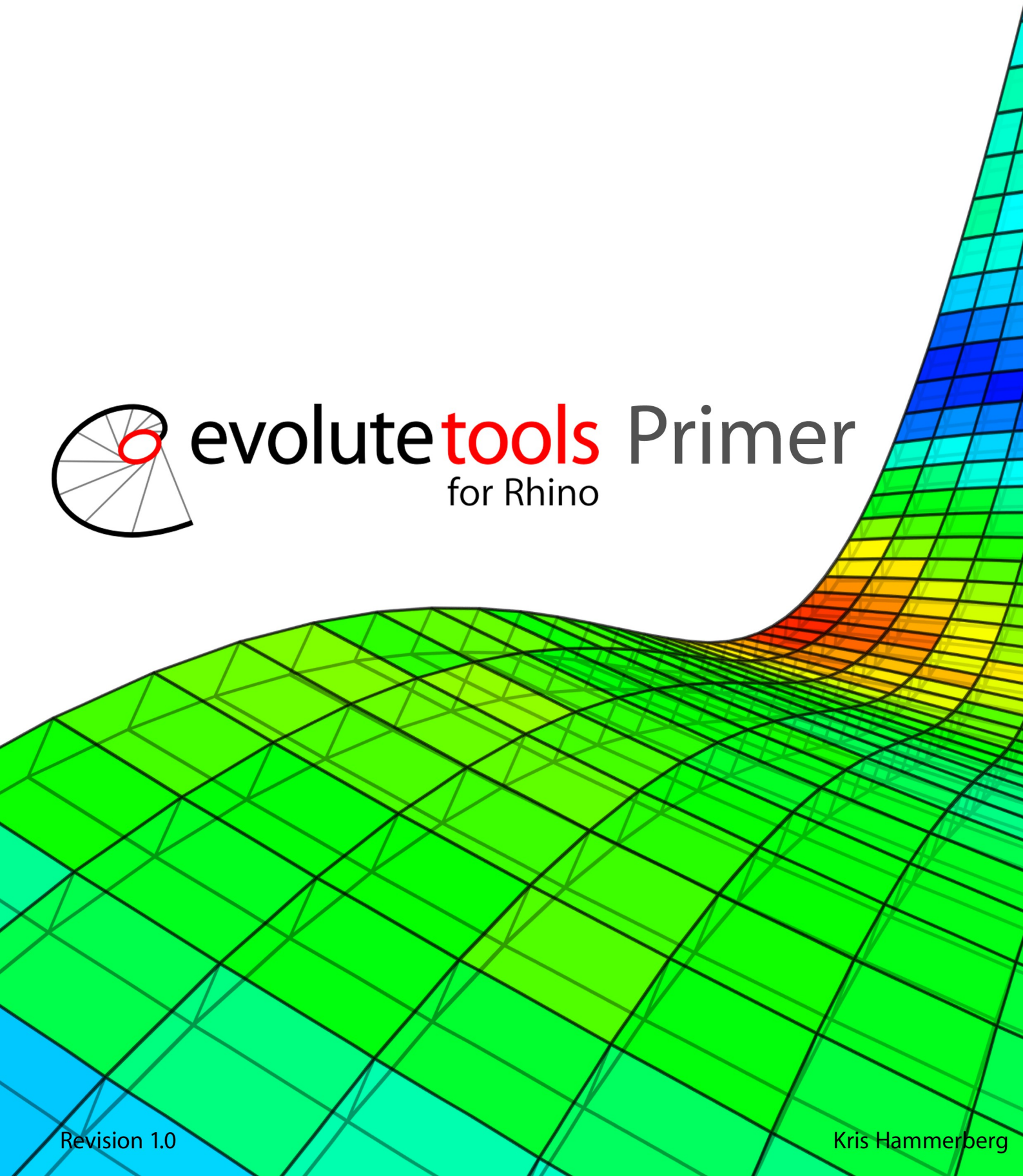
evolutetools Primer

for Rhino



Revision 1.0

Kris Hammerberg

Contents

1. Introduction : The EvoluteTools Primer

You are holding in your hands (or much more likely, viewing on your screen) the EvoluteTools Primer. This is intended to be the ultimate starting point for new users of EvoluteTools, as well as a reference for more experienced users. In the following chapters, we will examine the various functions of EvoluteTools along with examples to illustrate how these features can be used in practice. Additionally, there is a collection of hints, tricks, and tips that will help you get the most out of EvoluteTools. So whether this is your first step or your hundredth, we hope you enjoy this primer!

Before we head down the rabbit hole, I'd like to give you one quick tip on how to read this primer. We wanted to provide a document that would function both as a starting guide that could be read cover to cover and a (more or less) complete reference of EvoluteTools functionality. Fortunately for you, but unfortunately for this author, EvoluteTools is really packed with features! As a result, parts of this primer read like a list of functions and lose the narrative thread. To keep things exciting and manageable for first time readers, I've marked optional chapters and subchapters with this symbol: **«»**. When you see this you'll know that the content isn't really relevant for a first read through. It might be a rarely used feature or it may be a little too technical. Luckily it'll still be there when you need it later.

2. The Motive

No one understands the challenges, compromises, and sacrifice of bringing an idea into the world of material things like those involved in the design and construction of buildings. There is, as of this writing, no single, seamless, durable material that can take any form and satisfy all the requirements of building's outer shell. Buildings are constructed in layers of individual pieces (tiles, panels, shingles, vapor barriers, etc.). For forms that are composed of flat surfaces, it is a fairly easy task to break these surfaces into smaller components manufactured from flat materials.

However, once a designer moves away from Flatland into the realm of the Curve, this problem becomes exponentially more difficult. One of the most common solutions for the construction of free-form structures is to approximate the shape with a collection of panels attached to a structural support grid. This is easy enough to say, but how do you go about dividing up a complex shape? How do you convert a freeform design like the one shown below into a real building?

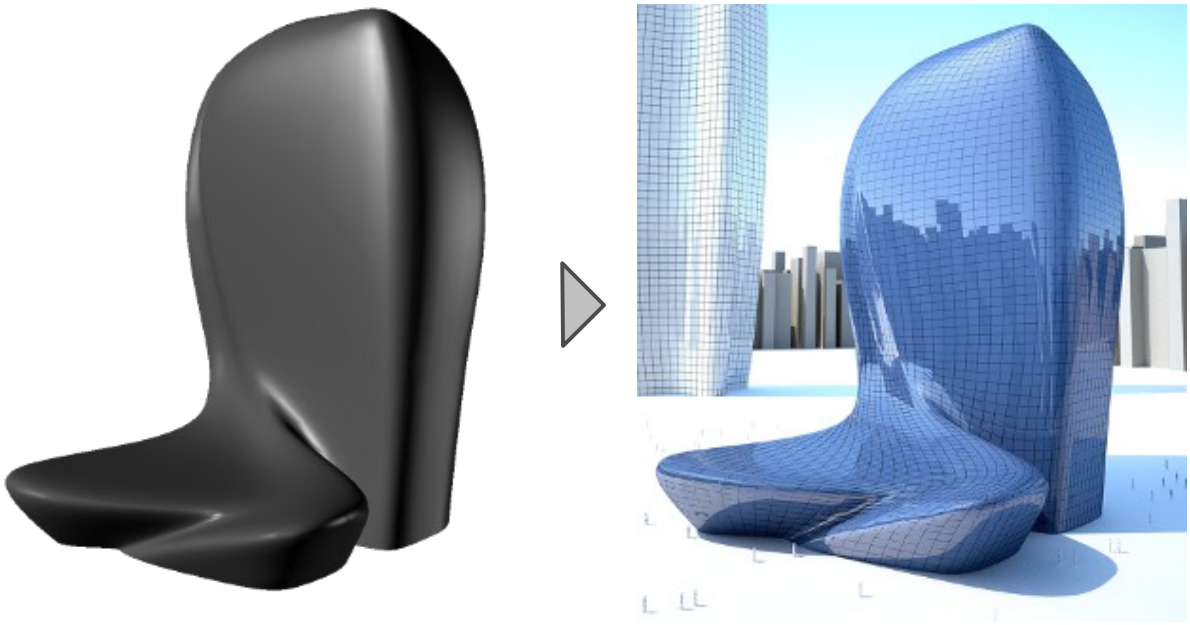


Fig 1. *– EvoluteTools makes the task of panelizing a complex geometry like this the work of minutes rather than days.*

EvoluteTools is a plug-in for Rhino that allows you to create a panel layout across any freeform surface according to your project-specific constraints and you can do it in just minutes. It is a powerful tool that can turn beautifully complex forms into beautifully buildable projects. Need to cover a surface in rectangular panels? No problem. Need them to be planar? Easy. Want to see what it will look like with a hexagonal panel layout? These solutions are just a few clicks away.

Whether you are an architect, contractor, designer, engineer, or manufacturer, EvoluteTools will **decrease cost, save time, lower risk, and improve aesthetics!**

However, the potential of EvoluteTools goes well beyond just panel layouts on abstract surfaces without any thickness. It provides you with the tools to rationalize actual structures composed of multiple layers and material depth. Using EvoluteTools mesh optimization in combination with its mesh offset command and RhinoScript

interface, it is possible to go much deeper. Below are images of the KREOD pavilion by Pavilion Architecture in partnership with Evolute. As part of the design team, Evolute was responsible for rationalizing KREOD's complex input surface, designing the panel layout, the parametric detailing of the wooden members and outputting the production geometry for fabrication. The original surface was analyzed, paneled and optimized using EvoluteTools and the techniques you'll learn in this primer. The result is a very regular and good looking hex mesh.

