines on the mesh[9]. The *et.Optimize* function takes the mesh as first input variable and the *arrResult* variable as second input. The *arrResult* variable is used by EvoluteTools to store some background data about the optimization. You do not have to worry about this. The only important thing is to define the *arrResult* variable somewhere in the script (I like to do that right at the beginning of the script) and to plug it into the *et.Optimize* function. The loop will continue until the selected edge becomes shorter than the specified value.

[9] A fair amount of *FairnessCurvature* is needed to distribute the polylines evenly on the mesh. All example files are pre-set with optimization options which are tested and work good in the particular case. You are however welcome to experiment and change those options as you like.

3.3.3 Max Edge Length Loop Cutting

The following example shows a script which densifies the mesh by adding loop cuts to certain edges. The user specifies a maximal possible edge length. The script will find the longest edge in the mesh, measure its length and if it is longer than it should be it adds a loop cut in the middle of that edge. After every loop cut the mesh is relaxed to distribute the polylines evenly. This process of loop cutting and relaxing repeats until all edges are shorter than the specified maximal length.



This approach will be helpful when you want to create an initial density in a mesh before you proceed to optimization. It is also helpful when a mesh is prepared for production. When it comes to manufacturing the mesh, there are always some limitations. An obvious case is the panel size which can be produced using a particular manufacturing technique. With this script you can limit the maximum edge length so it can be manufactured with the machines you are working with. After this example you should be able to write your own scripts which limit the area or the diagonal length of a face/panel.

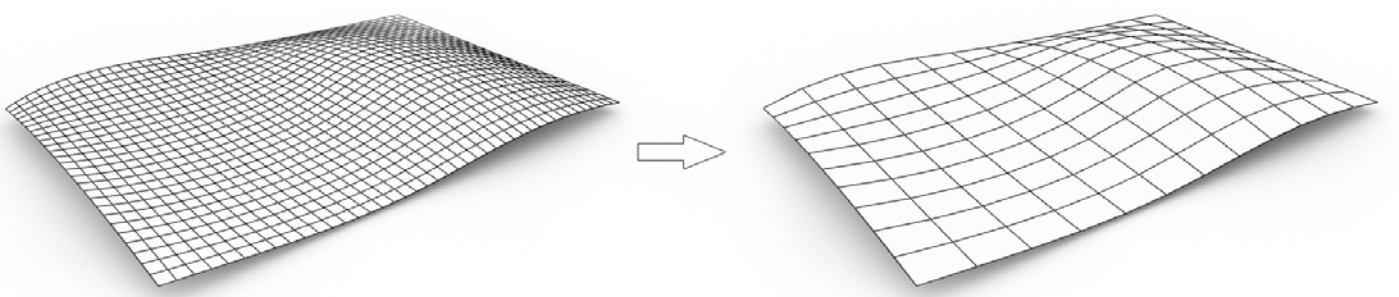
```
Option Explicit
'Script written by Marko Tomicic
'Script copyrighted by Evolute GmbH

Call MaxEdgeLengthLoopCutting ()
' This script densifies the mesh until all edges are shorter than a
' specified maximum length. This script works with quadrilateral meshes.
Sub MaxEdgeLengthLoopCutting ()

    ' Declaration of variables
    Dim et, objMesh, intNoOfEdges , dblMaxEdgeLength , dblLongestEdgeLength
    Dim i, arrResult, dblLongestEdgeAfter , LongestEdge , dblLongestEdgeBefore
    ' Get EvoluteTools plugin object
    Set et = Rhino.GetPlugInObject ("EvoluteTools for Rhino")
    If IsNull(et) Then Exit Sub
    ' Get inputs
    objMesh = Rhino.GetObject("Select the mesh", 32)
    intNoOfEdges = et.EmNumberOfEdges (objmesh)
    dblMaxEdgeLength = Rhino.GetReal("Maximum edge length? ", 2)
    ' Find the longest edge in the mesh
    dblLongestEdgeBefore = ExtremeEdges (objMesh,, LongestEdge )(1)
    ' Find longest mesh edge. If it is longer than the maximum length
    ' perform a loop cut.
    Do
        dblLongestEdgeLength = ExtremeEdges (objMesh,, LongestEdge )(1)
        If dblLongestEdgeLength > dblMaxEdgeLength Then
            Call et.MeshLoopCut (LongestEdge )
            Call et.Optimize(objMesh, arrResult )
        Else
            Exit Do
        End If
    Loop
    ' Optimize!!!
    Call et.Optimize(objMesh, arrResult )

```

```
36      ' Find the longest edge in the mesh
37      dblLongestEdgeAfter  = ExtremeEdges (objMesh,, LongestEdge )(1)
38
39      Call Rhino.Print("Longest edge's length before: " & CStr(dblLongestEdgeBefore ))
40      Call Rhino.Print("Longest edge's length after: " & CStr(dblLongestEdgeAfter ))
41  End Sub
42
43
44  Function ExtremeEdges (ByVal TheMesh, ByRef ShortestEdge , ByRef LongestEdge )
45      ' This function returns the handle of the shortest and the longest mesh edge.
46      ' The lengths of the longest and the shortest mesh edge are stored in an array.
47      ' The shortest edge's length is stored at position 0.
48      ' The longest edge's lenght is stored at position 1.
49
50      ' Declaration of variables
51      Dim et, intNoOfEdges , i, currentEdge , dblEdgeLength , arrEdgePoints , EdgesLengths (3)
52      Dim dblSmallestLength , dblLargestLength
53      ' Get EvoluteTools plugin object
54      Set et = Rhino.GetPlugInObject ("EvoluteTools for Rhino")
55      If IsNull(et) Then Exit Function
56
57      dblSmallestLength  = 99999999
58      dblLargestLength  = 0
59      intNoOfEdges  = et.EmNumberOfEdges (TheMesh)
60
61      ' Loop through all edges off the mesh, find the longest
62      For i=0 To intNoOfEdges  - 1
63          currentEdge  = et.EmEdge(TheMesh, i)
64          '' Extract the endpoints of the edge,
65          '' measure the distance between them
66          arrEdgePoints  = et.EmGetEdgePoints (currentEdge )
67          dblEdgeLength  = Rhino.Distance(arrEdgePoints (0), arrEdgePoints (1))
68          If dblEdgeLength  < dblSmallestLength  Then
69              dblSmallestLength  = dblEdgeLength
70              ShortestEdge  = currentEdge
71          End If
72          If dblEdgeLength  > dblLargestLength  Then
73              dblLargestLength  = dblEdgeLength
74              LongestEdge  = currentEdge
75          End If
76      Next
77
78      ' Return length of the shortest and the longest mesh edge
79      EdgesLengths (0) = dblSmallestLength
80      EdgesLengths (1) = dblLargestLength
81
82      ' Function output
83      ExtremeEdges  = EdgesLengths
84
85  End Function
```

After the variables are defined and the user is asked for the input information we run the *ExtremeEdges* function from the first example of this chapter (line 21). The function returns the length of the longest mesh edge before it makes any changes to the mesh. This information will be shown in the command line after the script is completed. On line 22 a new loop is initiated. The length of the longest mesh edge is compared to the user specified maximum edge length (line 26). If the length of the current edge is longer than allowed the script performs a loop cut through that edge and optimizes the mesh (line 27-28). The optimization step is necessary to relax the mesh and distribute its polylines evenly. This process of measuring and loop cutting is repeated until the *LongestEdge's* length becomes shorter than the value specified in the *dblMaxEdgeLength* variable. After all edges are subdivided we run the *ExtremeEdges* function again to make sure that the script worked. The longest edge values before and after running the script are displayed in the command line for visual reference (line 39-40).

3.3.4 Min Edge Length Delete Polylines

The script in this example does exactly the opposite of the previous. While the previous script added density to a mesh by performing loop cuts on selected edges the following script reduces the density in a mesh by deleting some polylines. This technique can be used when you are working on a mesh which consists of panels which are irrationally small for production. The script lets you specify a minimum edge length. It will then delete polylines from the mesh until there are no edges shorter than the specified value.



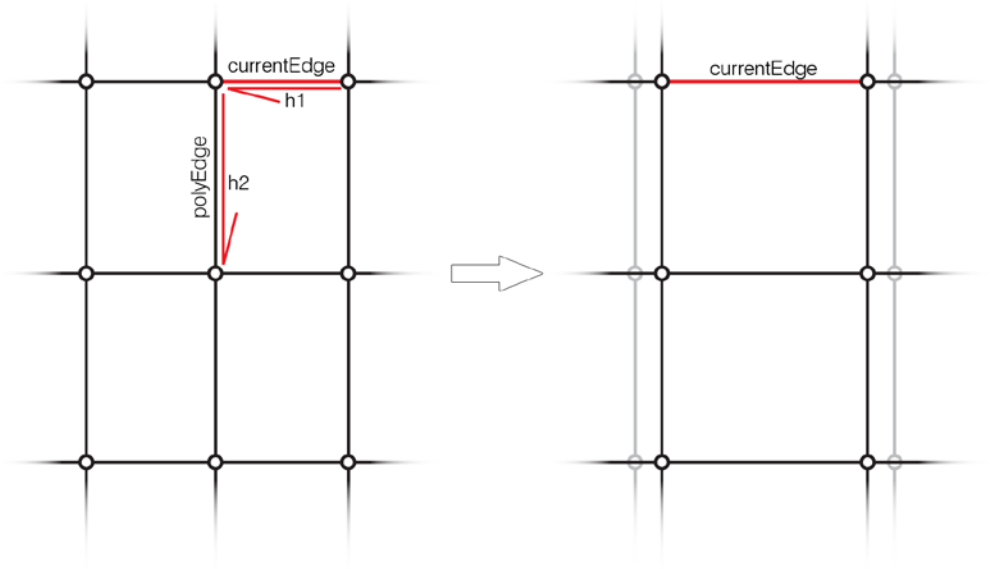
```
Option Explicit
'Script written by Marko Tomicic
'Script copyrighted by Evolute GmbH

Call MinEdgeLengthDeletePolylines ()
' This script makes the mesh less dense by deleting its polylines until all edges are
' longer than a specified minimum length. This script works with quadrilateral meshes.
Sub MinEdgeLengthDeletePolylines ()
    On Error Resume Next

    ' Declaration of variables
    Dim et, objMesh, intNoOfEdges , dblMinEdgeLength , currentHalfedge
    Dim dblShortestEdgeLength , polyEdge, arrResult, ShortestEdge , LongestEdge
    Dim dblShortestEdgeBefore , dblShortestEdgeAfter
    ' Get EvoluteTools plugin object
    Set et = Rhino.GetPlugInObject ("EvoluteTools for Rhino")
    If IsNull(et) Then Exit Sub

    ' Get inputs
    objMesh = Rhino.GetObject("Select the mesh", 32)
    intNoOfEdges = et.EmNumberOfEdges (objmesh)
    dblMinEdgeLength = Rhino.GetReal("Minimal allowed edge length? ", 3)

    ' Find the Shortest edge in the mesh
    dblShortestEdgeBefore = ExtremeEdges (objMesh, ShortestEdge , LongestEdge )(0)
    Do
        dblShortestEdgeLength = ExtremeEdges (objMesh, ShortestEdge , LongestEdge )(0)
        If dblShortestEdgeLength < dblMinEdgeLength Then
            currentHalfedge = et.EmHalfedge (ShortestEdge )
            polyEdge = et.EmEdge(et.EmNextHalfedge (currentHalfedge ))

            '' If the edge is a boundary find another one to delete
            If et.EmIsBoundary (polyEdge) Then
                polyEdge = et.EmEdge(et.EmPrevHalfedge (currentHalfedge ))
            End If
            '' Delete the polyline connected to polyEdge
            Call et.MeshDeletePolyline (polyEdge)
            Call et.Optimize(objMesh, arrResult)
        Else
            Exit Do
        End If
    Loop
    ' Optimize!!!
    Call et.Optimize(objMesh, arrResult)
```

```
45
46      ' Find the Shortest edge in the mesh
47      dblShortestEdgeAfter  = ExtremeEdges (objMesh, ShortestEdge , LongestEdge )(0)
48
49      Call Rhino.Print("Shortest edge's length before: " & CStr(dblShortestEdgeBefore ))
50      Call Rhino.Print("Shortest edge's length after: " & CStr(dblShortestEdgeAfter ))
51
52  End Sub
53
54
55  Function ExtremeEdges (ByVal TheMesh, ByRef ShortestEdge , ByRef LongestEdge )
56      ' This function returns the handle of the shortest and the longest mesh edge.
57      ' The lengths of the longest and the shortest mesh edge are stored in an array.
58      ' The shortest edge's length is stored at position 0.
59      ' The longest edge's lenght is stored at position 1.
60
61      ' Declaration of variables
62      Dim et, intNoOfEdges , i, currentEdge , dblEdgeLength , arrEdgePoints , EdgesLengths (3)
63      Dim dblSmallestLength , dblLargestLength
64      ' Get EvoluteTools plugin object
65      Set et = Rhino.GetPlugInObject ("EvoluteTools for Rhino")
66      If IsNull(et) Then Exit Function
67
68      dblSmallestLength  = 99999999
69      dblLargestLength  = 0
70      intNoOfEdges  = et.EmNumberOfEdges (TheMesh)
71
72      ' Loop through all edges off the mesh, find the longest
73      For i=0 To intNoOfEdges  - 1
74          currentEdge  = et.EmEdge(TheMesh, i)
75          '' Extract the endpoints of the edge,
76          '' measure the distance between them
77          arrEdgePoints  = et.EmGetEdgePoints (currentEdge )
78          dblEdgeLength  = Rhino.Distance(arrEdgePoints (0), arrEdgePoints (1))
79          If dblEdgeLength  < dblSmallestLength  Then
80              dblSmallestLength  = dblEdgeLength
81              ShortestEdge  = currentEdge
82          End If
83          If dblEdgeLength  > dblLargestLength  Then
84              dblLargestLength  = dblEdgeLength
85              LongestEdge  = currentEdge
86          End If
87      Next
88
89      ' Return length of the shortest and the longest mesh edge
90      EdgesLengths (0) = dblSmallestLength
91      EdgesLengths (1) = dblLargestLength
92
93      ' Function output
94      ExtremeEdges  = EdgesLengths
95
96  End Function
```

The user is asked to select a mesh and to specify a minimum edge length for the mesh. We find (similar to the previous example) the shortest mesh edge before we manipulate the mesh at all. We enter a loop on line 26. With each iteration of the loop we find the shortest mesh edge[10] and compare its length with the *dblMinEdgeLength* value (line 28). If the edge is shorter than the specified *dblMinEdgeLength* variable the script deletes the polyline associated with the neighboring edge. To find a neighboring edge we first need to find the edge's halfedge (line 29). From there we move to the next halfedge and set its edge as *polyEdge* (line 30). Next we have to check if this edge isn't a boundary edge because we don't want to delete the mesh boundaries. If *polyEdge* is a boundary edge we will have to redefine it. For this we go back to our initial halfedge (*currentHalfedge*), and from there we will go to the previous halfedge instead to the next. Now we can make the *polyEdge* variable store the handle of the edge associated to the *previousHalfedge* (line 33-34). Either way, after we defined the *polyEdge* variable we are going to delete the polyline associated to that edge using the

[10] We use the *ExtremeEdges* function from the previous examples again

et.MeshDeletePolyline function from EvoluteTools (line 37). Following, the mesh is optimized in order to distribute the polylines evenly. The script will remove polylines as long as the shortest mesh edge is shorter than the specified minimal edge length.