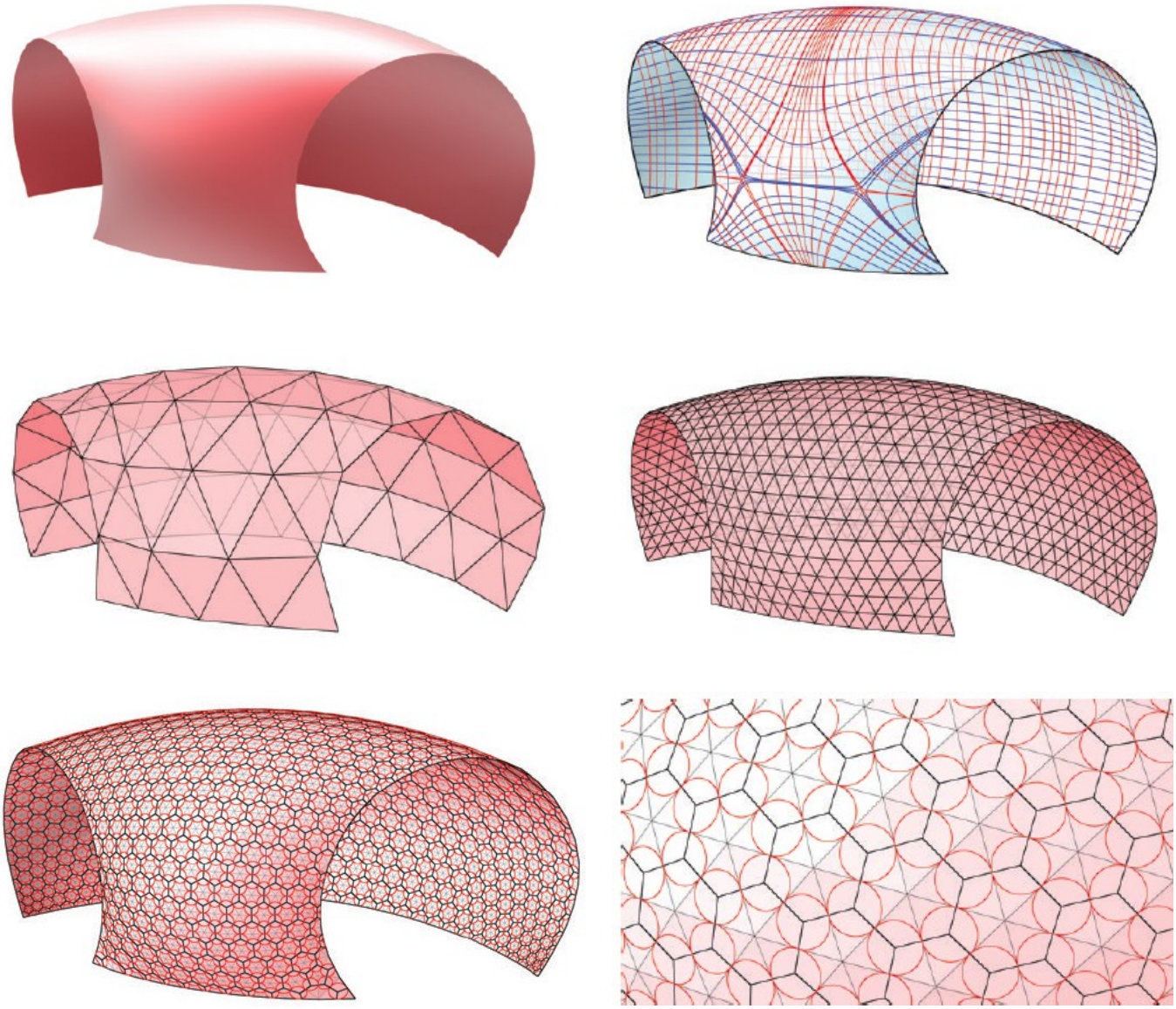


Fig. 2 – Top Left: *Original input surface from designer.* ***Top Right:*** *Analysis of principal curvature lines.* ***Middle Left:*** *Coarse panel layout.* ***Middle Right:*** *Panel refinement.* ***Lower Left:*** *Circle-packing optimization and the resulting hexagonal panel layout.* ***Lower Right:*** *Close-up of the circle-packing optimized panel layout, revealing the relation between the triangle and hexagonal layout.*

It is a nice mesh, but it is a far cry from a buildable structure. In consultation with the project's engineering team, Ramboll Engineering, a reciprocal construction detail was devised to connect the beams. [see images below] Based on this detail design, Evolute coded the generation of the member layout in tune with all the aesthetic and manufacturing constraints. Using EvoluteTools PRO and it's powerful RhinoScript interface, around one thousand members were parametrically generated, along with their labeling. The exported data was sent **directly** to CNC fabrication equipment.

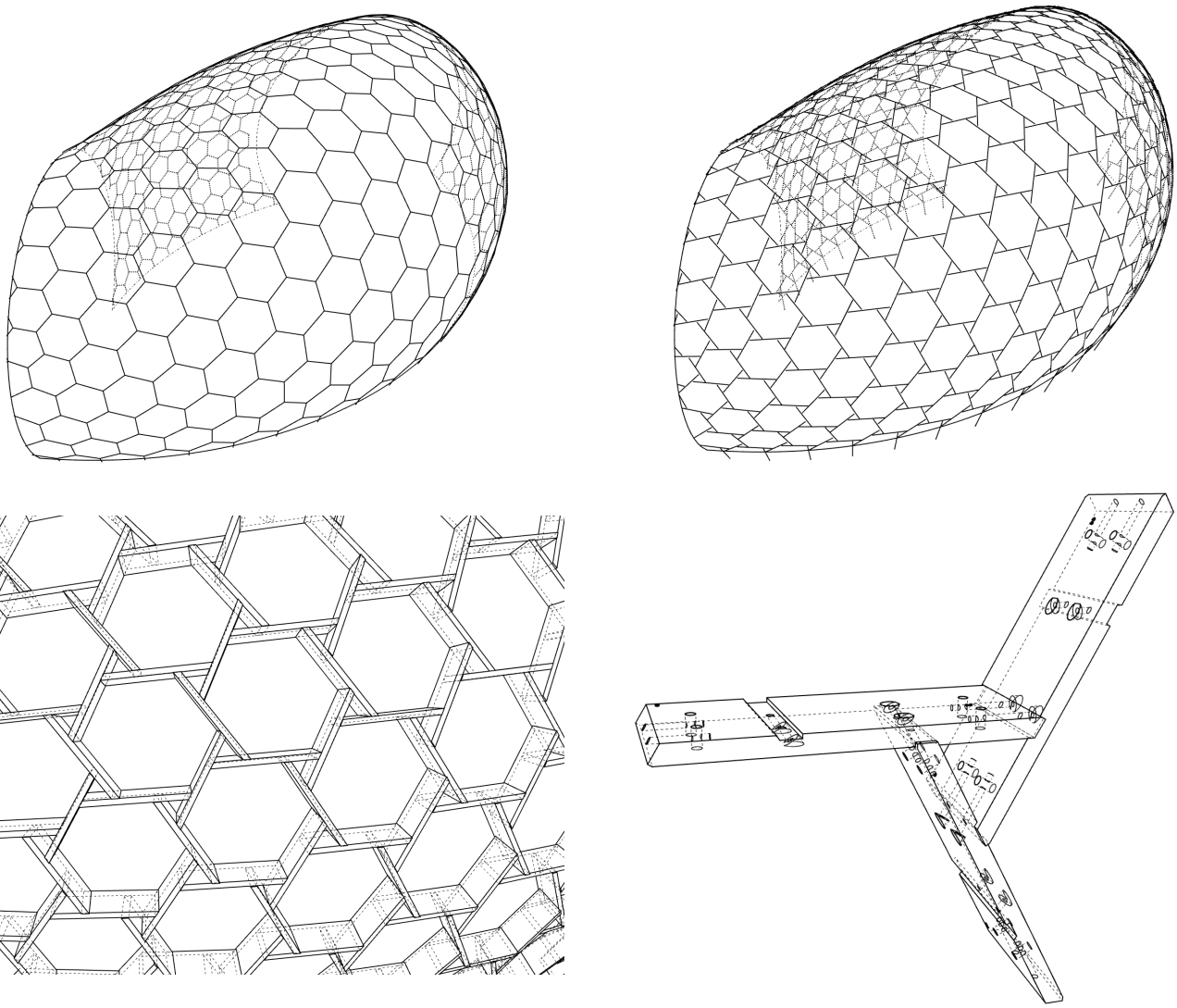


Fig. 3 *– **Top Left:** Hexagonal grid. **Top Right:** Edges rotated for reciprocal connection. **Middle Left:** Close-up of beam structure with thickness and depth added. **Middle Right:** Construction detail. **Lower:** KREOD Pavilion on display in London. Photo courtesy of Kin Ho.*

3. A Typical Workflow

The EvoluteTools plug-in is a practical tool that is very suited to professional use and solving real construction problems. As such, the primer is organized around the typical* workflow you'll use when creating panel layouts with EvoluteTools. To give you a preview of the information that will be covered and to provide some structure, I'll run you quickly through that workflow. Then in the next chapter we'll jump right in and try it out.