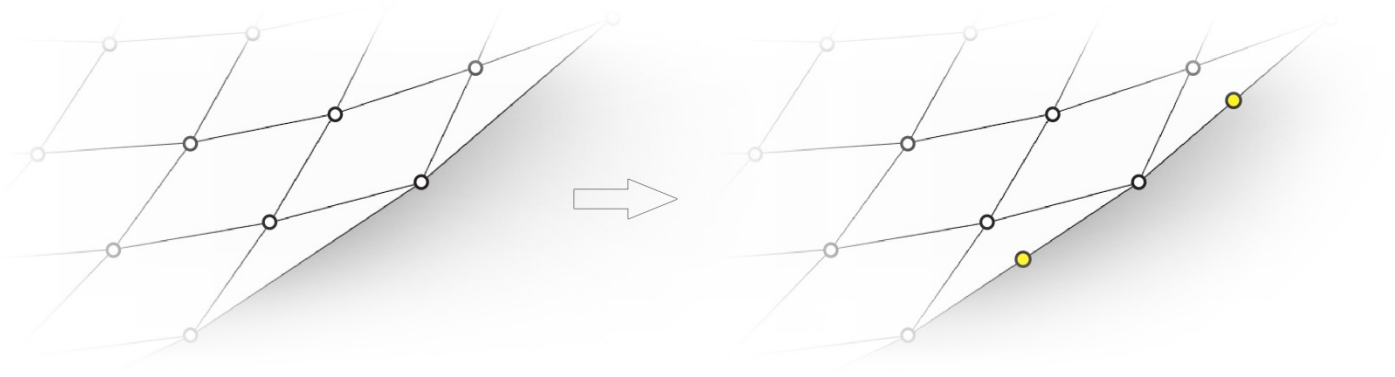


When the loop through is completed (line 26-42) the script optimizes the mesh one more time and measures the shortest edge after all the changes are done. The shortest edge's length before and after running the script are displayed in the command line for visual reference after the script completes (line 49-50).

3.3.5 Triangles to quads

The following example deals with the problem of mesh clean-up. During his research for the EvoluteTools Primer, Kris found that it would be helpful to have a script which would convert the triangular faces along the mesh boundary, which emerge when a quadrilateral mesh is diagonalized, into triangular shaped quad faces. This would allow you to continue working on a pure quadrilateral mesh after performing a diagonal subdivision. The mesh could further be subdivided with strip subdivision, which is not possible if the mesh has triangular faces along its borders. We've decided to showcase this script here in the EvoluteTools Scripting Primer.

The script will identify the triangular faces along the mesh boundary, find the boundary edge and divide it in two by adding a new vertex in the middle of the edge.



```
Option Explicit
'Script written by Marko Tomicic
'Script copyrighted by Evolute GmbH


Call TrianglesToQuads ()
' This script will find triangular faces on the mesh boundary and convert
' them to triangular shaped quad faces
Sub TrianglesToQuads ()

    Dim et, objMesh, intNoOfFaces , i, j, k, l, currentFace , faceValence
    Dim currentHalfedge , faceVertices , Points(1)
    Set et = Rhino.GetPlugInObject ("EvoluteTools for Rhino")

    ' Get user input
    objMesh = Rhino.GetObject("Select the mesh", 32)

    For l=0 To 1

        intNoOfFaces = et.EmNumberOfFaces (objMesh)

        ' Loop through mesh faces, convert boundary triangular faces to quads
        For i=0 To intNoOfFaces - 1
            currentFace = et.EmFace(objMesh, i)
            faceValence = et.EmFaceValence (currentFace )
            faceVertices = et.EmGetFaceVertices (currentFace )

            '' Continue only if the face is triangular, if not go to next face
            If faceValence = 3 Then
                currentHalfedge = et.EmHalfedge (currentFace )

                '' Find the face's boundary edge
                For j=0 To 2
                    currentHalfedge = et.EmNextHalfedge (currentHalfedge )
                    If et.EmIsBoundary (et.EmEdge(currentHalfedge )) Then
```

```
36
37                          '' Add Vertex to midpoint of boundary edge
38                          Dim midPoint, NewVertex, NewFace
39                          Points(0) = et.EmPoint(et.EmStartVertex (currentHalfedge ))
40                          Points(1) = et.EmPoint(et.EmEndVertex (currentHalfedge ))
41                          midPoint = MeanVector (Points)
42                          NewVertex = et.EmAddVertex (objMesh, midPoint)
43
44                          '' Add new vertex to the array of face vertices
45                          ReDim Preserve faceVertices (Ubound(faceVertices )+1)
46                          faceVertices (3) = NewVertex(0)
47
48                          '' Delete triangular face, create quadrilateral face
49                          Call et.EmDeleteFace (currentFace )
50                          NewFace = et.EmAddFace(faceVertices )
51
52                          Exit For
53                      End If
54
55                  Next
56
57              End If
58          Next
59      Next
60  End Sub
61
62  Function MeanVector (ByVal arrPoints)
63      Dim m : m = Array(0, 0, 0)
64      Dim p
65      For Each p In arrPoints
66          m = Rhino.VectorAdd(m, p)
67      Next
68      MeanVector = Rhino.VectorScale (m, 1 / (UBound(arrPoints) + 1))
69  End Function
```

The script starts by asking the user for an input mesh. Right after that it enters a loop (line 18-59). The loop starts before anything happens to the mesh and ends when everything else is done. The loop is used to repeat the whole script one more time after it has finished. This is because during the script, some mesh faces get deleted, others are created from scratch, so it happens that the script misses one or two faces when it is run only once. Sure there are more elegant ways to fix this problem, but since this script completes really fast and it is not a big deal in terms of time if it runs again one or two times I thought we could try this method for once. Inside the loop we count all the mesh faces (line 20). We then enter a new loop through all the faces on line 23. We select one face by its index and find its valence and its vertices (line 24-26). The loop will only continue if we have a triangular face, else it starts again with a new face (line 29). We find one halfedge and from there loop through all three of them to find the boundary edge of that face (line 30-35). When the boundary is found we extract its end points and find the midpoint using the *MeanVector* function (line 39-41). We add a new vertex in those coordinates to the mesh (line 42). In order to create a new face from the three vertices of the existing face and the newly created vertex we need to collect them into one array (line 45-46). The old triangular face is deleted and a new quadrilateral face is created in its position (line 49-50).

The *MeanVector* function takes any number of point coordinates as input and returns the average. It can be used (amongst other things) to calculate the coordinate of an edge's midpoint. We will use it later again in example 5.5.2 to find a face's midpoint. What it does is adding all point coordinates together (line 65-67) and finding the average x, y and z values (line 68).

4. Mesh Optimization

4.1 Optimization

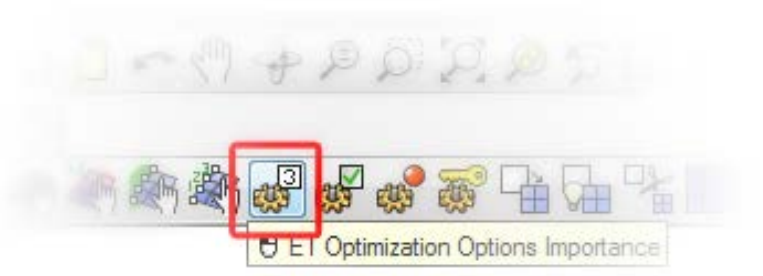
The fourth step in the EvoluteTools workflow is the optimization. In this step the subdivided mesh is bent into shape. The optimization process proceeds according to user specified optimization options.[11]

Create the reference surface
Create the coarse mesh
Subdivision
Optimization
Post processing

The final result of the optimization depends on the optimization options and the constraints which the user determines. Depending on the specifics of a project, there can be lots of constraints which you have to set. This process might become very exhausting and time consuming when dealing with large projects and large numbers of constraints. This chapter will try to deal with those problems using the EvoluteTools scripting interface. You will see some examples in which the script automates the complete workload around the setting of all the possible constraints. This should help you deal with such problems faster and more effective. It might seem to be a lot of work to write a complete script for a task like setting some vertices to be coplanar. It might be, if there are not too many of them and if you plan on doing it only once, but when you have a script which automates the task you can run it as many times as you want. This will give you the freedom to experiment and try different solutions for the given problem, plus when you start working on a new project you will have the script already waiting in your library.

4.2 Optimization options

The optimization options importance dialog is accessed by clicking the icon in the toolbar when working in Rhino.



Here we will set the importance of some optimization options depending on our needs in the particular case. I will explain just the few options which will be used the most here[12].

```
Importance ( SurfaceCloseness=3 CurveCloseness=1 OriginalCloseness=0 FairnessSprings=0.2 FairnessCurvature=2
FairnessCurvatureVariation=0 Ballpacking=0 Coplanarity=0 Planarity=0 Conical=0 Circular=0 NormalCloseness=0
IdealEdgeLength=0 IdealPanel=0 ):
```

[11] See Chapter 4.2 Optimization options
[12] For a more elaborate explanation of the optimization options see the EvoluteTools Primer

FairnessCurvature – *Fairness* or *niceness* determines the visual niceness of the mesh polylines. It tries to straighten the angles between two adjacent edges. Some of our scripts need some fairness to work

Coplanarity – *Coplanarity* will force vertices, which are flagged as coplanar to a plane, to snap to that plane

Planarity – *Planarity* tries to flatten faces. This might contradict other optimization options like Fairness.

IdealEdgeLength – *IdealEdgeLength* will adapt flagged edges to a specified length

4.3 Constraints

As mentioned above, the user is able to flag some constraints in the mesh prior to optimization. Usually this is done manually via the buttons on the EvoluteTools toolbar.

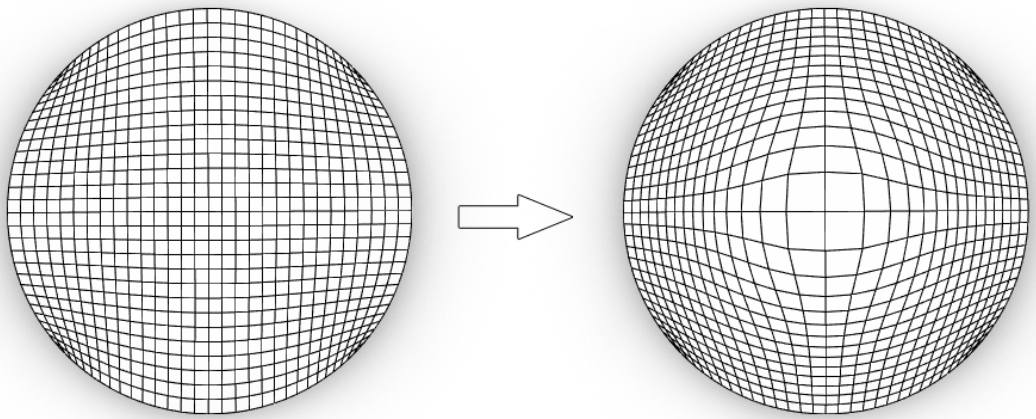
You can flag some vertices to stay in position during the optimization (ET Set Vertex Fixed), or to be curve points (ET Set Vertex Curve Point), you can set some edges to a desired length (ET Edge Length Optimization)[13] etc. The problem with this approach becomes clear when you work on large, complex meshes and you have to flag a large number of elements. On the following pages we will automate processes which would otherwise take hours to complete if you would work manually.

[13] For a more elaborate explanation of the flagging constraints see the EvoluteTools Primer

4.4 Examples

4.4.1 Face Attractor

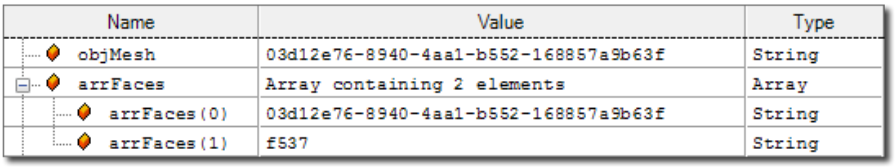
As usual we will start the chapter with an easy example. We will write a script which recreates an attracting/detracting effect within a mesh. The script will ask the user to select one or more faces and to tell it which is the desired edge length of those faces' edges. It then finds all those edges and flags them to be optimized for *Ideal Edge Length*. This little exercise is your inauguration to the optimization scripting with EvoluteTools. You will learn some basic principles about flaging constraints which will be used a lot in the following examples.



```
Option Explicit
'Script written by Marko Tomicic
'Script copyrighted by Evolute GmbH

Call FaceAttractor ()
' This script flags the edges of one or more selected faces to be optimized
' for ideal edge length. This script works with triangular and quadrilateral meshes.
Sub FaceAttractor ()
    ' Declaration of variables
    Dim et, objMesh, arrFaces, i, arrEdges, strCmd, strCmd2, arrResults

    ' Get EvoluteTools plugin object
    Set et = Rhino.GetPlugInObject ("EvoluteTools for Rhino")
    If IsNull(et) Then Exit Sub

    ' Get input
    objMesh = Rhino.GetObject("Select the mesh", 32)
    arrFaces = et.PickFace(0, 0, "Select the faces")

    ' Flag edges of the selected face(s)
    '' If only one face is selected
    If Ubound(arrFaces) = 1 And Not IsArray(arrFaces(0)) Then
        Dim singleEdges
        singleEdges = et.EmGetFaceEdges (arrFaces)
        Call et.SetEdgeLengthOptimization (singleEdges )
    Else
        '' If more than one face is selected
        For i=0 To Ubound(arrFaces)
            arrEdges = et.EmGetFaceEdges (arrFaces(i))
            Call et.SetEdgeLengthOptimization (arrEdges)
        Next
    End If

```

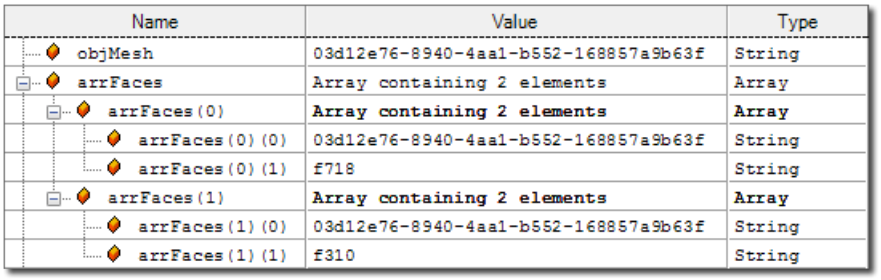
```
34      ' Command line
35      strCmd = "! _etOptionsImportance _IdealEdgeLength "
36      strCmd2 = "! _etOptionsToggles _IdealEdgeLength "
37      Call Rhino.Command(strCmd, 0)
38      Call Rhino.Command(strCmd2, 0)
39
40      ' Optimize!
41      Call et.Optimize(objMesh, arrResults )
42
43  End Sub
```

We start our script by asking the user to select one or more faces (line 18). The edges of those faces will be set to a fixed length. The script would be much shorter and simpler if we hadn't to distinguish between two possible cases: the case when only one face is selected and the case when more than one face is selected. Before we move on, let me explain what difference it makes. Faces are stored as arrays in the ET scripting interface. An array holding one face (*arrFaces*) contains two elements: the mesh ID (*arrFaces (0)*) and the face handle (*arrFaces (1)*).



Name	Value	Type
objMesh	03d12e76-8940-4aa1-b552-168857a9b63f	String
arrFaces	Array containing 2 elements	Array
arrFaces(0)	03d12e76-8940-4aa1-b552-168857a9b63f	String
arrFaces(1)	f537	String

If the user selects two or more faces, then the *arrFaces* array becomes nested. This means that the elements inside the array are arrays themselves. In the image above you see that both elements in the *arrFaces* array are strings. The image below shows what happens when there are two faces in the array.



Name	Value	Type
objMesh	03d12e76-8940-4aa1-b552-168857a9b63f	String
arrFaces	Array containing 2 elements	Array
arrFaces(0)	**Array containing 2 elements**	**Array**
arrFaces(0)(0)	03d12e76-8940-4aa1-b552-168857a9b63f	String
arrFaces(0)(1)	f718	String
arrFaces(1)	**Array containing 2 elements**	**Array**
arrFaces(1)(0)	03d12e76-8940-4aa1-b552-168857a9b63f	String
arrFaces(1)(1)	f310	String

Each element of *arrFaces* is another array holding two elements. The variable *arrFaces(0)* was the mesh ID in the first case and now it became the first of two selected faces. The variable *arrFaces(1)* was originally the face handle of the selected face and now it is the second of two selected faces. Both of these arrays hold the mesh ID and the face handle of their respective faces.

To flag all edges of the selected faces we need to perform a loop through all selected faces, find their edges and flag them. Because of the way faces are stored we might run into problems when the user selects only one face. We determine the number of repetitions of the loop with the number of faces (if there are six faces selected we want to repeat the loop six times). The loop is defined as follows:

```
For j=0 To Ubound(arrFaces)
```

This works well for more than one face because the *Ubound* function returns the number of elements in an array and thus the loop makes the right number of repetitions. This loop would however repeat two times if only one face was selected because the *arrFaces* array would still hold two elements: the mesh ID and the face handle.

According to the distinction between one and more than one selected faces that we made above the script is divided in two parts.

The first part (line 22-26) will be executed if the user selects one face. The second part (line 28-32) will be executed if more than one face is selected. First we need to know how many elements there are in the arrFaces array. If *Ubound(arrFaces)* returns any value different than one it means that there is more than one face in the arrFaces array and the script jumps directly to line 28. If *Ubound(arrFaces)* returns one it means that the array holds two elements. We still don't know if it holds one or two faces because of the way faces are stored as explained above. To be sure the script asks if the first element in the array (arrFaces (0)) is another array. If it is, it means that the arrFaces array holds two arrays, or in other words, two faces. The script jumps to line 28.

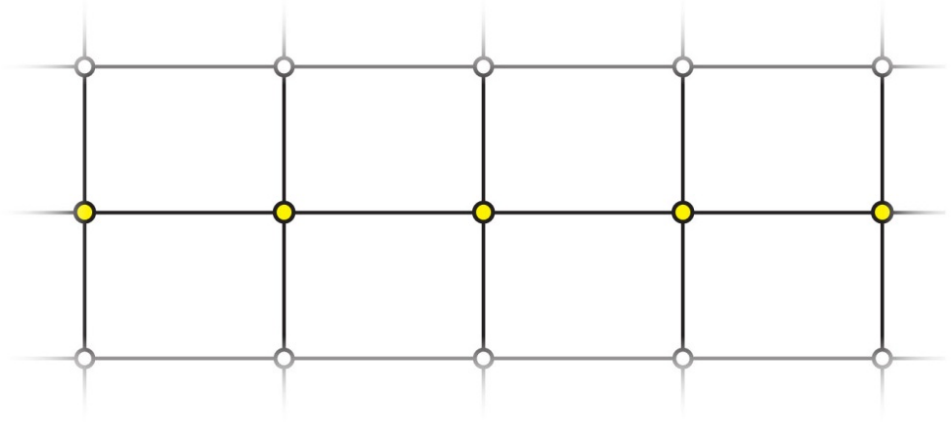
Now that we double checked and know for sure that the user has only one face selected, we can go on and extract the face's edges and flag them for edge length optimization (line 24-25). In this case (when one face is selected) the script would now jump to line 34.

If there were more than two faces selected in the beginning, the script would jump from line 22 to line 26. It loops through all selected faces (line 28). Now we only need to extract the edges of each face and flag them for edge length optimization (line 29-30).

When all face edges that we wanted to flag are flagged, the script continues on line 37. We invoke the command line to set the *etOptionsImportance* and the *etOptionsToggles* parameters for our edges (line 35-38) before optimizing the mesh (line 41).

4.4.2 Closest Polyline Vertices Function

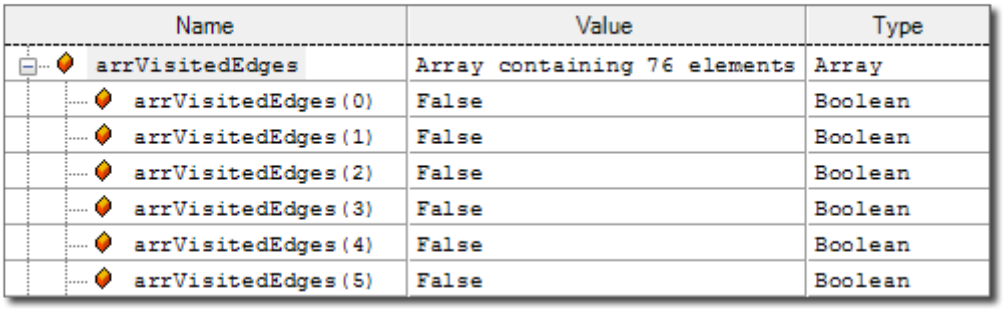
The following function will be very important for some of the forthcoming scripts, thus it will be explained in detail here in an autonomous example. The function takes the mesh and a plane object as input and returns the set of mesh vertices belonging to a polyline which are closest to the input plane. It does so by looping through all mesh polylines and measuring the distance from every polyline vertex to the plane. The maximum distance is returned for every polyline. The script then compares the results and returns the vertices belonging to the closest polyline. The image below shows mesh vertices which belong to the same polyline.



```
Function ClosestPolylineVertices (ByVal objMesh, ByVal objPlane, ByVal Tolerance, ByVal
PlaneNumber , ByVal NoOfPlanes)
  ' This function loops through mesh polylines and finds the one which is
  ' on average closest to a input plane. This function works with quadrilateral meshes.

  ' Declaration of variables
  Dim et, intNoOfEdges , i, j, k, arrVisitedEdges
  ' Get EvoluteTools plugin object
  Set et = Rhino.GetPlugInObject ("EvoluteTools for Rhino")
  If IsNull(et) Then Exit Function

  '' Find number of edges, create array of visited edges
  intNoOfEdges = et.EmNumberOfEdges (objMesh)

  ReDim arrVisitedEdges ((intNoOfEdges )-1)
  For i=0 To Ubound(arrVisitedEdges )
      arrVisitedEdges (i) = False
  Next

  '' For every edge find the polyline and all of its vertex coordinates
  Dim dblSmallestMax : dblSmallestMax = 100000
  For j=0 To intNoOfEdges - 1
      Dim arrCurrentEdge : arrCurrentEdge = et.EmEdge(objMesh, j)
      Dim dblDistance , arrEdgePoints
      arrEdgePoints = et.EmGetEdgePoints (arrCurrentEdge )
      dblDistance = Abs(Rhino.DistanceToPlane (objPlane, arrEdgePoints (0)))

      If Not arrVisitedEdges (j) = True Then
          If Not et.EmIsBoundary (arrCurrentEdge ) Then
              If Not dblDistance > Tolerance Then
                  '' Remember the visited edges for a faster query
                  Dim arrHalfedge , intIndex, nextHalfedge , currentHalfedge
                  nextHalfedge = et.EmHalfedge (arrCurrentEdge )
                  Do
                      '' Advance
                      currentHalfedge = nextHalfedge
                      '' Compute next halfedge
                      nextHalfedge = et.EmNextHalfedge (et.EmOppositeHalfedge
                      (et.EmNextHalfedge (currentHalfedge )))
                  Loop While Not ( et.EmVertexValence (et.EmEndVertex (currentHalfedge )) <>
                  4 Or et.EmEdge(nextHalfedge )(1) = arrCurrentEdge (1))
                  '' Change direction
                  currentHalfedge = et.EmOppositeHalfedge (currentHalfedge )
                  Do
                      intIndex = et.EmIndex(et.EmEdge(currentHalfedge ))
                      arrVisitedEdges (intIndex) = True
```

```
44                          currentHalfedge  = et.EmNextHalfedge (et.EmOppositeHalfedge
                            (et.EmNextHalfedge (currentHalfedge )))
45                          Loop While Not ( et.EmIsBoundary (currentHalfedge ) Or et.EmFaceValence
                        (et.EmFace(currentHalfedge ))
                        <> 4 Or et.EmFaceValence (et.EmFace(et.EmOppositeHalfedge (currentHalfedge
                        ))) <> 4 Or et.EmEdge(currentHalfedge )(1) = arrCurrentEdge(1))
46
47                          '' Measure distances to given plane, find maximum value and store it with
48                          '' the polyline it belongs to
49                          Dim arrCurrentPoints  : arrCurrentPoints  = et.EmGetPolylinePoints
                            (arrCurrentEdge )
50                          ReDim arrDist(-1)
51
52                          For k=0 To Ubound(arrCurrentPoints )
53                               ReDim Preserve arrDist(Ubound(arrDist)+1)
54                               Dim arrPoint  : arrPoint  = arrCurrentPoints (k)
55
56                               If IsArray(arrPoint) Then
57                                    If IsArray(objPlane) Then
58                                         arrDist(Ubound(arrDist))  = Rhino.DistanceToPlane (objPlane,
                                           arrPoint)
59                                         arrDist(Ubound(arrDist))  = Abs(arrDist(Ubound(arrDist)))
60                                    End If
61                               End If
62                          Next
63
64                          Dim dblCurrentMax  : dblCurrentMax  = Rhino.Max(arrDist)
65
66                          If dblCurrentMax  <= dblSmallestMax  Then
67                               dblSmallestMax  = dblCurrentMax
68                               Dim arrClosestEdge  : arrClosestEdge  = arrCurrentEdge
69                          End If
70                     End If
71                End If
72           End If
73      Next
74
75      '' Take vertices from closest polyline
76      Dim arrClosestVertices
77      arrClosestVertices  = et.EmGetPolylineVertices (arrClosestEdge )
78
79      Rhino.Print "Plane number " & CStr(PlaneNumber ) & " of " & CStr(NoOfPlanes )
80      Rhino.Print "Smallest maximum distance: " & CStr(dblSmallestMax )
81      ClosestPolylineVertices  = arrClosestVertices
82 End Function
```

The variables for this function are defined on line 6. Note that the mesh (*objMesh*), the plane (*objPlane*), *Tolerance*, *PlaneNumber* and *NoOfPlanes* are inputs in the function and therefore they should be defined in the script before the function is called. We find the number of edges in the mesh on line 12. Later we will loop through all edges but not one at a time. This time we will start the loop with one edge, but in the same loop iteration we will visit all other edges which belong to the polyline of the first edge. We want to make sure not to visit the same edge two or more times because that would make our script slow and maybe even unusable. We create an array, *arrVisitedEdges* (line 14) which will store all the edges of the mesh and provide information about whether the edge has been visited before or not. Right after the *arrVisitedEdges* array has been redimensioned on line 14, it offers only empty slots for all mesh edges. We will loop once through all mesh edges and set the Boolean `False` for every edge (line 15-17). The following image shows some of the elements of the *arrVisitedEdges* array at this point.



Name	Value	Type
arrVisitedEdges	Array containing 76 elements	Array
arrVisitedEdges(0)	False	Boolean
arrVisitedEdges(1)	False	Boolean
arrVisitedEdges(2)	False	Boolean
arrVisitedEdges(3)	False	Boolean
arrVisitedEdges(4)	False	Boolean
arrVisitedEdges(5)	False	Boolean

Later we will change the Boolean from `False` to `True` for each edge we visit. First we need to set the *dblSmallestMax* variable to a very high value to make sure that the maximum distance of the first polyline which we will measure is smaller than the current value (line 20). We now enter a loop through all mesh edges. First, we will take the edge with the index "j" and measure the distance of one of its points to the plane (line 25). After getting the distance of one of the edge's end points to the plane it is time to check if this edge has been visited before (line 27). Note that we set all edges to `False` before entering this loop to make sure that the first edge will pass this test. Further we check if the current edge (if it hasn't been visited before) is a boundary edge (line 28). Since we are not interested in the mesh boundary, here the script will move to the next edge if it is the case. If the edge has passed all previous steps we check the distance of one of that edge's end points to our plane. If it is larger than the provided *Tolerance,* we can eliminate this edge right away because there surely are other edges in the mesh which are closer to the plane. If this is the case the script will start the loop again and go to the next mesh edge. If our edge (*arrCurrentEdge*) has passed all the previous tests, we go on and extract its halfedge (line 32). With that halfedge we enter a loop where we find all other edges in its polyline and set their respective Boolean value in the *arrVisitedEdges* array to `True`. First the index of the current edge is extracted (line 42), the edge is identified in *arrVisitedEdges* with its index and the Boolean is set to `True` (line 43). The image below shows the *arrVisitedEdges* array after some edges have been visited.

Name	Value	Type
arrVisitedEdges	Array containing 76 elements	Array
arrVisitedEdges(0)	True	Boolean
arrVisitedEdges(1)	True	Boolean
arrVisitedEdges(2)	True	Boolean
arrVisitedEdges(3)	False	Boolean
arrVisitedEdges(4)	True	Boolean
arrVisitedEdges(5)	False	Boolean

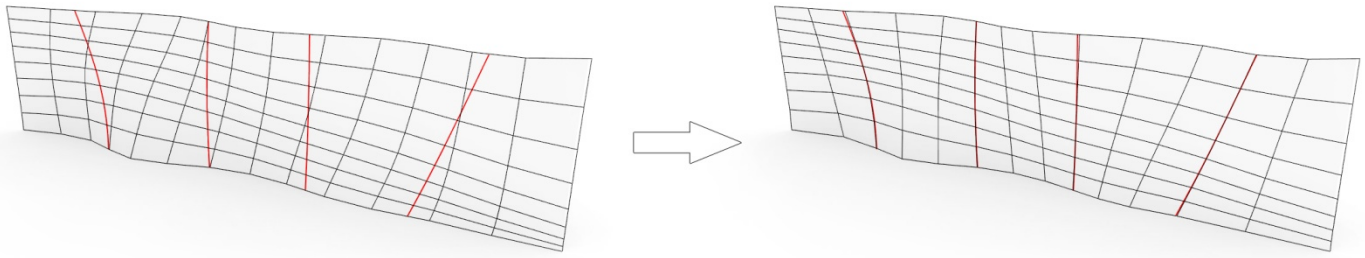
On line 34 we advance to the next edge/halfedge of the polyline. The loop is repeated until we hit a boundary `Or` the face valence of one of the edge's faces is different than four `Or` until we arrive at the starting edge (line 45). Now that we have remembered all visited edges, in order to not visit them over and over again, we get all polyline points. EvoluteTools has a pretty handy function for this which allows us to get them all at once with one line of code, instead of having to loop again through all polyline halfedges and extract the start and end points of each halfedge. The function is called *et.EmGetPolylinePoints* and it's used on line 49. It takes only one edge as input and returns the point coordinates of all vertices which belong to that edge's polyline. The point coordinates are now stored in *arrCurrentPoints*. On line 52 we start a loop through all points in *arrCurrentPoints* in which we will measure the distance of each point to the plane and calculate the average distance. The *arrDist* array is used to store the distance of every point as a double. Every time the loop repeats, the array is broadened by one new space to store the new information. You have seen this technique in action before. Before measuring the distance between the point and the plane, the script checks quickly if everything is ok (line 56-57). The distance is measured on line 58. You need to be careful in this sort of situations. If the point is behind the plane, the

Rhino.DistanceToPlane function returns a negative value. For that reason we will take the absolute value of what is returned by *Rhino.DistanceToPlane* (line 59). You could do this using just one line if you like. After the distance from every polyline vertex to the plane is calculated and stored in *arrDist,* we calculate the maximum distance of that polyline to the plane (line 64). The current maximum distance is compared to the smallest maximum distance so far, and if it is shorter, it will replace the smallest maximum distance *dblSmallestMax*. The *arrCurrentEdge* is then set to be the *arrClosestEdge* (line 66-68). After we have determined which the closest polyline to our plane is, we store its vertices in the *arrClosestVertices* array (line 77). The script will print the smallest maximum distance together with the number of the current plane to the command line for visual reference while the script is running (line 79-80). This can be deleted or commented out if you don't want it to appear and slow down your computer.