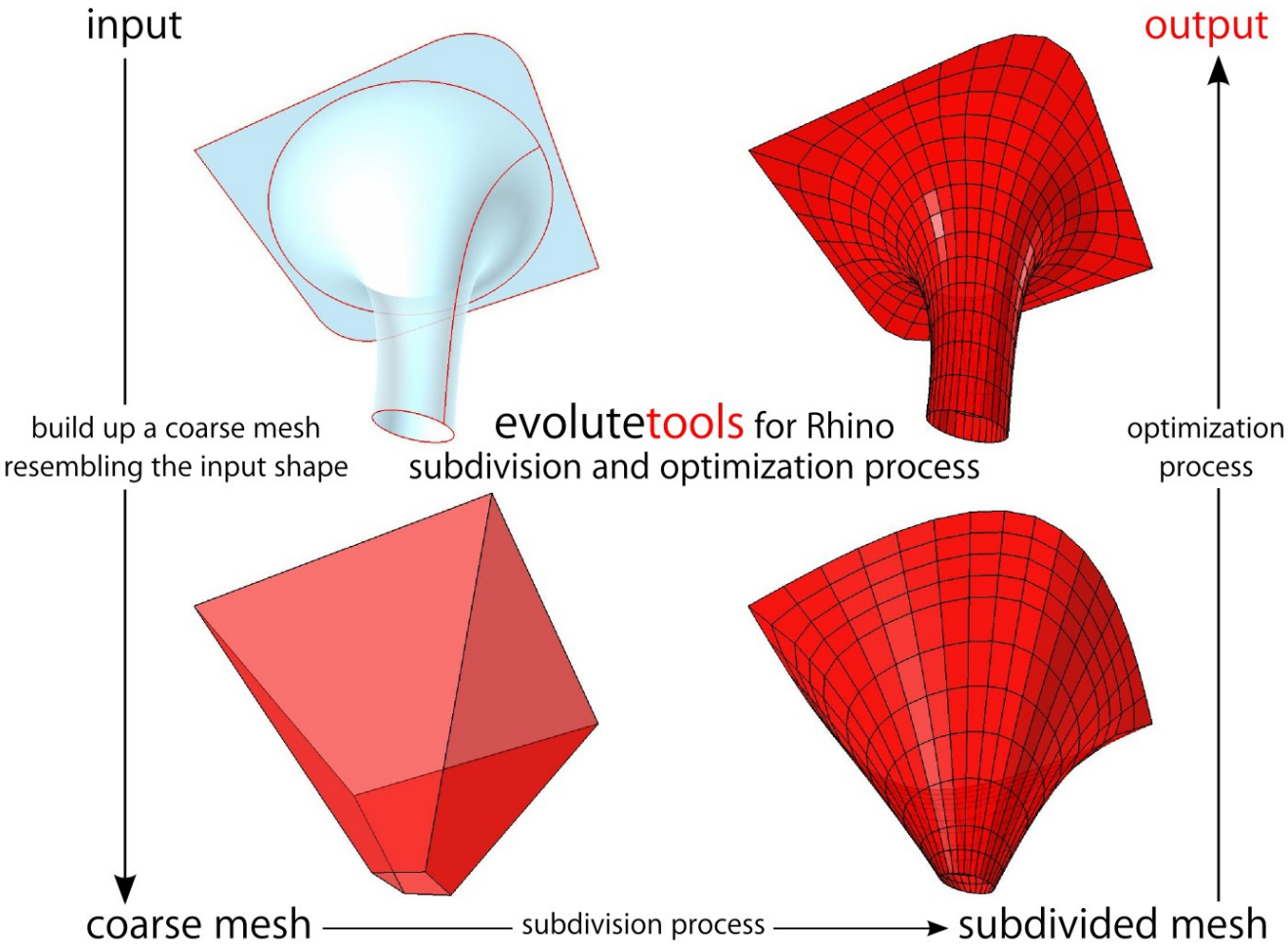


***Fig. 4** – Your workflow for using EvoluteTools will generally follow this process.*

* But not set in stone! As you become more experienced with EvoluteTools and adapt it to specific projects you will find it can be used in a variety of ways.

3.1 Step 1 : Define Your Reference Geometry

For architects and designers this should be easiest step; you will not want for a lack of forms. In this step, you let EvoluteTools know what shape you want to end up with and, if things go well, the end result will not deviate much from this shape. That target geometry can be created in any number of CAD programs as long as it can be imported into Rhino. The reference geometry can be a mesh, a surface, a polysurface, a T-Splines surfaces or any type of curve and polyline. This step is up to you and limited only by your imagination.

3.2 Step 2 : Create the "Coarse Mesh"

EvoluteTools works through the manipulation and optimization of meshes. Some of you will already be familiar with meshes and will understand why this type of geometry is preferred for panel layouts. If you aren't one of these know-it-alls, don't worry! We will discuss meshes and the reasons they are used by EvoluteTools in the Chapter 5 – Mesh Basics. For now, it is enough to know that we will start our work with a very simple mesh and the final product will be a much more detailed mesh with faces representing the desired panel layout. We call this simple first mesh the "coarse mesh."

Variations in the coarse mesh will often have profound impacts on the final product and will be discussed in Chapter 6 – The Coarse Mesh. The most important thing to remember is that this should be a **simple** mesh that roughly approximates the target volumetry.

3.3 Step 3 : Subdivision

A coarse mesh will generally be as simple as possible and thus have a very low number of faces. These large faces need to be divided into smaller faces that have roughly the dimensions of the panels intended for the final project. Instead of doing this division manually, EvoluteTools includes a tool that will automatically divide a mesh according to a number of different predefined rules. The process of dividing a mesh using a set of rules is called "subdivision" and the various available rules are discussed in Chapter 7 - Subdivision. This subdivision step can be repeated any number of times until the density of the mesh meets your requirements.

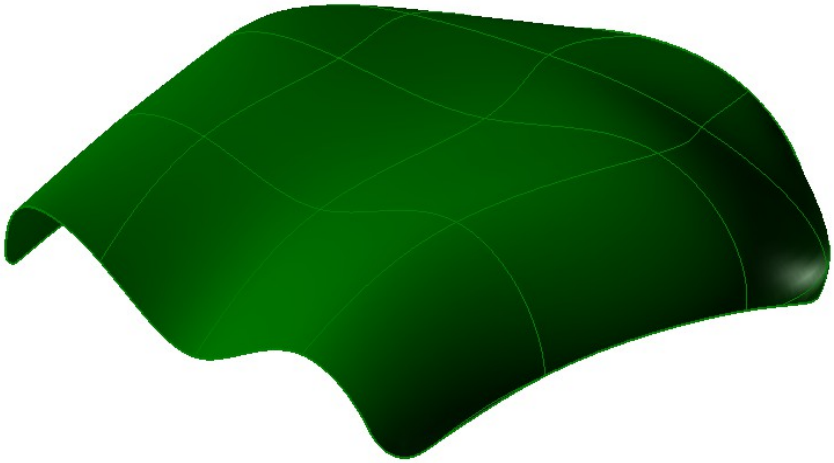
In addition to these automated subdivision tools that effect the entire mesh, there are local subdivision tools as well. These allow the user to control the subdivision with at a finer level of detail. They will also be discussed in Chapter 7.

3.4 Step 4 : Optimization

By now, you most likely won't have a result that looks much like your reference surface. This is where the true power of EvoluteTools shines. Its powerful optimization algorithm will adjust the shape of your mesh to match the target geometry. In addition to matching the shape, you will be able to set and fine-tune a number of constraints (such as planarity of the panels) to adjust the panel layout to the specific requirements of your project. [see Chapter 8 - The Optimization Chapter]

4. Getting our hands dirty – The first example

If this were a book on learning how to write code you would start with printing „Hello World!“ on the screen. I think you'll agree that this example is a whole lot cooler. In the Primer Examples folder [Download from: evolute.at], you will find a Rhino file called Primer_Example1. In this file you should find the strange object pictured below. If there is a pop up window that asks if you want to load saved optimization settings, select „Yes“.



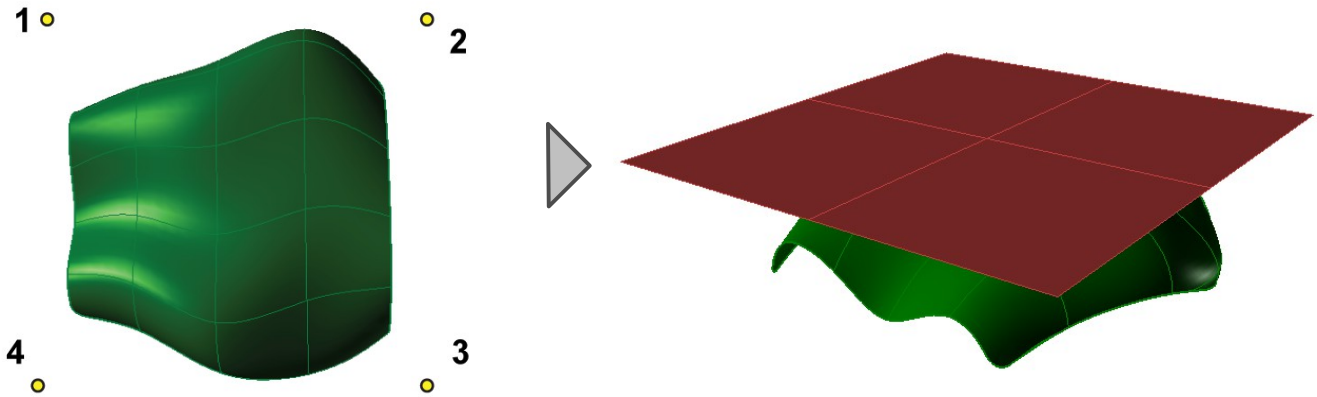
4.1 Example 1

Using EvoluteTools you will create a panel layout for this shape in just seven simple steps.

1. Following the typical workflow, our first step is to define our reference geometry.

   To do this simply select the object and then click on this button:  in the ET tool bar or type *'_etSetReference'* in the Rhino command line. Your reference is now set. To clear that reference object right click on that button or type *'_etClearReference'*. Clearing the reference is rarely needed though.