The last step is to define the output of the function. This function will return the vertices which belong to the polyline which is closest to an input plane (line 81).

4.4.3 Snap Mesh to Curves

The following script will enable you to align the mesh polylines as close as possible to previously drawn curves on the reference geometry. It creates a plane through the start, mid and endpoint of a curve and sets mesh vertices to be coplanar to that plane, so it is accurate only when the curves are coplanar. Maybe a good exercise for you would be to try to find another solution which would be accurate regardless of the shape of the input curves.



```
Option Explicit
'Script written by Marko Tomicic
'Script copyrighted by Evolute GmbH

Call SnapMeshToCurves ()
' This script will snap the closest mesh vertices to curves on the
' reference geometry. This script works with quadrilateral meshes.
Sub SnapMeshToCurves ()
    ' Declaration of variables
    Dim et, objMesh, arrCurves, Tolerance, i, j , arrCurvePoints (), objPlane()
    Dim closestVertices (), arrVisitedEdges (), arrResult

    ' Get EvoluteTools plugin object
    Set et = Rhino.GetPlugInObject ("EvoluteTools for Rhino")
    If IsNull(et) Then Exit Sub

    ' Get the mesh and the curves
    objMesh = Rhino.GetObject("Select the mesh", 32)
    arrCurves = Rhino.GetObjects ("Select the curves", 4)
    Tolerance = Rhino.GetReal("Maximum distance from curve to be considered? ", 3)

    ReDim closestVertices (-1)
    ReDim objPlane(-1)

    ' Loop through selected curves, snap closest vertices to the curve
    For i=0 To Ubound(arrCurves)
        Dim currentCurve
        currentCurve = arrCurves(i)

        '' Check if it is a valid curve
        If Not Rhino.IsCurve(currentCurve ) Then
            Exit Sub
        End If

        '' Array for the 3 curve points
        ReDim arrCurvePoints (2)

        '' Get Start, Mid and Endpoint of the curve
        arrCurvePoints (0) = Rhino.CurveStartPoint (currentCurve )
        arrCurvePoints (1) = Rhino.CurveMidPoint (currentCurve )
        arrCurvePoints (2) = Rhino.CurveEndPoint (currentCurve )

        '' Create Plane from the three curve points
        ReDim Preserve objPlane(Ubound(objPlane)+1)
```

```
45          objPlane(i) = Rhino.PlaneFitFromPoints (arrCurvePoints )
46          '' Find closest set of vertices for that plane
47          ReDim Preserve closestVertices (Ubound(closestVertices )+1)
48          closestVertices (i) = ClosestPolylineVertices (objMesh, objPlane(i), Tolerance,
            i+1, Ubound(arrCurves)+1)
49      Next
50
51      ' Flag vertices as coplanar
52      For j=0 To Ubound(objPlane)
53          Call et.SetVerticesCoplanar (closestVertices (j), 7, 1, objPlane(j))
54      Next
55      ' Optimize!
56      Call et.Optimize(objMesh, arrResult)
57
58  End Sub
59
60
61  Function ClosestPolylineVertices (ByVal objMesh, ByVal objPlane, ByVal Tolerance,
        ByVal PlaneNumber , ByVal NoOfPlanes )
.
.
.
```

We start by asking the user for a mesh, some curves to which the mesh should be aligned and a tolerance value (line 18-20). The curves should be on the surface and coplanar for best results. We will now loop through all selected curves. For each curve there will be one plane created through its start, mid and endpoint. We will then find the closest mesh polyline to that plane using the *ClosestPolylineVertices* function that we discussed in the previous example. We enter the loop on line 26. The loop repeats once for every selected curve *(Ubound(arrCurves))*. The *currentCurve* variable holds the selected curve with the index "i" (line 28). The script will check if everything went according to plan (line 31-33). We will now extract the curve's start, mid and endpoint to create a plane from those points. First, we need an array with three empty slots to store our points (line 36). Now we can extract the three points and save them to the right place in the array (line 39-41). The final steps are to create a plane from those points and to find the closest polyline vertices to that plane. We need to store the plane and its closest polyline vertices for later, thus we need an array for each. The arrays have been defined as empty arrays on line 22 and 23. Now we will add one empty space to the arrays in each loop repetition (line 44 and 47). The plane which we create from our three curve points is stored as new element in the *objPlane* array (line 45). We use the *ClosestPolylineVertices* function to find the closest set of vertices to that plane, those vertices are then stored in the *closestVertices* array (line 48). Notice that the plane and the vertices are stored under the same index "i" in their respective arrays. The function *ClosestPolylineVertices* takes five input parameters: the mesh, the plane, the tolerance, the plane number and total number of planes.

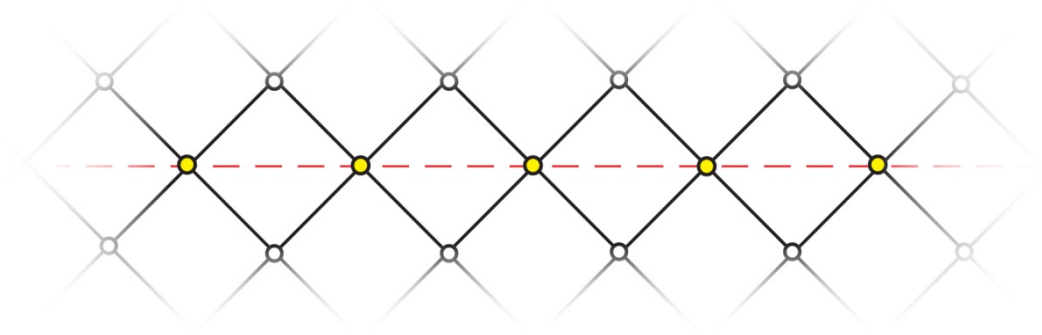
```
Function ClosestPolylineVertices (ByVal objMesh, ByVal objPlane, ByVal Tolerance,
ByVal PlaneNumber , ByVal NoOfPlanes )
```

The number of the current plane *(planeNumber)* is *i+1* and the total number of planes *(NoOfPlanes)* is the number of elements in the *arrCurves* array *(Ubound(arrCurves)+1)*.

After the loop has completed we will have an array of "n" planes and an array of "n" sets of vertices. We will loop now through all planes in *objPlane* and set the vertices from *closestVertices* with the same index to be coplanar to that plane (line 52-54). We optimize the mesh in the end to see the result in the viewport.

4.4.4 Closest Non Polyline Vertices function

This function returns the set of vertices which are closest to an input plane, but do not belong to a common polyline. Those vertices belong to an imagined polyline which is diagonal to the mesh faces. The image below shows the set of vertices which we will try to find automatically. This technique will be used practically in the next example (4.4.5) but first, let's take a look at how this function works.



```
Function ClosestNonPolylineVertices (ByVal objMesh, ByVal objPlane, ByVal Tolerance, ByVal
    PlaneNumber , ByVal NoOfPlanes)
    ' This function loops through mesh vertices and finds the group of vertices
    ' which is closest to an input plane. This function works with quadrilateral meshes.

    ' Declaration of variables
    Dim et, i, j, intNoOfEdges , intNoOfHalfedges , arrVisitedHalfedges , startEdge
    Dim dblSmallestMax , closestSetOfVertices , Distance

    ' Get EvoluteTools plugin object
    Set et = Rhino.GetPlugInObject ("EvoluteTools for Rhino")
    If IsNull(et) Then Exit Function

    intNoOfEdges = et.EmNumberOfEdges (objMesh)
    intNoOfHalfedges = intNoOfEdges * 2

    ReDim arrVisitedHalfedges ((intNoOfHalfedges )-1)
    For i=0 To Ubound(arrVisitedHalfedges )
        arrVisitedHalfedges (i) = False
    Next

    dblSmallestMax = 99999999999
    For j=0 To intNoOfHalfedges - 1
        Dim currentHalfedge , nextHalfedge , arrVertices , dblDistances , dblCurrentMax , intIndex,
        otherHalfedge

        ' Check whether the current halfedge has been visited before
        If arrVisitedHalfedges (j) = False Then
            nextHalfedge = et.EmHalfedge (objMesh, j)
            Distance = Rhino.DistanceToPlane (objPlane, et.EmPoint(et.EmStartVertex
          (nextHalfedge )))
            Distance = Abs(Distance)

            If Distance < Tolerance Then
                Rhino.Print "Plane number " & CStr(PlaneNumber ) & " of " & CStr(NoOfPlanes )
                Rhino.Print "Halfedge number " & CStr(j) & " of " & CStr(intNoOfHalfedges -
                1)
                startEdge = et.EmEdge(nextHalfedge )

                ' Remember visited halfedges for a faster query
                Do
                    '' Advance
                    currentHalfedge = nextHalfedge
                    '' Compute next halfedge
                    nextHalfedge = et.EmNextHalfedge (et.EmOppositeHalfedge (et.EmNextHalfedge
            (et.EmOppositeHalfedge (et.EmNextHalfedge (et.EmNextHalfedge (currentHalfedge ))))))
```

```
42                  Loop While Not (et.EmIsBoundary (et.EmEdge(nextHalfedge )) Or et.EmVertexValence
            (et.EmStartVertex ( currentHalfedge )) <> 4 Or et.EmVertexValence (et.EmEndVertex
            (currentHalfedge )) <> 4 Or et.EmEdge(nextHalfedge )(1) = startEdge(1))
43                  '' Change direction
44                  nextHalfedge  = et.EmOppositeHalfedge (currentHalfedge )
45
46                  ReDim arrVertices (-1)
47                  ReDim dblDistances (-1)
48                  '' Remember all visited halfedges and store start vertex of each halfedge
49                  Do
50                       currentHalfedge  = nextHalfedge
51                       '' Remember visited halfedge
52                       intIndex = et.EmIndex(currentHalfedge )
53                       arrVisitedHalfedges (intIndex) = True
54
55                       '' Remember other halfedge
56                       otherHalfedge  = et.EmNextHalfedge (et.EmNextHalfedge (currentHalfedge ))
57                       intIndex = et.EmIndex(otherHalfedge )
58                       arrVisitedHalfedges (intIndex) = True
59
60                       '' Save start vertex to an array
61                       ReDim Preserve arrVertices (Ubound(arrVertices )+1)
62                       arrVertices (Ubound(arrVertices )) = et.EmStartVertex (currentHalfedge )
63                       '' Save distance to plane
64                       ReDim Preserve dblDistances (Ubound(dblDistances )+1)
65                       dblDistances (Ubound(dblDistances )) = Rhino.DistanceToPlane (objPlane,
                         et.EmPoint(et.EmStartVertex (currentHalfedge )))
66                       dblDistances (Ubound(dblDistances )) = Abs(dblDistances (Ubound(dblDistances )))
67
68                       '' Compute next halfedge
69                       nextHalfedge  = et.EmNextHalfedge (et.EmOppositeHalfedge (et.EmNextHalfedge
            (et.EmOppositeHalfedge (et.EmNextHalfedge (et.EmNextHalfedge (currentHalfedge ))))))
70                  Loop While Not (et.EmIsBoundary (et.EmEdge(nextHalfedge )) Or
            et.EmVertexValence (et.EmStartVertex ( currentHalfedge )) <> 4 Or
            et.EmVertexValence (et.EmEndVertex (currentHalfedge )) <> 4 Or
            et.EmEdge(nextHalfedge )(1) = startEdge(1))
71
72                  dblCurrentMax  = Rhino.Max(dblDistances )
73                  If dblCurrentMax  <= dblSmallestMax  Then
74                       dblSmallestMax  = dblCurrentMax
75                       closestSetOfVertices  = arrVertices
76                  End If
77              End If
78          End If
79      Next
80      ClosestNonPolylineVertices  = closestSetOfVertices
81 End Function
```

The function takes five inputs just like the *ClosestPolylineVertices* function. Those variables have to be defined in the script which calls this function.[14] It first finds the number of edges and the number of halfedges in the mesh (line 13-14). Similar to the *ClosestPolylineVertices* function, we will create an array for all halfedges. With a quick loop through that array the Boolean value `False` is set for all halfedges. Later, when we visit a halfedge this value will be changed to `True`. The *dblSmallestMax* value is set to a very high value to make sure that the first measured value is smaller and therefore replaces it (line 21). The function starts looping through the mesh halfedges on line 22 and checks if the current halfedge has been visited before. Only if it has not been visited the function proceeds, else it starts over again with the next halfedge (line 26). The distance of the halfedge's start vertex to the input plane is measured (line 27-29). Only if that distance is within a given tolerance the function proceeds (line 31). Now that our halfedge has passed all previous tests we print some info about it to the command line (line 32-33). The function remembers the edge to which the current halfedge belongs for later (line 34). The next step is to move along the imagined polyline until we hit a boundary or until we arrive at the *startEdge* from line 34 again (line 37-42). The loop is repeated until

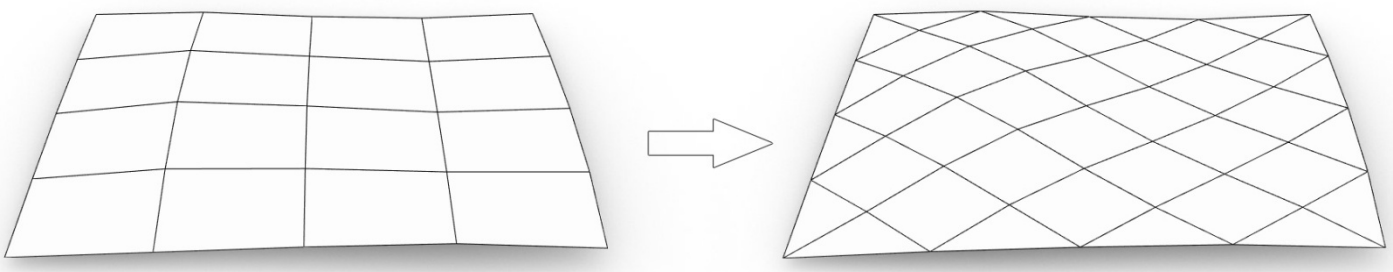
[14] See next example – 4.4.5 Snap Vertices To Planes

we hit a boundary, `Or` the vertex valence of the start or end vertex of our *currentHalfedge* becomes different than four, `Or` until we arrive at the starting edge (line 42). On line 44 we change the movement direction. This plays no role if the polyline is closed and we previously found the start edge after looping through all polyline edges. If we find a mesh boundary, then we have to turn around and walk in the opposite direction to the other mesh boundary[15]. The next step is to remember all visited halfedges in order not to visit them repeatedly. This is done by extracting the current halfedge's index and changing the Boolean value at this spot in the *arrVisitedHalfedges* array to `True` (line 50-53). Each halfedge in a quad mesh has a pair on the opposite side of its face which belongs to the same imagined polyline. This halfedge is memorized on line 56-58. Now the start vertex of the current halfedge is added to the *arrVertices* array and its distance to the input plane to the *dblDistances* array (line 61-66). The function moves to the next halfedge in our imagined polyline until it hits a boundary or the *startEdge* (line 69-70). When all vertices belonging to a polyline are found and their distances to the plane stored, we find the largest distance (line 72). This value is compared to the currently smallest maximum distance. If *dblCurrentMax* is smaller than *dblSmallestMax* (line 73) it will replace it and become *dblSmallestmax* (line 74). The current set of vertices will become the *closestSetOfVertices* (line 75), which is also the output of this function (line 80).

[15] See image 2.6.4 Select Polyline

4.4.5 Is Mesh Diagonalized function

This is a very short function which asks the user whether the input mesh has been diagonalized or not. We will use this function in the following example but there will surely be other situations where you can use the same function again. The mesh becomes "diagonalized" if it is subdivided with the *Diagonalize* option. *Diagonalize* adds a new vertex in the middle of each face and connects it to the existing corner vertices. The image below shows a mesh before and after diagonalization.

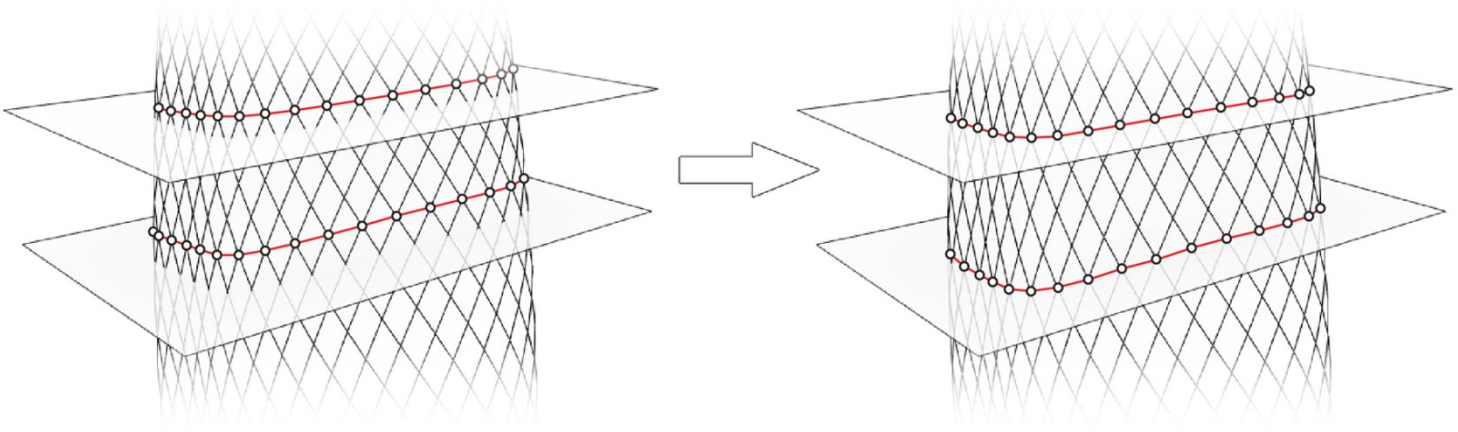


```
1 Function  IsMeshDiagonalized
2     ' Declaration  of variables
3     Dim  arrItems,  arrDefaults , arrResults , blnVal
4
5     arrItems  = Array("Diagonalized" , "No", "Yes")
6     arrDefaults  = array(False)
7
8     arrResults  = Rhino.GetBoolean ("Is the mesh diagonalized" , arrItems,
  arrDefaults )
9
10    If IsArray(arrResults ) Then
11        For Each blnVal In arrResults
12            Rhino.Print  blnVal
13        Next
14    End If
15
16    IsMeshDiagonalized  = arrResults (0)
17 End Function
```

The function is called in the script without providing any input information. Two default arrays are defined on line 5 and 6. The function then asks the user if the mesh has been diagonalized or not (line 8). The result of this query is printed to the command (line 10-14) and the *IsMeshDiagonalized* output is set to be *arrResults* (line 16). Since we used the *Rhino.GetBoolean* function on line 8 the *IsMeshDiagonalized* function can only return `True` or `False`.

4.4.6 Snap Vertices to Planes

This script will let the user choose a mesh and a set of surface objects (planes) from the Rhino viewport. It will then find the closest set of vertices to each plane and flag them as coplanar to the corresponding plane. This can be used to align the mesh automatically to floor slabs in a tower.



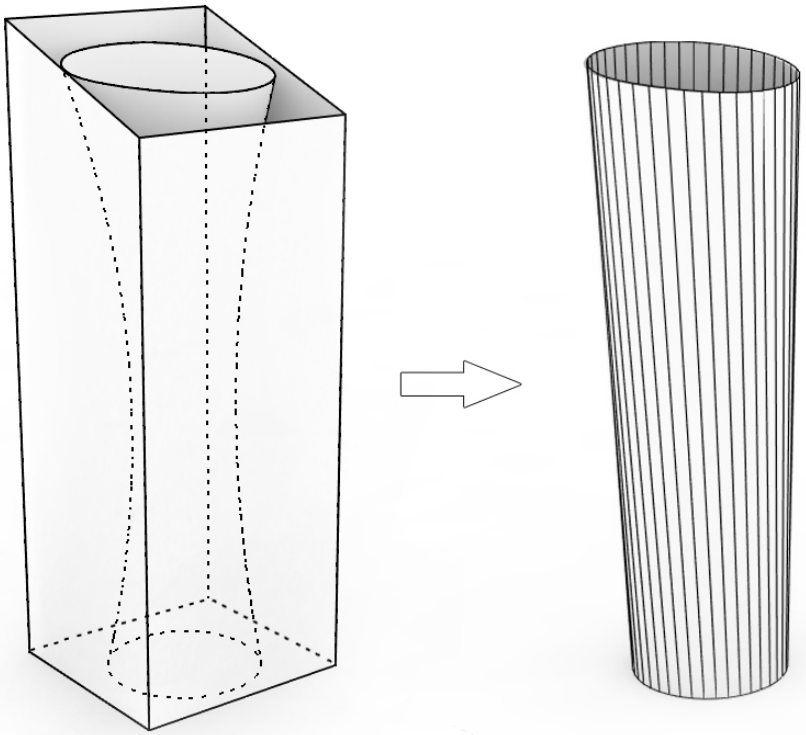
```
Option Explicit
'Script written by Marko Tomicic
'Script copyrighted by Evolute GmbH
Call SnapVerticesToPlanes ()
' This script will snap the closest set of mesh vertices to planes in the viewport
' This script works with quadrilateral meshes.
Sub SnapVerticesToPlanes ()
    ' Declaration of variables
    Dim et, objMesh, objSurfaces , arrCurves, blnDiagonalized , Tolerance
    Dim arrSurfacePoints , i, j , k, arrCurvePoints (), objPlane()
    Dim closestVertices (), arrVisitedEdges (), arrResult
    ' Get EvoluteTools plugin object
    Set et = Rhino.GetPlugInObject ("EvoluteTools for Rhino")
    If IsNull(et) Then Exit Sub

    ' Get user input
    objMesh = Rhino.GetObject("Select the mesh", 32)
    objSurfaces = Rhino.GetObjects ("Select the planes", 8)
    Tolerance = Rhino.GetReal("Maximum distance from plane to be considered? ", 5)

    ' Ask the user if the mesh is diagonalized
    blnDiagonalized = IsMeshDiagonalized

    ' Convert input surfaces to planes
    ReDim closestVertices (-1), objPlane(-1)
    For k=0 To Ubound(objSurfaces )
        ReDim Preserve objPlane(Ubound(objPlane)+1)
        arrSurfacePoints = Rhino.SurfaceEditPoints (objSurfaces (k))
        objPlane(Ubound(objPlane)) = Rhino.PlaneFitFromPoints (arrSurfacePoints )
    Next

    ' Loop through selected planes, snap closest vertices to each plane
    For i=0 To Ubound(objPlane)
        '' Find closest set of vertices for that plane
        ReDim Preserve closestVertices (Ubound(closestVertices )+1)
        If blnDiagonalized = True Then
          closestVertices (i) = ClosestNonPolylineVertices (objMesh, objPlane(i),
          Tolerance, i + 1, Ubound(objPlane)+ 1)
        Else
```

```
39          closestVertices (i) = ClosestPolylineVertices (objMesh, objPlane(i), Tolerance,
            i+1, Ubound(objPlane)+1)
40          End If
41      Next
42
43      ' Flag vertices as coplanar
44      For j=0 To Ubound(objPlane)
45          Call et.SetVerticesCoplanar (closestVertices (j), 7, 1, objPlane(j))
46      Next
47
48      ' Optimize!
49      Call et.Optimize(objMesh,  arrResult)
50  End Sub
51
52
53  Function ClosestPolylineVertices (ByVal objMesh, ByVal objPlane)
.
.
.
127 End Function
128
129 Function ClosestNonPolylineVertices (ByVal objMesh, ByVal objPlane)
.
.
.
211 End Function
212
213 Function IsMeshDiagonalized
.
.
.
232 End Function
```

This script uses three functions which you might remember from earlier, the *ClosestPolylineVertices* function, the *ClosestNonPolylineVertices* function and the *IsMeshDiagonalized* function. The content of those functions has been left out in the script above because they have been elaborated in the previous examples. The script above starts by asking the user for an input mesh, some planes (Rhino surface objects) to which the mesh should be aligned and a tolerance value (line 17-19). It then asks the user if the mesh is a normal or a diagonalized mesh using the *IsMeshDiagonalized* function (line 22). We need this information in order to choose the right function for finding the closest set of mesh vertices to a plane (line 36-40). Since the user selects Rhino surfaces from the viewport and we need planes in RhinoScript, we will convert the input surfaces to planes. We will loop through all surfaces, extract their surface points and fit a plane through those points (line 26-30). Now that we have the planes we will loop through them and find the closest set of vertices for each plane. For each plane we will run the *ClosestPolylineVertices* or the *ClosestNonPolylineVertices* function, depending on the outcome of the *IsMeshDiagonalized* function, to get the closest set of vertices to that plane (line 33-41). When we have stored all sets of vertices we will loop once more through all planes to set the corresponding vertices as coplanar to each plane (line 44-46). In the end we optimize our mesh to see the result of the script (line 49). A fair amount of *coplanarity* should be set manually via the *ET* Optimization Importance Options button in the EvoluteTools toolbar in Rhino.

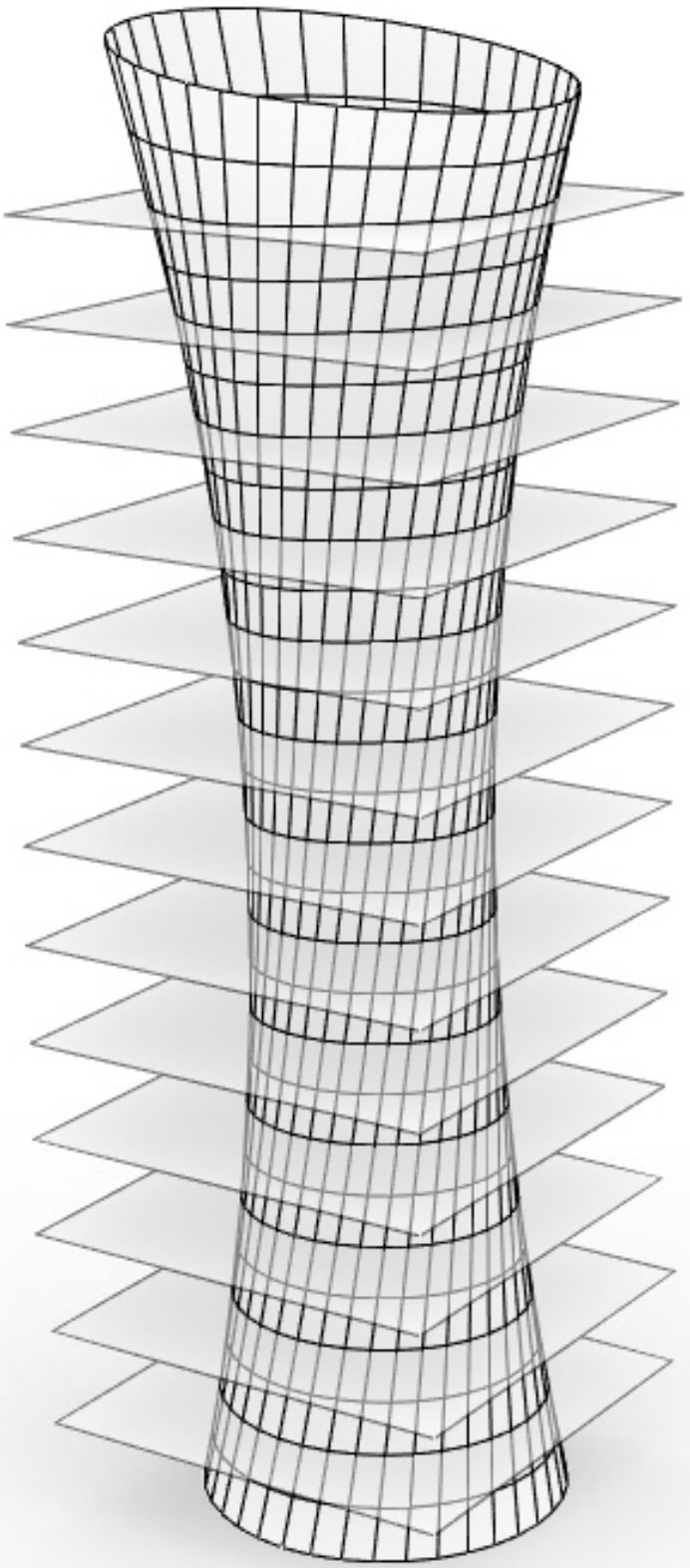
4.4.7 Tower from Coplanarity Constraints

The following script might be one of the longest in this Primer, but it is also very powerful. It creates the mesh layout for a tower in a matter of seconds. The result might look similar to the tower we created in chapter 3.3.2, where we subdivided the mesh to get a desired density and a target edge length. In fact, we will use the same script here to get the initial mesh density in the horizontal direction, since this script here subdivides the mesh only in the vertical direction. So you should start with the provided coarse mesh and run the script from example 3.3.2 on it to get a similar result as below.