2. Next you will create a coarse mesh. The primary rule of coarse meshes is the tried and true K.I.S.S. method [keep it simple stupid] and it doesn't get much simpler than a single mesh face. From a top view of the object, select  or type *'_3DFace'* in the command line. Select four points to create a rectangular face with roughly the dimensions of the target geometry [see image below].

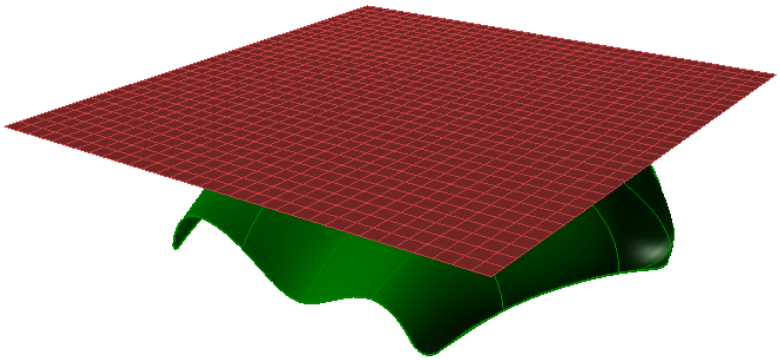


Now you should have a single mesh face that is approximately 50m by 50m. This would present a challenge for fabrication and obviously doesn't look very much like our intended shape. We need smaller panels!

3. Now you will divide up this mesh with the subdivision method called Catmull-Clark (in general, a safe bet for quadrilateral meshes). Select the coarse mesh. Then click on this button:  or type *'_etSubdivide'*. The command line will now ask you to select the subdivision rule. This prompt also shows you the currently selected rule. The default should be Catmull-Clark, but if yours says something else, click on rule name or type 'R'. This will bring up the list of available subdivision rules.

   There are a few to choose from, but at the moment we are only interested in Catmull-Clark. Select this rule and then hit [Enter]. You should see a new mesh has been created that fills the same area but now has 4 faces where before there was 1. [see image]

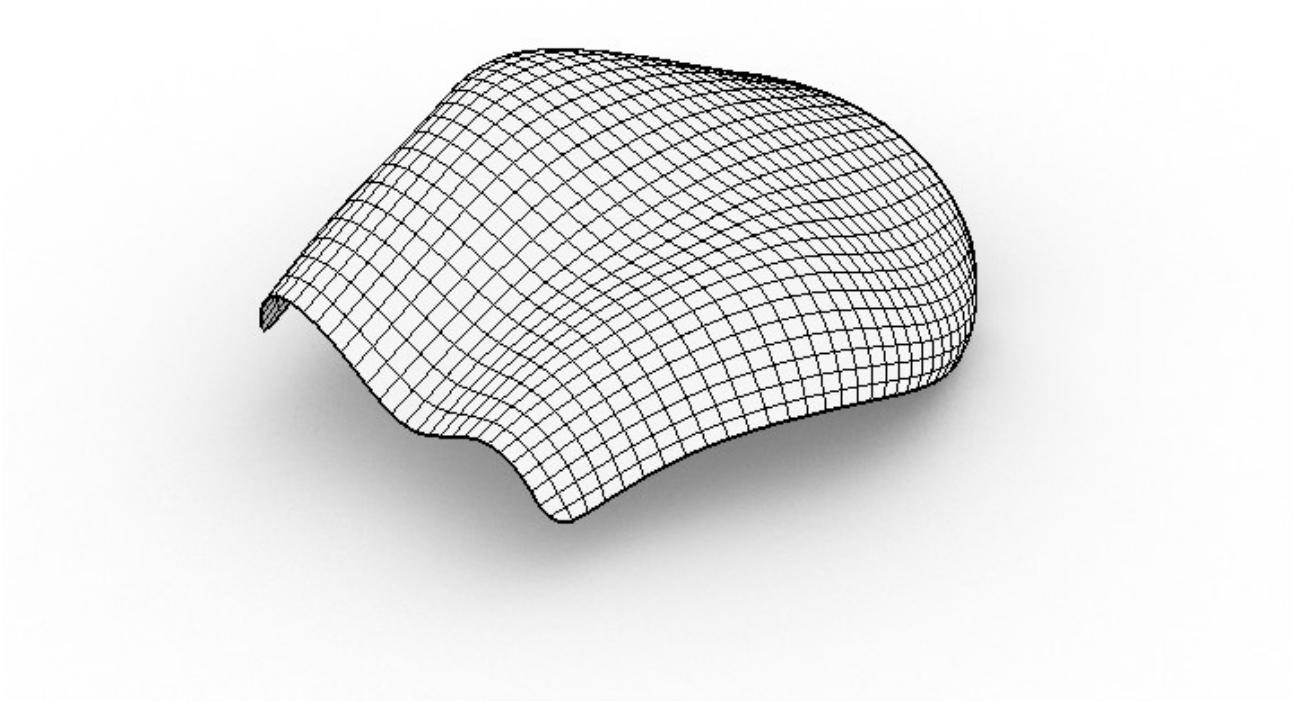
4. These panels are still too big to be useful for an actual building. You need to repeat the subdivision step a few more times. With the original coarse mesh still selected, run the subdivision command 4 more times. [Hint: typing [Enter] will restart the last command and typing a second time will run the subdivision].
   In the end, you should have 1024 faces that are about 1.6m x 1.6m.



5. Now that you have a subdivided mesh to work with, you need to delete the original coarse mesh. **Important:** If you skip this step, you won't be happy with the results! Find out more in Chapter 9.

6. There is just one more step before we see EvoluteTools work its magic. You have to set the Optimization settings. First, make sure they are at the default values by selecting this button:  or typing *'_etOptionsReset'* in the command line. The default setting is 0 for all the options except Surface Closeness and Curve Closeness. For this example, you will also add a bit of Fairness Curvature. For more on the specifics of the optimization options see Chapter 8. Click on this button:  or type *'_etOptionsImportance'*. Then select '*FairnessCurvature'* from the command line prompt or type 'A'. Type in '0.3' and then hit [Enter] twice.

7. The moment you've been waiting for is here! It is time to witness first hand the power of EvoluteTools. Hover the mouse confidently over the *'_etOptimize'* Button  . Now click! What happened? Doesn't quite look like you want? Click it again! In fact, keep clicking as long as you think the panel layout is improving or *"Optimization converged, no more changes were done to the mesh"* is printed on the command line.* Your result should look similar to the image below.



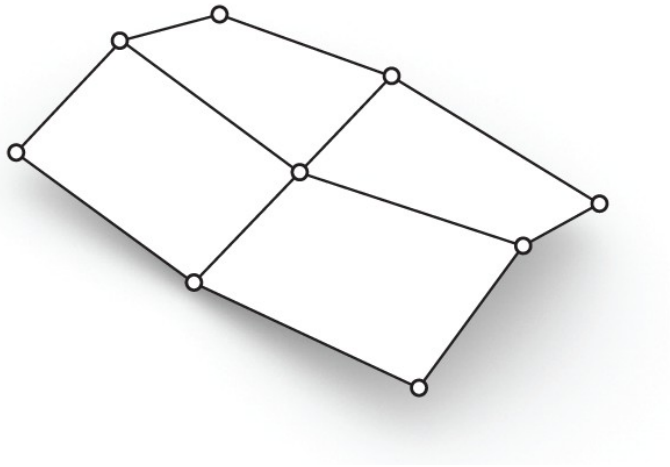
Pretty nice for a few minutes work and handful of mouse clicks!

* The reason for the repetitive clicking is that the optimization command only runs a limited number of iterations [set by the Optimization Toggles; default is 1] each time it is executed. This allows you to get a sense of how the mesh is changing between each optimization step [or steps]. If you like the improvements you can keep optimizing, but if something unexpected happens or you are unhappy with the results you can change the Optimization options and try again.

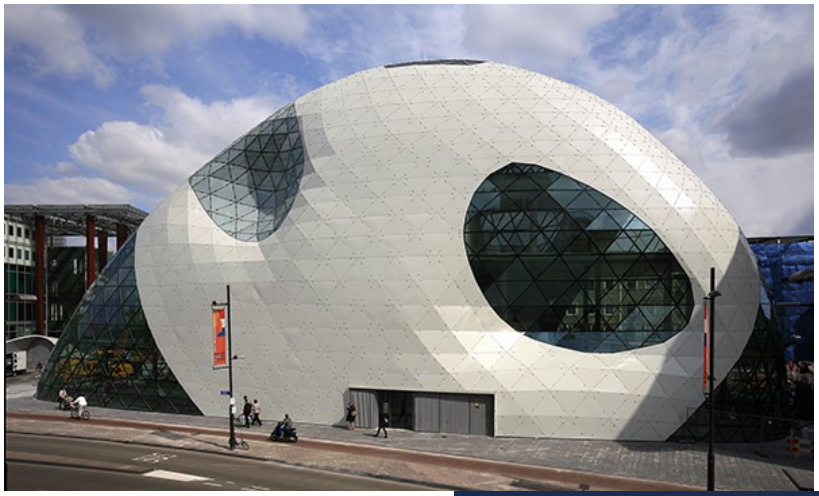
5. Mesh Basics

This chapter aims to give you a crash course in what meshes are and why they are used by EvoluteTools for panel layout and optimization. If you are familiar with meshes already or just want to get back to the action and leave the theory for later, feel free to skip ahead to next chapter on creating coarse meshes. If you are looking for a more in-depth treatment than is given here, I highly recommend the book, *Architectural Geometry* by Pottmann, Asperl, Hofer, and Kilian.

In the most general terms, meshes are a number of points (vertices) that are grouped together to define polygons (faces). These faces are typically one type of polygon, such as a triangle or quadrilateral. The faces are interconnected along shared common edges. See the image to the right for an example of a basic quadrilateral mesh.



This is not only an academic concern. The parallels between meshes and architecture are not difficult to imagine. Vertices, edges and faces can represent nodes, beams and the panels that comprise a building's skin, respectively. Indeed, mesh-like structures are becoming increasingly more common. The buildings shown below are a few such projects that have been worked on by Evolute.



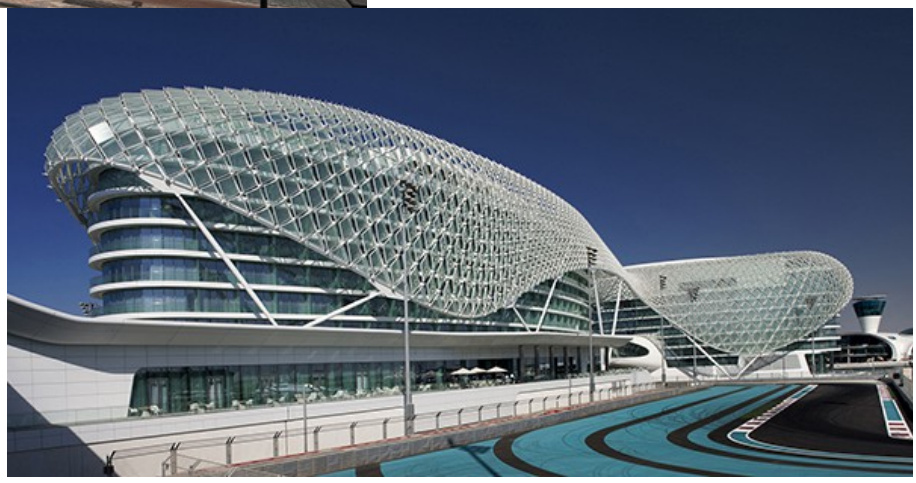


***Fig. 5** – A couple buildings with mesh-like properties.*
***Left:** The Blob by Massimiliano Fuksas. Eindhoven, Netherlands*

***Below:** Yas Island Marina Hotel by Asymptote Architecture, Abu Dhabi, United Arab Emirates*
Images courtesy of Waagner Biro

5.1 Connectivity and Geometry

Every mesh is defined by two primary properties: connectivity and geometry. The connectivity of a mesh is the description of how the vertices are connected to form the edges of the mesh faces. Each vertex is given a label and these labels are collected into sets that define a face polygon. [see the image below]

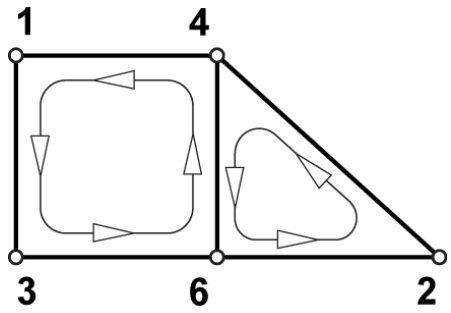
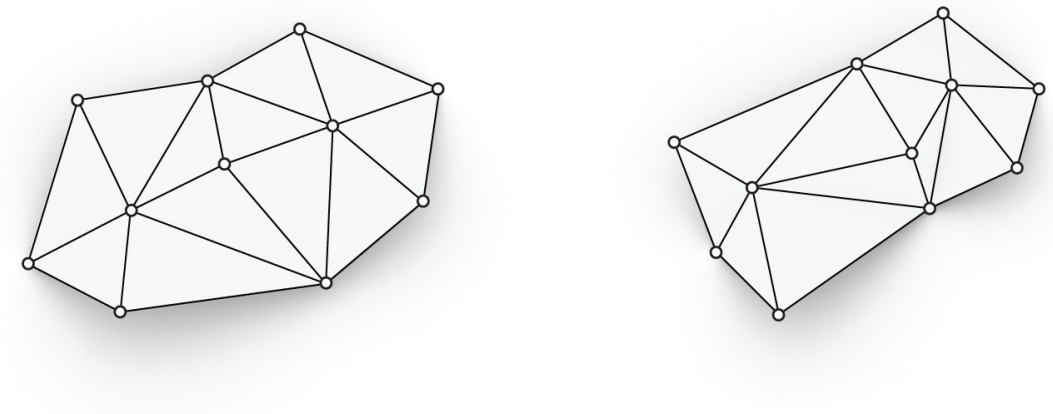
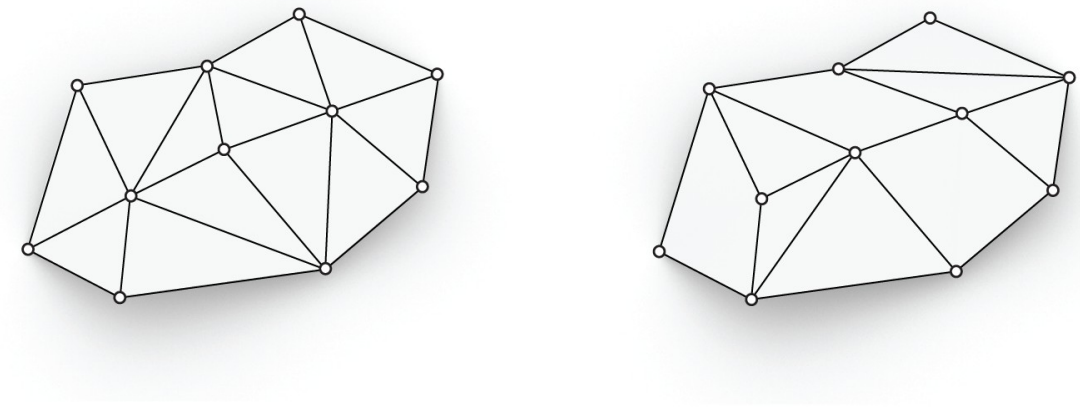


Fig 6. *- The face information (f 1 3 6 4) and (f 2 4 6) tells us: A quadrilateral face is defined (in a counterclockwise direction) by the vertices with labels 1, 3, 6 and 4. It meets a triangular face with vertices 2, 4, 6 along the common edge with vertices 4 and 6.*

The geometry of a mesh is simply the location of the vertices in space or, in other words, their coordinates. It is this property that the optimization engine in EvoluteTools acts on. It repeatedly moves the vertices around according to a set of rules and the settings defined by the user, but does not change the connectivity between the vertices. The various subdivision rules, on the other hand, often change both the geometry and connectivity of a mesh. To demonstrate how these two properties are independent take a look at the two pairs of meshes below. The first pair has the same connectivity and a very different geometry.

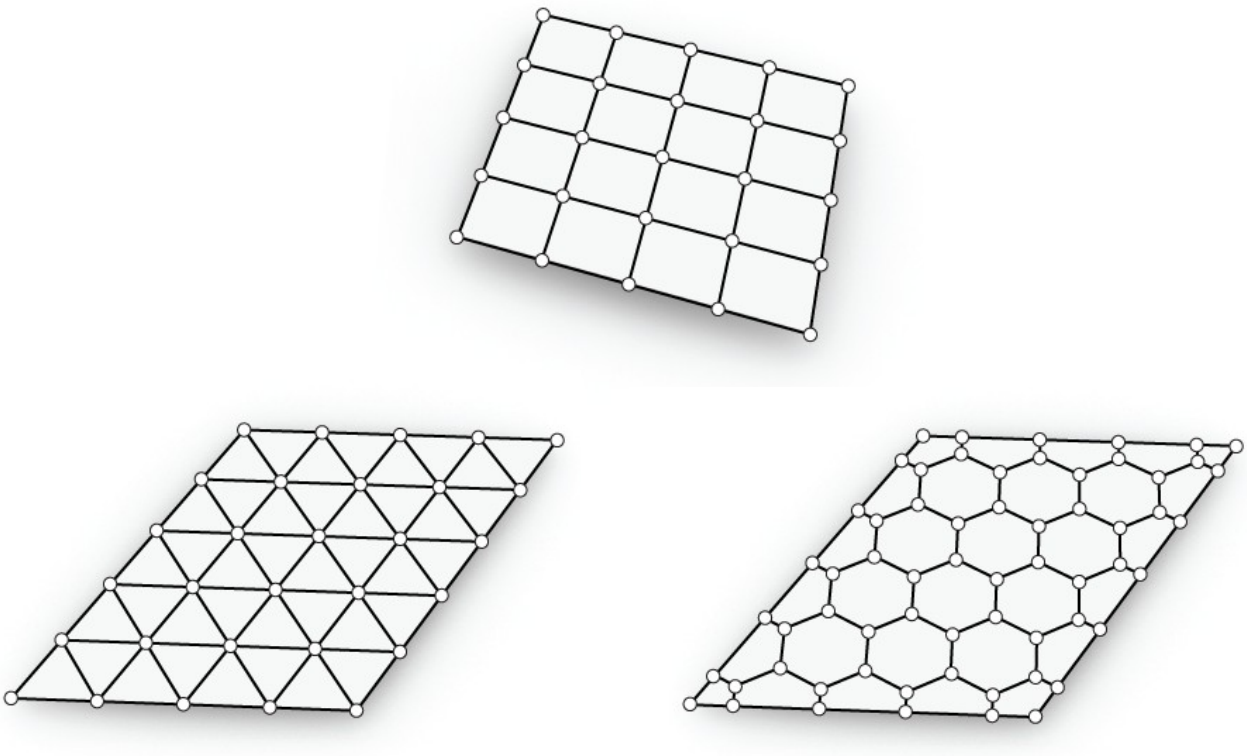


The second has the same geometry, but different connectivity.