This will provide a new coarse mesh for testing this script. When running the script you will be asked to select an edge. Please select one of the small boundary edges at the top or bottom of the mesh. You will see why when we get to the explanation of this script. The new script (the current one) will add loop cuts to the vertical edges and flag the newly created vertices to be coplanar with a horizontal plane on a specified height. This way we can predefine at what height our floor slabs will be and align the mesh vertices to them while we create the mesh. The script will also ask for the number of loop cuts between two floors. This number will usually be around one or two. You can however use this script to experiment with your design and find the optimal solution. After the script finishes, there will be sets of vertices aligned to planes which represent the floors of the tower and one or more loops of vertices between them. Oh, and don't worry about the shapes that your mesh might take on while the script is running. It will turn out just fine. You should just manually optimize the mesh a few times after the script completes, to relax it a little and to let everything fit into its right place.

The image below shows the result of this script. You can see how the loop cuts are aligned with equally spaced planes. The height of the planes is defined by the floor height on line 19. This example shows the mesh with one loop cut between the floors of the tower.



```
1 Option Explicit
2 'Script written by Marko Tomicic
3 'Script copyrighted by Evolute GmbH
4 Call TowerFromCoplanarityConstraints ()
5 ' This script generates a mesh according to user specified constraints.
6 ' This script works with quadrilateral meshes.
7 Sub TowerFromCoplanarityConstraints ()
8     Dim et, objMesh, startingEdge , dblFloorHeight , intBetweenCuts ,
      arrResult
9     Set et = Rhino.GetPlugInObject ("EvoluteTools for Rhino")
10
11    ' Get user input
12    objMesh = Rhino.GetObject("Select the mesh", 32)
13    startingEdge = et.PickEdge(1,1, "Select the edge")
14
15    If Not et.EmIsBoundary (startingEdge ) Then
16        Exit Sub
17    End If
18
19    dblFloorHeight = Rhino.GetReal("Floor height ", 5)
20    intBetweenCuts = Rhino.GetInteger ("How many polylines between two floors?
  ",1)
21
22    ' Find the height of the mesh
23    Dim arrPoints, dblMeshHeight , arrHalfedge , currentEdge
24
25    arrHalfedge = et.EmHalfedge (startingEdge )
26    arrHalfedge = et.EmNextHalfedge (arrHalfedge )
27    currentEdge = et.EmEdge(arrHalfedge )
28
29    arrPoints = et.EmGetEdgePoints (currentEdge )
30    dblMeshHeight = Rhino.Distance(arrPoints(0), arrPoints(1))
31    Rhino.Print "Input mesh height: " & CStr(dblMeshHeight )
32
33    ' Calculate number of floors depending on the object height and the floor
  height
34    Dim intNoOfFloors
35    intNoOfFloors = dblMeshHeight \ dblFloorHeight
36    Rhino.Print "Number of floors: " & CStr(intNoOfFloors )
37
38    If intBetweenCuts <> 0 Then
39        Dim n
40        For n=1 To intBetweenCuts
41            '' Go back to the starting edge
42            arrHalfedge = et.EmHalfedge (startingEdge )
43            arrHalfedge = et.EmNextHalfedge (arrHalfedge )
44            currentEdge = et.EmEdge(arrHalfedge )
45            '' Subdivide
46            Call et.MeshLoopCut (currentEdge )
47            '' Optimize!
48            Call et.Optimize(objMesh, arrResult)
49        Next
50    End If
51
52    ' For every floor create one loop cut
53    ' flag the newly created vertices as coplanar and align to floor height
54    ' add loop-cuts between floors if necessary
55    Dim i, j, k, currentHeight , arrCurrentVertices
56    Dim intNoOfVertices , arrVisitedVertices ()
```

```
57  Dim intNewNoOfVertices , arrNewVisitedVertices ()
58  Dim blnFound , l, m, currentVertex1 , currentVertex2 , arrNewVertices ()
59  currentHeight = 0
60  For i=0 To intNoOfFloors -2
61      '' Go back to the starting edge
62      arrHalfedge = et.EmHalfedge (startingEdge )
63      arrHalfedge = et.EmNextHalfedge (arrHalfedge )
64      currentEdge = et.EmEdge(arrHalfedge )
65      '' Calculate the current height
66      currentHeight = currentHeight +dblFloorHeight
67
68      '' Store all existing vertices in an array
69      ReDim arrVisitedVertices (-1)
70      intNoOfVertices = et.EmNumberOfVertices (objMesh)
71
72      For j=0 To intNoOfVertices -1
73          ReDim Preserve arrVisitedVertices (Ubound(arrVisitedVertices )+
              1)
74          arrVisitedVertices (Ubound(arrVisitedVertices )) = et.EmVertex(
          objMesh, j)
75          Next
76
77          Call et.MeshLoopCut (currentEdge )
78
79          '' Store all existing vertices in an array
80          ReDim arrNewVisitedVertices (-1)
81          intNewNoOfVertices = et.EmNumberOfVertices (objMesh)
82
83          For k=0 To intNewNoOfVertices -1
84              ReDim Preserve arrNewVisitedVertices (Ubound(
          arrNewVisitedVertices )+1)
85              arrNewVisitedVertices (Ubound(arrNewVisitedVertices )) = et.
          EmVertex(objMesh, k)
86          Next
87
88          '' Compare arrays to find newly created vertices
89          ReDim arrNewVertices (-1)
90
91          For l=0 To Ubound(arrNewVisitedVertices )
92              blnFound = False
93              currentVertex1 = arrNewVisitedVertices (l)
94
95              For m=0 To Ubound(arrVisitedVertices )
96                  currentVertex2 = arrVisitedVertices (m)
97                  If currentVertex1 (1) = currentVertex2 (1) Then
98                      blnFound = True
99                  End If
100             Next
101
102             If blnFound = False Then
103                 ReDim Preserve arrNewVertices (Ubound(arrNewVertices )+1)
104                 arrNewVertices (Ubound(arrNewVertices )) = currentVertex1
105             End If
106
107             Next
108
109         '' Flag new vertices as coplanar
110         Dim thePlane : thePlane = Rhino.PlaneFromPoints (Array(0,0,
      currentHeight ), Array(1,0,currentHeight ), Array(0,1,currentHeight ))
111
```

```
112
113             Call et.SetVerticesCoplanar (arrNewVertices , 7, 1, thePlane)
114             Call et.Optimize(objMesh,  arrResult)
115
116             If intBetweenCuts  <> 0 Then
117                 For n=1 To intBetweenCuts
118                     '' Go back to the starting edge arrHalfedge =
119                     et.EmHalfedge (startingEdge ) arrHalfedge  =
120                     et.EmNextHalfedge (arrHalfedge ) currentEdge  =
121                     et.EmEdge(arrHalfedge )
122                     '' Subdivide
123                     Call et.MeshLoopCut (currentEdge )
124                     '' Optimize!
125                     Call et.Optimize(objMesh,  arrResult)
126                 Next
127             End If
128         Next
129
130 End Sub
```

The script starts by asking for the mesh and a starting edge (line 12-13). This edge has to be one of the small boundary edges on either the top or the bottom of the coarse mesh. If no boundary edge is selected the script will abort (line 15-17). The script then asks the user for the floor height and how many loop cuts should be made between two floors (line 19-20). In the next small block we will find the height of the mesh. We find the halfedge of the selected edge, move to the next halfedge and get its edge (25-27). This is the reason why you have to select one of the boundary edges when asked. This way, by going to the next halfedge, we will always come to one of the vertical edges. The vertical edges will change during the script because they will be subdivided numerous times, but the edges at the top and bottom of the mesh will stay untouched. Well, now that we are at one of the long vertical edges of the coarse mesh we can measure its length and therefore the approximate height of the coarse mesh, which is precise enough for this task (line 29-31). The number of floors is calculated by dividing the height of the mesh with the user specified floor height (line 34-36). If the user wants to have loop cuts between the floors we will add the first now (line 38-50). For each loop cut we will first go back to the starting edge (the one the user has selected) and from there advance to one neighboring vertical edge (line 42-44). Now we call a loop cut through the vertical edge and optimize the mesh to relax it (line 46-48). These cuts will be located between the ground and the first floor of our tower. We enter a loop through all floors of the tower except the last (line 60). Remember how we determined the number of floors by dividing the coarse mesh height and the floor height. Well, usually you would loop through all the floors calculated before, but our towers design is angled at the top (see big image) so the last plane would stick out of the reference surface. Every time the loop starts we go back to our starting edge from where we find one neighboring vertical edge (line 62-64). The height of the current floor is calculated on line 66. Now we need to define an empty array in which all currently existing vertices are stored (line 69-75). We will need this array later to find out which are the newly created vertices after we loop cut the mesh. We perform the loop cut on line 77 and again store all currently existing vertices in an array (line 77-86). This second time the array contains the new vertices created by the loop cut as well. We will now compare the two arrays of vertices that we have previously created. First, we start a loop through the array which holds the old as well as the new vertices (line 91). In this loop we take one vertex from the array by its index (line 93) and enter a new loop with that vertex (line 95). The second loop is through all the old vertices that

were there before we performed the loop cut on line 77. The vertex from the first loop is compared to all vertices in the second loop and if a match is found the *blnFound* value is changed to `True` (line 95-100). If the vertex has no match in the *arrVisitedVertices* array it means that it is one of the vertices which have been created by the loop cut. The vertex is added to the array *arrNewVertices* (line 102-105). After all new vertices have been found the plane representing the current floor is created on line 110. We set the vertices to be coplanar to the plane and optimize the mesh to relax it and move the vertices to the right position (line 112-113). If there should be any loop cuts between two floors we will create them now before exiting the loop (line 115-126).

5. Post Processing

5.1 About this chapter

The last chapter of this tutorial is dedicated to post processing the optimized mesh.

Create the reference surface
Create the coarse mesh
Subdivision
Optimization
Post processing

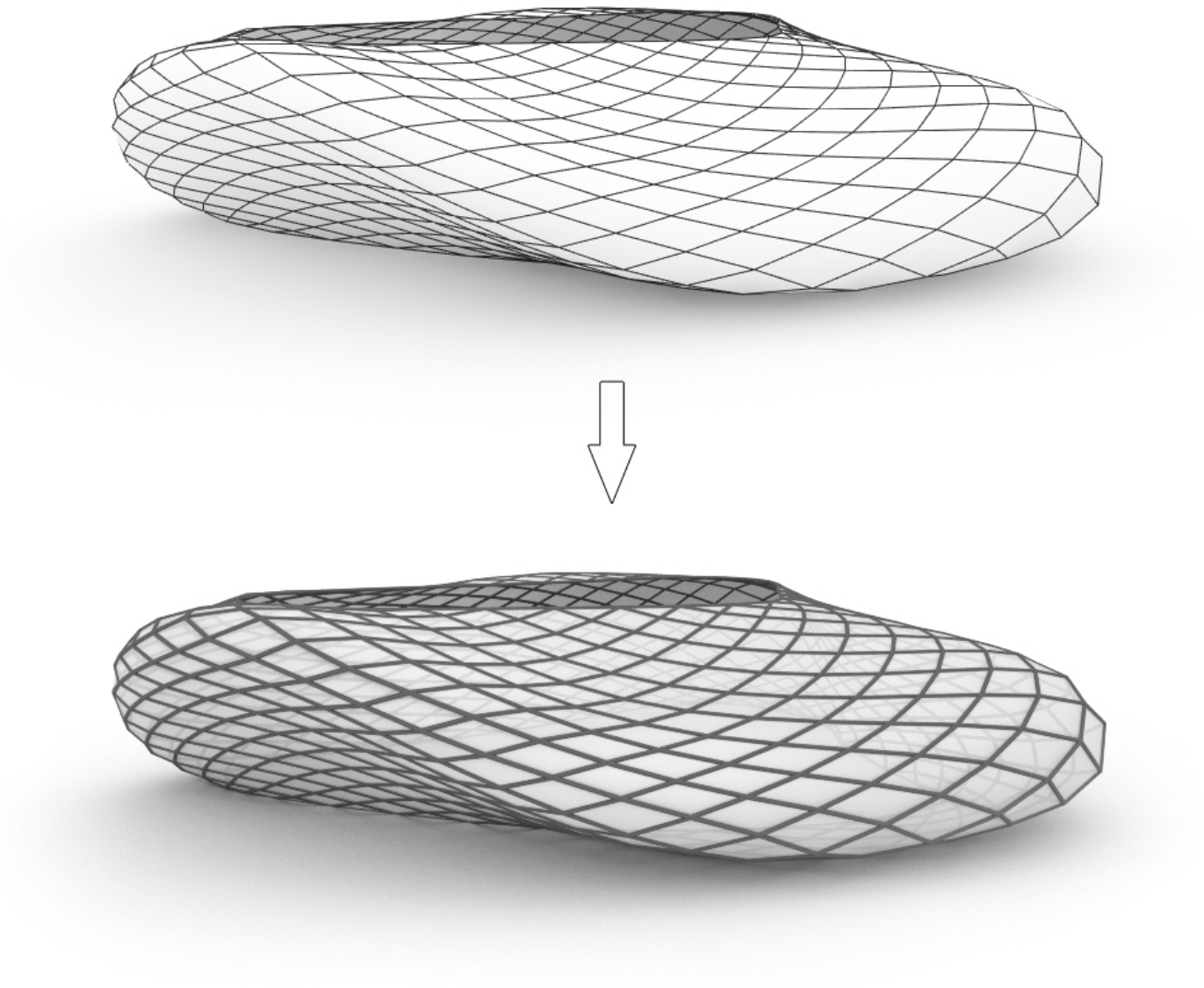
During my research for this tutorial I found out that EvoluteTools not only enables you to optimize your mesh but it takes you a step further. With the EvoluteTools scripting interface you can take the leap from optimized 3D mesh to producing your beams and panels. EvoluteTools for Rhino is by definition a plugin for mesh optimization which means that this is not an actual step in the conventional EvoluteTools workflow but rather a ~~little~~ huge extra that you get for free.

This chapter will teach you to manipulate the optimized mesh and to get the most out of the EvoluteTools scripting interface. Please remember that the example scripts below are there to give you the power to create your own scripts that suit your needs. Now, with great power comes great responsibility. Every freeform project needs an individually tailored solution. It would be impossible to write one script and guarantee that it will work on every project as intended. Because of that, the author cannot be held responsible for anything that might go wrong if you use one of the examples below in your work without checking by yourself whether the script suits your needs or not.

5.2 Examples

5.2.1 Pipe Substructure

In some cases you might want to render an image of the design you have created with EvoluteTools for Rhino. This script will help you to move your project from mesh to physical structure. It takes the mesh as input and creates, in a matter of seconds, a solid, renderable substructure for that mesh.