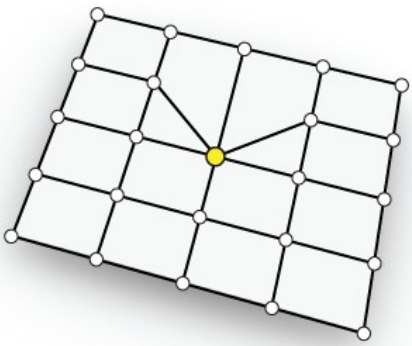
For most architectural applications it is desirable to have panels that have a similar shape and a uniform connectivity, but this is not guaranteed in every mesh. If you were making a curving grid shell out of rectangular panels, the occasional pentagon would look very out of place. Likewise, if you had 4 beams intersecting at every node, but at one you had 5 this would break the uniformity of the structure.

When dealing with meshes there is one bit of technical jargon that is worth remembering: the term “valence.” This describes the number of edges that intersect at a vertex. A picture is worth a thousand words, so let's take a look at the three most common kinds of meshes and the valence of their vertices.



The upper mesh is a regular quad mesh. Its interior vertices all have 4 coincident edges, thus they have a valence of 4. The edges have a valence of 3 and the corners have a valence of 2. The lower left mesh is your typical regular triangular mesh. The interior vertices all have a valence of 6, the edge vertices are valence 4 and the corners are valence 2 or 3. On the lower right is a regular hexagonal mesh. The typical interior vertex has a valence of 3. The edges and corners depend on the shape of the boundary, but in this case they are valence 3 and 2 respectively.

Interior vertices that have a valence different from the regular value (e.g. 5 versus 4 in a quad mesh or 4 versus 6 in triangular mesh) are given several names, irregular vertices, extraordinary vertices, or singularities. [see image right] It is worth noting that while these may be aesthetically undesirable, there are cases where they are unavoidable. Choosing where to place these singularities is an important part of the process of panelizing complex surfaces and will be discussed in more detail in the following chapters.



That should be enough background knowledge for now. In the next chapter, you'll put all this book-learning to good use creating the basic mesh that will become backbone of your final panel layout.

6. The Coarse Mesh

After you've created and set your reference surface, the first real challenge is to create a coarse mesh that can then be subdivided and optimized. If you are like me, you will be feeling a powerful urge to rush right through into the subdivision and optimization steps, but it is ***really*** worth it to take a moment and carefully consider the impacts that the coarse mesh has on the finished product.

6.1 Connectivity & Singularities

Perhaps the most significant of these impacts is the inheritance of connectivity and location of singularities. As you subdivide a mesh in the later steps, faces on a regular mesh will be nicely subdivided into new regular faces and interior vertices will have a consistent, regular valence. Meshes with irregular faces and vertices will be subdivided as well, but the result will be irregular faces and/or vertices with irregular valences. [For more on mesh connectivity and vertex valence see Chapter 5 - Mesh Basics] To put it simply, the mesh you end up with will inherit the characteristics, the good and the bad, of the coarse mesh you start with.

In order to demonstrate this, let's revisit the first example from Chapter 4.1.

6.1.1 Example 2

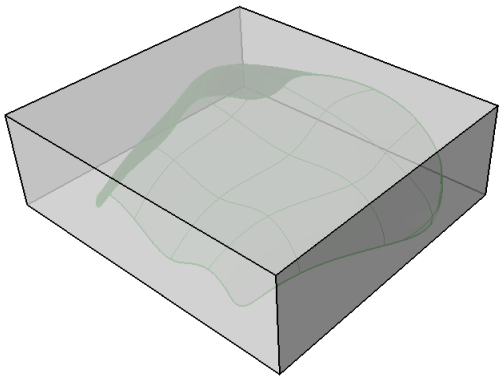
1. Start by opening the same file as before [Primer_Example1]. Again, we will set this shape as our reference geometry [same as Step 1 from Example 1].

2. In this example, you will start with a different kind of coarse mesh than used previously. You will create the coarse mesh using the Rhino Mesh Box tool  . Select that button or type *'_MeshBox'* in the command line.

    a. In the resulting prompt, make sure XFaces, YFaces, and ZFaces are set to 1, as shown below.

```
First corner of base ( Diagonal  3Point  Vertical  Center  XFaces=1  YFaces=1  ZFaces=1 ):
```

    b. Next, from an overhead view, select two corner points that describe a rectangle enclosing the reference geometry and then a third point to define the height. You should end up with box that is roughly 50 x 50 x 20m. [see image right]



    c. This is a good start, but our reference shape isn't enclosed. You need to delete the bottom face of the box.

    To do this select the Delete Mesh Face button:  from the EvoluteTools toolbar or type *'_etMeshDeleteFace'*. Then select the bottom face of the mesh box you created and hit enter.

d. After you've created or edited a coarse mesh it is always necessary to use the ET Weld command  to make sure the mesh is clean and ready to be used by EvoluteTools. This command can also be accessed by typing *'_etWeld'*.

3. For the final steps, follow the same procedure as Example 1. You should end up with a mesh similar to Mesh A in the figure below. Compare it to the result that started from a single mesh face.

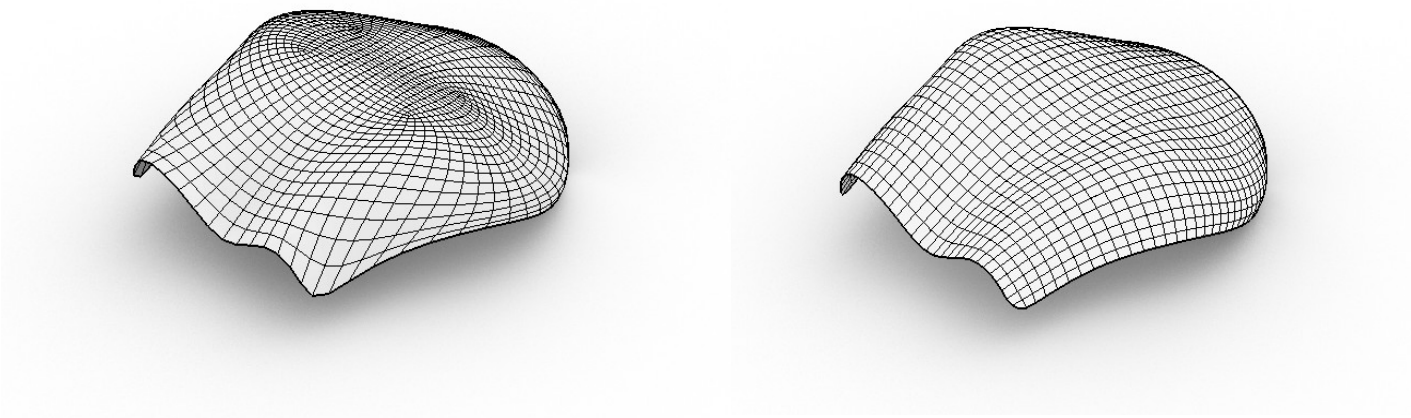


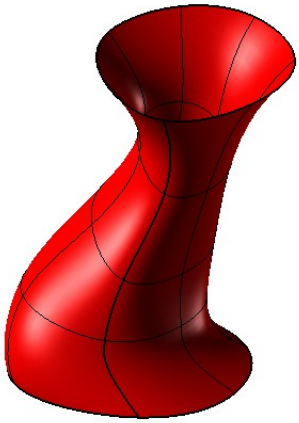
Fig. 7 *– Left: the result from Example 2, let's call it Mesh A. Right: the result from Example 1 - Mesh B.*

Notice how in Mesh A the panels radiate out from a cluster of smaller panels at the top, whereas this area is more regular in Mesh B. Take a closer look at that cluster on the top of the mesh. Do you see the 4 singularities? It is no coincidence that there are 4 singularities in the final mesh. They are inherited from the corners of the original coarse mesh box. Note that the corner vertex of a box (where three faces meet) has a valence of 3, where most vertices in a quad mesh have a valence of 4.

The coarse mesh in this example violated the K.I.S.S. principle by needlessly adding faces and creating singularities. That being said, as the complexity of the reference surface and its boundaries increases, there are cases where a starting with a single mesh face will not work (i.e. a closed geometry with no boundary). We'll take a look at another quick example:

6.1.2 Example 3

In this example you'll be panelizing a surface with a cylindrical form and exploring how this requires a more complex coarse mesh than the single plane used previously.



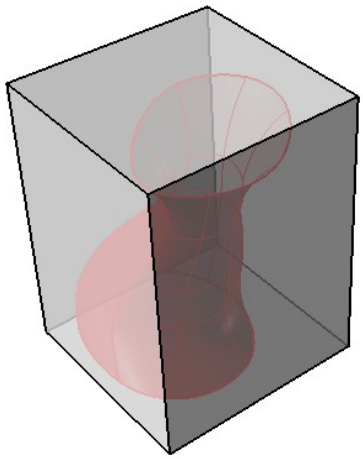
1. Open the file called Primer_Example3. This file contains a new shape. Set this as your reference surface.

2. Then follow steps 3 – 7 from Example 1. You should see pretty quickly that this isn't working.

Let's try using a mesh box for the coarse mesh.

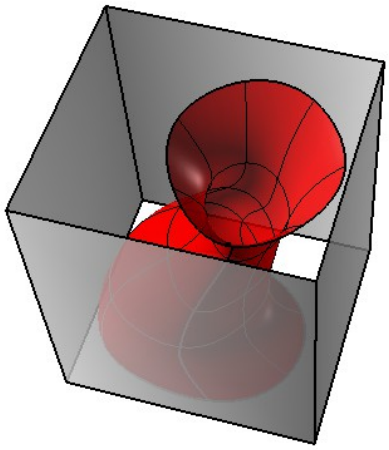
3. Delete the current mesh you were working with.