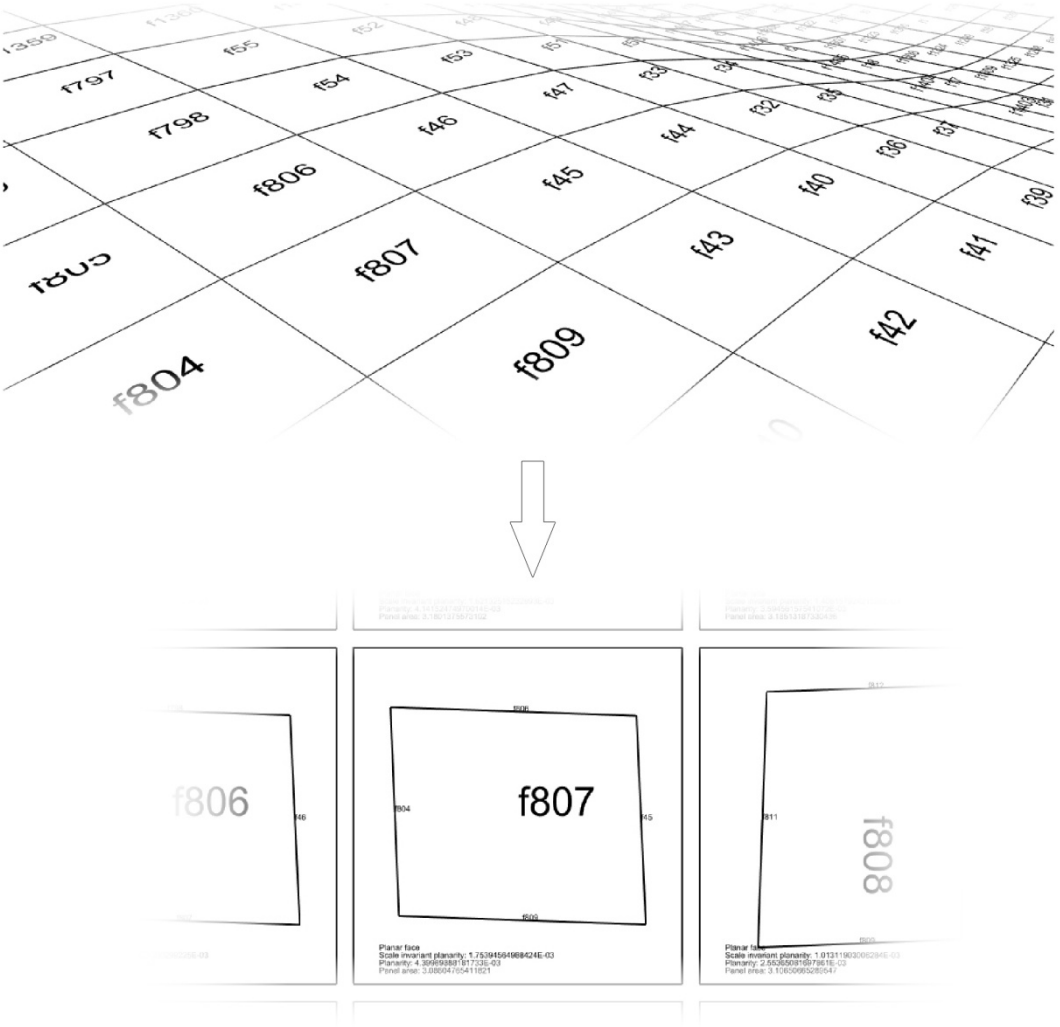
The following script is a really simple example of how to build a substructure for a mesh, thus the outcome of this script is suitable only for rendering purposes and such.

```
Option Explicit
'Script written by Marko Tomicic
'Script copyrighted by Evolute GmbH


Call PipeSubstructure ()
' This script will create a pipe substructure for rendering purposes
Sub PipeSubstructure ()

    Dim et, objMesh, nEdges, i, dblRadiusNormal , dblRadiusDiagonal , dblCircle, dblRadius
    'We access the Scripting interface
    Set et = Rhino.GetPlugInObject ( "EvoluteTools for Rhino" )

    ' Get user input
    objMesh = Rhino.GetObject( "Select Mesh", 32 )
    If IsNull(objMesh) Then Exit Sub
    et.EmGarbageCollection ( objMesh )
    nEdges = et.EmNumberOfEdges ( objMesh )

    ' Find out which is the unit system used and propose a standard pipe radius
    If Rhino.UnitSystem = 2 Then 'millimeters
        dblRadiusNormal = 200
    ElseIf Rhino.UnitSystem = 3 Then 'centimeters
        dblRadiusNormal = 20
    ElseIf Rhino.UnitSystem = 4 Then 'meters
        dblRadiusNormal = 0.2
    Else
        dblRadiusNormal = 0.2
    End If

    dblRadiusNormal = Rhino.GetReal( "Enter radius of the pipes", dblRadiusNormal , 0 )

    Rhino.EnableRedraw (False)
    ReDim arrVertices (-1)
    ReDim arrNormals (-1)
    ReDim arrFaces(-1)

    Dim arrStartPoint , arrNormal, arrBasePlane , pipe, arrEndPoint , rail

    ' Pipe the edges
    For i = 0 To nEdges-1
        Dim arrEdge : arrEdge = et.EmEdge( objMesh, et.EmEdge( i ) )
        Dim arrHalfedge : arrHalfedge = et.EmHalfedge ( arrEdge, 0 )

        '' Creating a line to use as rail for the sweep1 command
        '' this line replaces the actual edge
        arrStartPoint = et.EmStartVertex ( arrHalfedge )
        arrStartPoint = et.empoint (arrStartPoint )
        arrEndPoint = et.EmEndVertex ( arrHalfedge )
        arrEndPoint = et.EmPoint (arrEndPoint )
        '' Creating the normal for the plane orientation
        arrNormal = et.EmHalfedgeVector ( arrHalfedge )

        '' Creating the base plane for the circle which will be sweeped
        arrBasePlane = Rhino.PlaneFromNormal (arrStartPoint , arrNormal)

        Dim dblSize
        dblRadius = dblRadiusNormal

        '' Pipe the edge
        rail = Rhino.AddLine (arrStartPoint , arrEndPoint )
        dblCircle = Rhino.AddCircle (arrBasePlane , dblRadius)
        pipe = Rhino.AddSweep1 (rail, dblCircle)

        Next
    Rhino.EnableRedraw (True)
End Sub
```

First the script asks the user for an input mesh and counts the edges of that mesh (line 15-18). It then determines which unit system is used in the current file. We will use this information to choose which standard value to offer the user when we ask him/her for the radius of the pipes (line 21-29). Now we ask the user for the pipe radius (line 31). We enter a loop through all mesh edges in which the edges will be piped one by one on line 41. In order to create the pipe around the edge we will need a circle and a rail (the edge) to sweep the circle over. We will first extract the start and the end point of the edge (line 47-50). We will use those points to create the rail for the sweep command. The circle that we want to sweep along the edge should be perpendicular to the edge and located with its center at the start point of that edge. First we extract the halfedge vector (line 52). This will serve to orient the circle perpendicular to the edge. We can create a circle from a base plane and its radius. The base plane is created from an origin point (start point of our edge) and a normal (the edge's direction) on line 55. Now that we have the endpoints of the edge, we create a line between them to serve as rail for the sweep command (line 61). The circle is created in its right position and orientation using the base plane that we created earlier (line 62). We can now add the pipe by sweeping the circle along the line/edge (line 63).

5.2.2 The Ultimate Panel Layout for Production Script

The following script will take your finished and optimized mesh from digital concept to production. All mesh faces, regardless of the size of the mesh or the number of faces, will be taken out of the mesh, named and laid out for production. We will give each panel a unique name with the index “f” for face and a unique number. The face will be copied to its right place in a grid on the World XY plane. We will add the names of the neighboring faces to each edge of every face, so you know how to put them back together after they have been manufactured.



We will measure the scale invariant planarity of each face. The user will specify his/her planarity tolerance and if the planarity of a face is within that tolerance the face’s edges will be projected to the ground and the face deleted. This will leave you with planar curves which should be usable as input for almost every cutting machine. If the face cannot be regarded as planar and thus cannot be manufactured from a planar material the script will return it as a 3d surface which is usable in other more complicated production processes. The panels will further be sorted into two layers, one (blue) for planar panels and the other (red) for non planar panels. Along with each panel there will be some information (scale invariant planarity, planarity, panel area …) printed in the lower left corner of its frame. You will be able by now to change the information which is displayed here on your own by editing the script a little if you need to. After the script is completed you will have the input mesh with a name tag on each mesh face and a grid of the same faces laid out on the XY plane, named, described and sorted by planarity.

Because of its length, the script will be located after the explanation text.

The script starts with the dimensioning of a lot of variables which we will discuss as we stumble upon them. First the script asks for the user input. The user is asked to select a mesh, to tell the desired tolerance for the scale invariant planarity and to decide how many panels should be laid out in one row (line 14-17). The next thing for us to do is to create a grid on which the panels will be laid out. Through playing with these options I found that it is a good way to find the longest edge of the mesh and set the grid spacing to be 1.2 times that value. The text height for the main text or the face name is set to 1/10 of the length of the longest mesh edge and the other text to 1/40 of the same value (line 19-23). You can change these values to increase/decrease text sizes or to increase/decrease the grid spacing. The longest mesh edge is found using the *ExtremeEdges* function (line 215-257) which was introduced in chapter 3.3.1 – Edge Length Analysis.

The next thing we need is a big surface on the ground on which we can project all of our planar panels. The surface's length in X direction will be the number of elements in a row times the grid spacing plus one length of the grid spacing so there is some extra space (line 31). For the surface's length in Y direction we first need to calculate the number of elements in a column by dividing the number of mesh faces with the number of faces in one row. The result is then multiplied with the grid spacing and one grid spacing length is added for more comfort (line 32). Now that we have the length and the width of the surface we go on to create it and move it into place (line 33-35). We are ready now to start looping through our faces (line 44-198). We first get a face from the mesh by its index. Then the script extracts the face's normal and reverses it (if everything turns out upside down comment this step out). We copy the face by extracting its face points and creating a new surface through those points (line 41-45). We find out the value of the face planarity scale invariant function (line 48) and compare it to the user specified tolerance (line 49-52). We will need the face's approximate midpoint and orientation to place the face's name in the middle. A new plane is created for this purpose on line 59. The plane provides us with an origin which is close to the face's midpoint and with a normal vector for orientation. On line 58 we created the plane which is used to transform the face from its position in the mesh to its position in the layout grid (line 58). There is no way known to man to fit a plane exactly through four points if they are not coplanar. You can't always get what you want, Jagger might say. The best way to create this plane for a quadrilateral face is to try and fit a plane through the face's points as close as possible. This method will never give you a plane which fits 100% through the face points, unless the face is planar, but this is the best we can get in this universe. Note that if you plan to build your design, you have to count on these deviations and adjust the script you work with according to your needs in terms of form, manufacturing process and materials used. If you are not careful enough your panels might not fit together in the end on the building. After we created our plane we need to check whether it is oriented properly. We do this by measuring the angle between the plane's normal and the face's normal. It should not be larger than a few degrees if they are both oriented in the same direction. If the angle is larger than 10 degrees the new face should definitely be flipped (line 61-65). We now add the text to our face while it is still in its original position in the mesh (line 69-70). For the text on the face edges we need to find the insertion point or edge midpoint (line 83) and the names of the neighboring faces (line 78). In each edge's midpoint we need first to create a base plane (line 85) on which we will create our text (line 86). Now the copy of the face with all the text objects that we created previously needs to be moved from its original position in the

mesh to a new position in the grid and all those objects have to be aligned with the World XY plane. We will first transform all objects to the world origin and then move it into position. For the transformation we need a transformation matrix (line 91). The matrix basically stores all the information that the computer needs to know in order to align the x, y and z coordinates of one plane to the x, y and z coordinates of another plane. The function first takes the target plane as input and then the plane in space. Now all objects are moved and aligned to the world origin one by one (line 92-100). We need to loop through the edge text objects and to move them into position one by one too (line 102-106).

Now that we transformed the faces and the text objects on them from the mesh to the world origin we can say that the hardest part of this script is behind us. What's left is to add a frame with some descriptions around the face and to move all together to the right position in the grid (and eventually to project the planar faces onto the ground surface that we created on line 33. First let's create the frame. The frame is going to be a polyline so we need four corner points to create it. Actually, it needs five corner points to create a closed polyline but the first and last point is the same (line 111-118). Now we will add some description text to the lower left corner of the frame. I've created four lines of text here (line 120-132). The right spacing of the text and positioning of the same is a matter of trial and error (or precise calculation, whichever you like best).

The next thing on our checklist is to move the face, the frame and all descriptions to the right position in our layout grid. Each time when the current index "i" divided by the number of rows gives no remainder and the index is not zero the Y value is set to *Y – dblGridSpacing* and X is reset to zero. This means that the current position in the grid is reset to first position on the x axis but it is shifted on row down on the y axis. Now for every face the X value is increased by *dblGridSpacing* which means that we move to the next position in relation to the place where we laid down the last plane. Using the newly calculated X and Y value we create a translation array from the world origin to that point (line137-143). We use that translation array to move all our objects one by one to their right position (line 146-153). The only thing that's left to do is to sort the panels according to their planarity into layers. If the panel is planar as decided on line 51 we proceed to check if there is already a layer for planar panels. If there is no such layer we create it (line 159-162). We go on to project the planar face onto the ground surface and delete the original surface (line 165-169). Next thing to do is to move all elements belonging to that face (projected lines, frame, descriptions…) onto the *Planar Faces* layer (line 172-179).

If the current panel cannot be regarded as planar then we create the *Curved Faces* layer if there isn't one already in the scene (line 183-186). Just as before, all elements belonging to the current face are moved onto that layer (line 189-201). The last step is to delete the ground surface which we used to project the planar faces onto (line 203).