4. Just as you did in Example 2, create the coarse mesh using the Rhino Mesh Box tool.



   a. In the resulting prompt, make sure XFaces, YFaces, and ZFaces are set to 1.

   b. Next, from an overhead view, select two corner points that describe a rectangle enclosing the reference geometry and then a third point to define the height. You should end up with box that is roughly 60 x 70 x 80m. [see image right]



   c. This is a good start, but our reference shape isn't a closed surface. This time you need to delete the top **AND** bottom face of the box.

   d. Then weld the open mesh box using ET Weld command  .

   e. The result should look like the following image to the right:

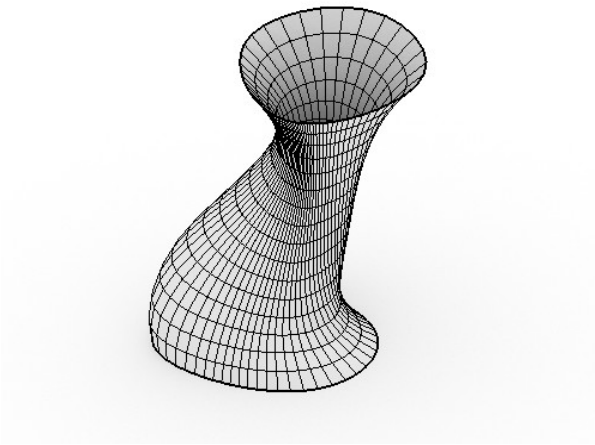
5. Now that you have your coarse mesh, subdivide it 4 times using the Catmull-Clark subdivision.

6. Delete to coarse mesh.

7. This time you'll use slightly different optimization settings from the previous examples. Restore the default settings and then set *'FairnessSprings'* to '0.1', *'FairnessCurvature'* to '0.3', and *'FairnessCurvatureVariation'* to '0.5'.* Leave the rest of the settings at their default.

```
Importance ( SurfaceCloseness=1  CurveCloseness=1  OriginalCloseness=0  FairnessSprings=0.1
FairnessCurvature=0.3  FairnessCurvatureVariation=0.5 … )
```

8. Now run the *'_etOptimize'* command  until you see a nice distribution of faces. This looks a lot better than the first try with a single face as a coarse mesh!



* For more on why these specific importance options were set for the fairness terms see Chapter 8 - The Optimization Chapter.

6.2 Resulting Panel Proportions

In addition to the connectivity, meshes resulting from the use of the subdivision tool on the coarse mesh will also inherit the general panel proportions of the coarse mesh. In other words, if the coarse mesh is composed of long skinny rectangular faces (i.e. a high aspect ratio), the resulting subdivision mesh will have long and skinny faces as well (although depending on the subdivision method, not necessarily rectangular). Here is a quick example of a case where this property is passed to the final mesh:

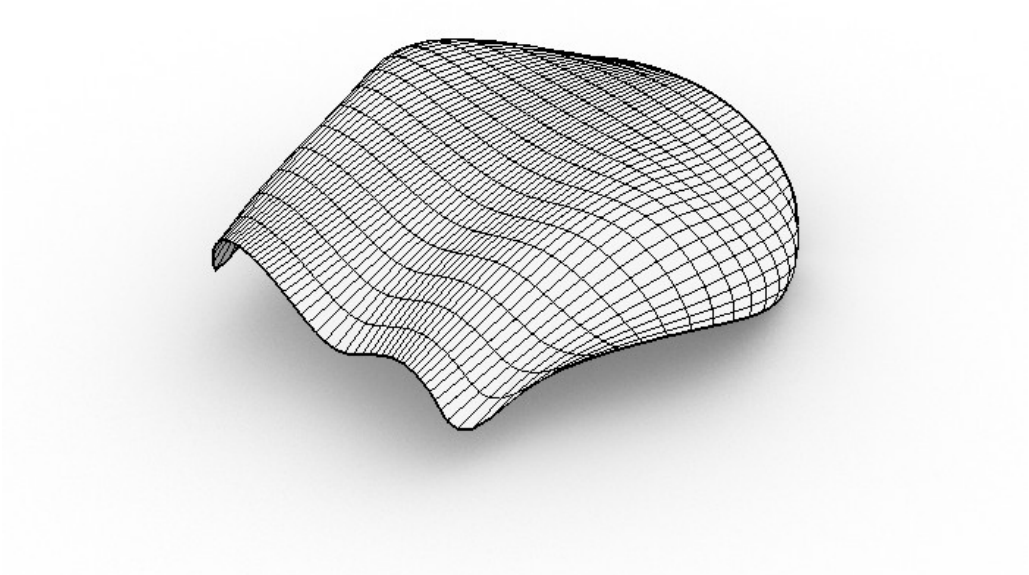
6.2.1 Example 4

1. Start by opening the file called [Primer_Example4]. You will see the same shape we worked with in the first example.

2. Next set this shape as your reference geometry [same as Step 1 from Example 1].

3. You should also see a mesh plane already positioned above the object. You will use this as your coarse mesh. Notice that this mesh has four long and narrow faces rather than the single square face you used in Example 1.

4. Follow the steps from Example 1 starting from Step 3. Your result should look something like this:



See how the resulting panels retain the thin rectangular proportion of the coarse mesh?

Note: Unlike the connectivity, there isn't always such a direct connection between the panel proportions of the final optimized mesh and the aspect ratio of the coarse mesh faces. You will often have to subdivide and run a few optimization steps in order to examine the result. Then you can step back to change the coarse mesh or simply adjust the subdivided mesh with the local subdivision tools described below in [Chapter 7 - Subdivision].

6.3 Where to Begin?

Now you've seen a few examples of how a coarse mesh can impact the final result and should be getting the idea that not all coarse meshes are created equal. Moreover, it will not always be immediately apparent what kind of coarse mesh to use. There are no hard and fast rules. It is a matter of user experience, project requirements, and an understanding of how the subdivision and optimization tools work. What follows is a general set of steps for creating coarse meshes, as well as an example to demonstrate these steps in action.

Step 1 – Create

You will have the most control over the coarse mesh when you create it face by face using Rhino's 3D Face command *'_3DFace'*,  . Using this tool and the appropriate OSnaps, it is fairly easy to create a basic mesh that approximates the reference surface.

As you've already seen, the *'_MeshBox'* command can be very useful in certain cases. To a lesser extent, any of Rhino's Mesh Primitives could be used. If your reference surface has any openings, it may be necessary to delete some faces as you did in Example 3.

Warning: Rhino's *'_Mesh'* command will convert surfaces into meshes, but the results are generally messy, irregular, and way too detailed for a coarse mesh.

Step 2 - Join & Weld

If you have made your reference mesh from a collection of meshes or mesh faces, you need to join them together using the Rhino *'_Join'* command or clicking  . Regardless of how you've constructed your coarse mesh, it is important to use EvoluteTools' Weld command  or *'_etWeld'*. This will clean the mesh of duplicate vertices and prepare it for use with the other EvoluteTools commands.

Step 3 – Set Vertices as corners and/or fixed

If your target geometry has sharp corners, points, spikes, or any jabby bits that coincide with vertices of the coarse mesh, it is wise to set these vertices as corners and fixed so that EvoluteTools knows to treat these special cases. The functionality of *'_etSetVertexCorner'* /  and *'_etSetVertexFixed'* /  is addressed in Chapter 8.4.

Step 4 - Local subdivision (i.e. Loop Cuts, diagonals, mesh cut)

At this stage, you may want to increase the detail in certain areas of your coarse mesh or change the panel proportions before you start the subdivision. To do this you'll need the local subdivision tools [more in Chapter 7.3]. Knowing what faces to divide and why is somewhat a matter of experience, but you'll soon get a feel for it.

6.3.1 Coarse Mesh Tips & Tricks

1. Keep it simple

This has been mentioned before, but can't be said enough. The creation of the coarse mesh is an exercise in minimalism.

an ancient pond
a frog jumps in
-- plop!

2. Watch your singularities

There will be cases when singularities in your mesh are unavoidable, but you can exert a lot of control over where these points will be located by the careful design of the coarse mesh. Remember that box corners are singularities.

3. Mesh from NURBS control polygon

If your target geometry is a single surface with a few control points, you can use the Rhino command *'_ExtractControlPolygon'* to quickly create a simple mesh that will approximate your surface very well.

4. Mesh from T-Splines surface

Additionally, if you used the T-Splines plugins to create your reference surface, the command *'_tsMesh'* will convert a T-Spline into a simple mesh.

7. Subdivision

Subdivision serves two primary purposes. The first purpose is to provide a convenient method for creating a more detailed mesh with smaller faces from a simple coarse mesh. This is the way you've used the subdivision command in the previous examples. The second role it fulfills is that of a design tool for enhancing the aesthetic (or geometric) properties by changing the connectivity of the mesh.

In this chapter, we will cover the various subdivision tools available in EvoluteTools and how they affect meshes. Then, through a couple of examples, we will examine how those subdivision routines can be used as a design tool.

7.1 Selecting the right tool

There are two types of subdivision tools available in EvoluteTools, global and local. The global subdivision tools refer to the various options in the *'_etSubdivide'* command,  . These modify the entire mesh. The local subdivision tools  , allow the user to subdivide a mesh in a specific area only.