Note: If the text on the mesh faces is upside down after the script completes it means that the mesh is not properly aligned and that the result is **not** correct. If this happens you can try to weld the mesh and run the script again. If it is still not fixed, go to the line 43 and enable the vector reversion which has been commented out. Please always check if the text on the mesh and on the extracted faces is properly aligned.

```
Option Explicit
'Script written by Marko Tomicic
'Script copyrighted by Evolute GmbH

Call PanelLayout()
' This script will lay out the mesh panels for production and label them according
' to their position in the structure. This script works with triangular and
  quadrilateral meshes.
Sub PanelLayout()
    Dim et, objMesh, intNoOfFaces , dblTolerance , dblMainTextHeight ,
    dblOtherTextHeight
    Dim intRow, dblLongestEdge , dblGridSpacing , arrTheSurface , i, j, k, currentFace
    Dim arrTextOnEdge (), GroundText (), X, Y, arrRectPoints (),ShortestEdge, LongestEdge
    Set et = Rhino.GetPlugInObject ("EvoluteTools for Rhino")
    ' Get inputs
    objMesh = Rhino.GetObject("Select mesh", 32)
    intNoOfFaces = et.EmNumberOfFaces (objMesh)
    dblTolerance = Rhino.GetReal("Planarity tolerance: ", 0.015)
    intRow = Rhino.GetInteger ("How many panels in a row?", 25)

    '' Decide on the grid spacing and font size
    dblLongestEdge = ExtremeEdges(objMesh, ShortestEdge, LongestEdge)
    dblGridSpacing = 1.2*dblLongestEdge
    dblMainTextHeight = dblLongestEdge /10
    dblOtherTextHeight = dblLongestEdge /40

    X=0
    Y=-dblGridSpacing

    Rhino.EnableRedraw (False)

    '' Create the projection surface for the planar panels
    Dim TheSurfaceX : TheSurfaceX = dblGridSpacing *intRow+dblGridSpacing
    Dim TheSurfaceY : TheSurfaceY = intNoOfFaces /intRow*dblGridSpacing +
    dblGridSpacing
    arrTheSurface = Rhino.AddPlaneSurface (Rhino.WorldXYPlane ,
    TheSurfaceX , TheSurfaceY )
    Dim arrTranslationVector : arrTranslationVector = Array(0, -
    TheSurfaceY , 0)
    arrTheSurface = Rhino.MoveObject (arrTheSurface , arrTranslationVector )

    ' Add name tag and plane to every face
    ' Copy the face with the info to xy plane
    For i=0 To intNoOfFaces -1
        '' Current Face
        currentFace = et.EmFace(objMesh, i)
        Dim Normal : Normal = et.EmNormal(currentFace )
        'Normal = Rhino.VectorReverse (Normal)
        Dim Points : Points = et.EmGetFacePoints (currentFace )
        Dim FaceCopy : FaceCopy = Rhino.AddSrfPt (Points)

        '' Face's planarity
        Dim dblPlanarity : dblPlanarity = et.FacePlanarityScaleInvariant (
      currentFace )
        Dim blnPlanar : blnPlanar = False
        If dblPlanarity <= dblTolerance Then
            blnPlanar = True
        End If

        '' Find aproximate face midpoint
```

```
55              Dim arrFaceMidpoint  : arrFaceMidpoint  = MeanVector(Points)
56
57              '' Create a plane through the face points
58              Dim arrFacePlane  : arrFacePlane  = Rhino.PlaneFitFromPoints (
    Points)
59              Dim arrFacePlaneText : arrFacePlaneText =
                Rhino.PlaneFromNormal(MeanVector(Points), Normal)
60              '' If the plane is upside down
61              If Rhino.VectorAngle  (arrFacePlane (3), Normal) > 10 Then
62                  Dim newNormal  : newNormal  = Rhino.VectorReverse (arrFacePlane (
      3))
63                  Dim newFacePlane  : newFacePlane  = Rhino.PlaneFromNormal (
                        arrFacePlane (0), newNormal)
64              arrFacePlane  = newFacePlane
65          End If
66
67          '' Add text to the faces
68          Dim Text  : Text  = CStr(currentFace (1))
69          Dim TextFaces  : TextFaces  = Rhino.AddText  (Text, arrFacePlaneText
            ,dblMainTextHeight )
70
71
72          '' Add text to the edges
73          ReDim arrTextOnEdge (-1)
74          Dim arrHalfedges  : arrHalfedges  = et.EmGetFaceHalfedges (
            currentFace )
75          For j=0 To Ubound(arrHalfedges )
76              ReDim Preserve  arrTextOnEdge (Ubound(arrTextOnEdge )+1)
77              Dim Halfedge  : Halfedge  = arrHalfedges (j)
78              Dim arrOppositeFace : arrOppositeFace  = Et.EmFace(et.
                    EmOppositeHalfedge (Halfedge))
79              Dim arrEdgeText  : arrEdgeText  = CStr(arrOppositeFace (1))
80              Dim Point1  : Point1  = et.EmPoint(et.EmStartVertex  (Halfedge))
81              Dim Point2  : Point2  = et.EmPoint(et.EmEndVertex  (Halfedge))
82              Dim LineEdge  : LineEdge  = Rhino.AddLine(Point1,  Point2)
83              Dim MidpointEdge  : MidpointEdge  = Rhino.DivideCurve (LineEdge,
                    2)
84              Call Rhino.DeleteObject (LineEdge)
85              Dim arrEdgePlane  : arrEdgePlane  = Rhino.PlaneFromNormal (
                MidpointEdge (1), Normal)
86              arrTextOnEdge (Ubound(arrTextOnEdge )) = Rhino.AddText(
                  arrEdgeText ,  arrEdgePlane , dblOtherTextHeight )
87          Next
88
89
90          '' Transform to xy plane
91          Dim arrMatrix  : arrMatrix  = Rhino.XformChangeBasis  (Rhino.
            WorldXYPlane , arrFacePlane )
92          Dim FaceOnGround  : FaceOnGround  = Rhino.TransformObject (FaceCopy,
            arrMatrix)
93          Dim MainTextOnGround  : MainTextOnGround  = Rhino.TransformObject (
            TextFaces, arrMatrix, True)
94          ReDim GroundText(Ubound(arrTextOnEdge ))
95
96          '' Move the face and the text to the world origin
97          Dim FaceOnGroundPoints  : FaceOnGroundPoints  = Rhino.
            SurfacePoints (FaceOnGround )
98          Dim FaceOnGroundMidpoint  : FaceOnGroundMidpoint  = MeanVector(
            FaceOnGroundPoints )
99          FaceOnGround  = Rhino.MoveObject (FaceOnGround ,
            FaceOnGroundMidpoint , Array(0,0,0))
```

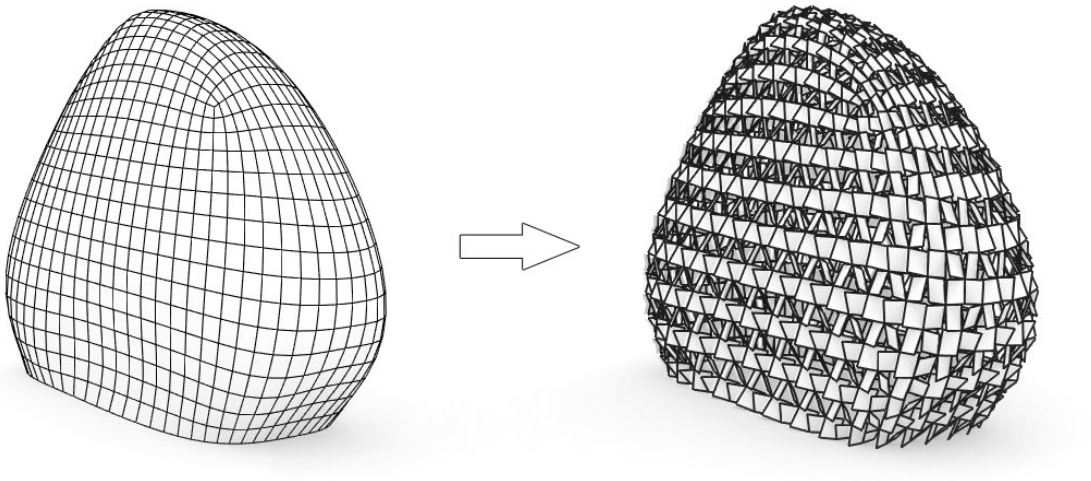
```
100             MainTextOnGround  = Rhino.MoveObject (MainTextOnGround ,
          FaceOnGroundMidpoint , Array(0,0,0))
101
102             For k=0 To Ubound(arrTextOnEdge )
103                 Dim EdgeTextOnGround  : EdgeTextOnGround  = Rhino.
            TransformObject (arrTextOnEdge (k), arrMatrix )
104                 EdgeTextOnGround  = Rhino.MoveObject (EdgeTextOnGround ,
            FaceOnGroundMidpoint , Array(0,0,0))
105                 GroundText(k) = EdgeTextOnGround
106             Next
107
108             ' Add descriptions
109             '' The frame rectangle
110             Dim arrRectangle , curvature, dblUnit, Text1, Text2, Text3, Text4,
            arrSrfArea
111             ReDim arrRectPoints (4)
112             dblUnit  = dblGridSpacing /2*0.95
113             arrRectPoints(0) = Array(dblUnit,dblUnit,0)
114             arrRectPoints (1) = Array(-dblUnit,dblUnit,0)
115             arrRectPoints (2) = Array(-dblUnit,-dblUnit,0)
116             arrRectPoints (3) = Array(dblUnit,-dblUnit,0)
117             arrRectPoints (4) = Array(dblUnit,dblUnit,0)
118             arrRectangle  = Rhino.AddPolyline (arrRectPoints )
119
120             '' Create the description text
121             Text1 = Rhino.AddText(("Scale invariant planarity: " & CStr(
        dblPlanarity )), Array(-dblUnit+0.2,-dblUnit+0.2,0), dblOtherTextHeight ) If
122             blnPlanar  = True Then
123                 Text2 = "Planar face"
124             Else
125                 Text2 = "Curved face"
126             End If
127             Text2 = Rhino.AddText(Text2, Array(-dblUnit+0.2,-dblUnit+0.2+1.3*
          dblOtherTextHeight ,0), dblOtherTextHeight
128             ) Text3 = et.FacePlanarity (currentFace )
129             Text3 = Rhino.AddText(("Planarity:  " & CStr(Text3)), Array(-
        dblUnit+0.2,-dblUnit+0.2-1.3*dblOtherTextHeight ,0), dblOtherTextHeight )
130             arrSrfArea  = Rhino.SurfaceArea (FaceOnGround )
131             Text4 = arrSrfArea (0)
132             Text4 = Rhino.AddText(("Panel area: " & Text4), Array(-dblUnit+0.
                2,-dblUnit+0.2-2.6*dblOtherTextHeight ,0), dblOtherTextHeight )
133
134
135             ' Move objects to specified location
136             '' Layout
137             Dim translation
138             If i Mod intRow = 0 And i><0 Then
139                 Y = Y-dblGridSpacing
140                 X = 0
141             End If
142             X = X+dblGridSpacing
143             translation  = Array(X,Y,0)
144
145             '' Translation
146             Rhino.MoveObject  FaceOnGround , translation
147             Rhino.MoveObject  MainTextOnGround , translation
148             Rhino.MoveObjects  GroundText, translation
149             Rhino.MoveObject  arrRectangle , translation
150             Rhino.MoveObject  Text1, translation
151             Rhino.MoveObject  Text2, translation
```

```
152             Rhino.MoveObject  Text3, translation
153             Rhino.MoveObject  Text4, translation
154
155             ' Sort objects into layers
156             If blnPlanar = True Then
157
158                 '' Create layer for the planar panels if there isn't one
        already
159                 If Not Rhino.IsLayer("Planar  Faces") Then
160                     Rhino.AddLayer("Planar  Faces") Rhino.LayerColor
161                     "Planar Faces", RGB(0,0,255)
162                 End If
163
164                 '' Project edges of the planar panels to the surface
165                 Dim arrFaceEdges : arrFaceEdges = Rhino.DuplicateEdgeCurves (
         FaceOnGround )
166                 Rhino.DeleteObject (FaceOnGround )
167                 arrFaceEdges = Rhino.JoinCurves (arrFaceEdges , True)
168                 Dim arrProjectedEdges : arrProjectedEdges = Rhino.
       ProjectCurveToSurface (arrFaceEdges , arrTheSurface , Array(0,0,1))
169                 Rhino.DeleteObjects  arrFaceEdges
170
171                 '' Move objects to their layer
172                 Rhino.ObjectLayer  arrProjectedEdges , "Planar Faces"
173                 Rhino.ObjectLayer  MainTextOnGround , "Planar Faces"
174                 Rhino.ObjectLayer  GroundText, "Planar Faces"
175                 Rhino.ObjectLayer  Text1, "Planar Faces"
176                 Rhino.ObjectLayer  Text2, "Planar Faces"
177                 Rhino.ObjectLayer  Text3, "Planar Faces"
178                 Rhino.ObjectLayer  Text4, "Planar Faces"
179                 Rhino.ObjectLayer  arrRectangle , "Planar Faces"
180
181             Else
182                 '' Create layer for the planar panels if there isn't one
            already
183                 If Not Rhino.IsLayer("Curved  Faces") Then
184                     Rhino.AddLayer("Curved  Faces") Rhino.LayerColor
185                     "Curved Faces", RGB(255,0,0)
186                 End If
187
188                 '' Move objects to their layer Rhino.ObjectLayer
189                 FaceOnGround , "Curved Faces" Rhino.ObjectLayer
190                 MainTextOnGround , "Curved Faces" Rhino.ObjectLayer
191                 GroundText, "Curved Faces" Rhino.ObjectLayer Text1,
192                 "Curved Faces" Rhino.ObjectLayer  Text2, "Curved
193                 Faces" Rhino.ObjectLayer  Text3, "Curved Faces"
194                 Rhino.ObjectLayer  Text4, "Curved Faces"
195                 Rhino.ObjectLayer  arrRectangle , "Curved Faces"
196             End If
197         Next
198
199         Rhino.EnableRedraw (True)
200
201         ' Delete the projection surface
202         Rhino.DeleteObject  arrTheSurface
203
204 End Sub
205
206 Function  MeanVector ( ByVal arrPoints  )
207         Dim m : m = Array(0,0,0)
```

```
208         Dim p
209         For Each p In arrPoints
210             m = Rhino.VectorAdd(m,p)
211         Next
212         MeanVector  = Rhino.VectorScale (m,1/(UBound(arrPoints)+1))
213 End Function
214
215 Function  ExtremeEdges  (ByVal  TheMesh, ByRef ShortestEdge, ByRef LongestEdge)
.
.
.
218 End Function
```

5.2.3 FaceRotation

This is the last example in this Primer. You might remember it from earlier. It was also the first script that we executed. We have come a long way since then. I hope it wasn't too hard and that you have learned a lot.



Now that you have come so far you might take it as an insult if I tried to explain this easy little piece of code to you. That's why I won't do it. Your final exercise is to analyze it by yourself and maybe even improve it?! Good luck!

```
Option Explicit
'Script written by Marko Tomicic
'Script copyrighted by Evolute GmbH

Call FaceOpeningRotation ()
' This script rotates the faces of a mesh by a specified rotation angle
Sub FaceOpeningRotation ()

    On Error Resume Next

    Dim et, arrHalfedge , i, arrVertex, axis, midpoint, point, dblAngle,
    objMesh, nFaces
    Set et = Rhino.GetPlugInObject ("EvoluteTools for Rhino")

    ' Get user input
    objMesh = Rhino.GetObject("Select Mesh", 32)
    If IsNull(objMesh) Then Exit Sub
    et.EmGarbageCollection (objMesh)
    dblAngle = Rhino.GetReal("Opening Angle", 30)

    ' Find number of mesh faces
    nFaces = et.EmNumberOfFaces (objMesh)

    Call Rhino.EnableRedraw (False)

    For i=0 To nFaces - 1

        ' Get the face, find face points
        Dim arrface : arrface = et.EmFace(objMesh, et.EmFace(i))
        Dim arrPointsAxis : arrPointsAxis = et.EmGetFacePoints (arrFace)

        ' Create rotation axis
        axis = Rhino.AddLine(arrPointsAxis (0), arrPointsAxis (2))
        midpoint = Rhino.DivideCurve (axis, 2, True)

        ' Create new face
```

```
36          Dim arrFacePoints  : arrFacePoints  = et.EmGetFacePoints (arrFace)
37          Dim arrNewFace  : arrNewFace  = Rhino.AddSrfPt(arrFacePoints )
38
39          ' Rotate face
40          Dim arrVectorAxis  : arrVectorAxis  = Rhino.VectorCreate (arrPointsAxis (
        0), arrPointsAxis (2))
41          Dim arrNewFace2  : arrNewFace2  = Rhino.RotateObject (arrNewFace,
            midpoint(1), 30, arrVectorAxis )
42
43      Next
44      Call Rhino.EnableRedraw (True)
45
46  End Sub
```

Acknowledgements

The development of EvoluteTools was partially supported by the European Community's 7th Framework "People" Program through an IAPP project under grant agreement 230520 (ARC).

Thanks for reading and enjoy using EvoluteTools !

Revision 1.0

19th of November 2012