When we talk about the various options in the *'_etSubdivide'* command, we refer to them as subdivision algorithms. This is really just a way of saying that they are step-by-step procedures for subdividing a mesh. Each option follows a different set of steps and thus each one will have a different, but predictable, effect on a given mesh. EvoluteTools includes 11 global subdivision options (9 in the Lite version) and 5 local subdivision tools. This may seem daunting at first, but by the time you've finished this chapter you'll be on a first name basis with these subdivision tools.

If any of the descriptions below aren't 100% clear to you, I suggest that you experiment with the subdivision tools on single planar faces both triangular and quadrilateral. (Remember the *'_3DFace'* tool!) Then try them out on increasingly complex geometries.

Note: After running the *'_etSubdivide'* command there will be two meshes in the drawing: the original mesh that was selected and the newly created subdivided mesh. These two meshes are connected. This means that all the vertices of the subdivided mesh depend on one or more vertices of the original mesh. For more on this and its implications see [Chapter 9.2 - Logical Connections]. At the moment, we will concern ourselves with the functionality of the subdivision tools.

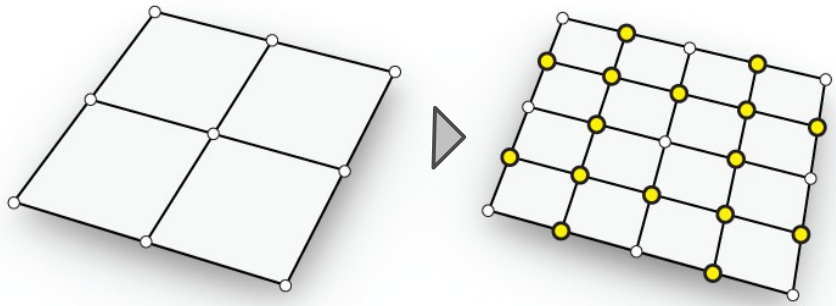
7.2 Global Subdivision Tools

In this section, I will discuss the various subdivision algorithms contained in the *'_etSubdivide'* command. These tools will affect the entire mesh.

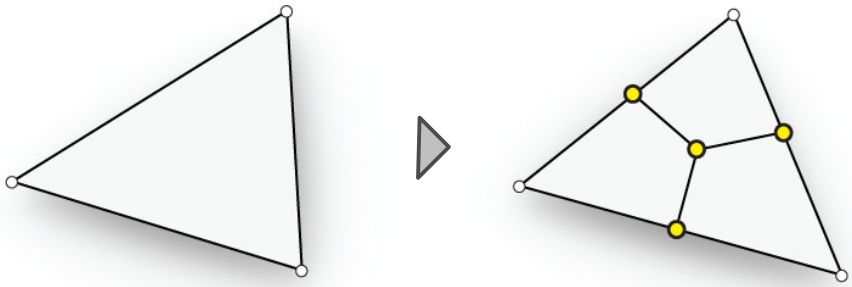
7.2.1 Catmull-Clark

When you are working with quad meshes, Catmull-Clark will be your go-to subdivision tool. In general terms, the algorithm works by adding a vertex in the center of each face and near the mid point of each edge. Then the new edge vertices are each connected to the new center. This essentially divides each quad face into 4 smaller quads.

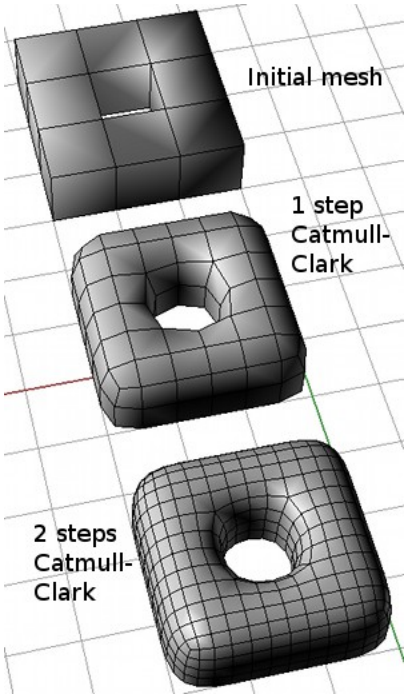
The precise details of the algorithm aren't necessary to know, but they can be found easily enough online if you are interested.



You do not need to start with a quad mesh, the Catmull-Clark tool will accept any mesh input. However, the output will always be a quad mesh. For example, applying this algorithm to a triangular face will yield 3 quad faces. [see image below]



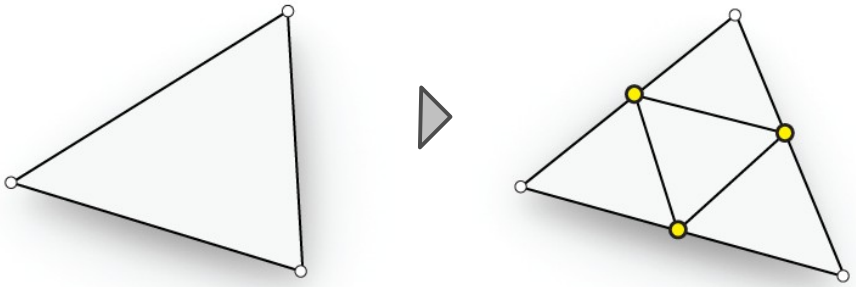
One important caveat and something you've seen in action already when using Catmull-Clark: It does not just add new vertices, edges, and faces. The original vertices are moved as well based on a weighted average with surrounding vertices. This results in a gradual smoothing of the mesh with each subdivision step.



In this section and in the following descriptions, I am only roughly describing a how these algorithms handle the simplified case of a planar mesh. Applied to 3-dimensional forms, their behavior is slightly more complex. However, this will give you a starting point for understanding how they function.

7.2.2 Loop

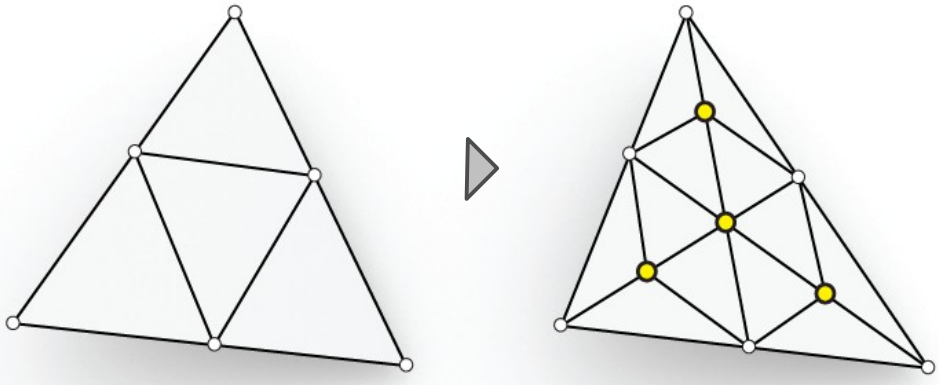
Loop Subdivision works only with triangular meshes and always produces triangular meshes. The essential mechanic of the Loop subdivision algorithm is creating new vertices at the midpoint of the 3 edges and then connecting them. This form a triangle inside the original triangular face and divides it into 4 smaller triangles. Thus, the number of faces will quadruple each time this subdivision is run.



7.2.3 Sqrt3 «»

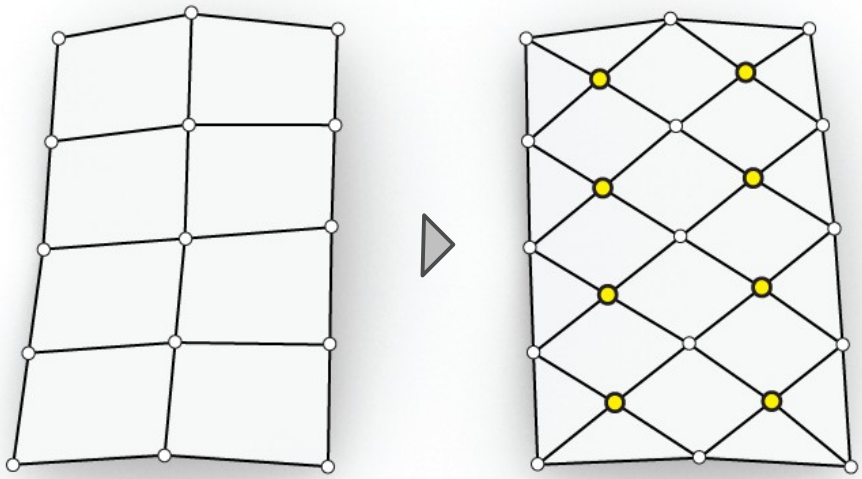
The output of the Sqrt3 subdivision is also a triangular mesh. Unlike the Loop method this can take any kind of mesh as an input. That said, if there are non-triangle faces, these will be subdivided into triangles, but the connectivity will not be regular. This is best used on uniformly triangular meshes.

Sqrt3 creates new vertices at the center of each face and then connects them to existing face vertices as well as the new vertices of adjacent faces. The existing edge lines are removed. [see image below]



7.2.4 Diagonalize

There are no mesh restrictions with the Diagonalize subdivision. This command will accept meshes with any kind of connectivity. The algorithm essentially adds a vertex in the center of each face and then connects that new vertex to the existing face vertices. Then the original edge lines are deleted. When it is applied to a quad mesh, it easy to see why it is called diagonalize. [see image]



When working with quad meshes, this is a good way to switch the orientation of your panel layout from orthogonal to a diagrid pattern. Additionally, it will often give your panel layouts a more flowing or sweeping look.

Regardless of the input (even non-quad and poly meshes), the result will be a predominately quad mesh with triangular faces along the boundary. If you examine the image above, it should be clear why. Each new inner face, which is not at the boundary, will be composed of the two midpoints of the old adjacent faces and the four lines connecting those to the two vertices of the previous shared edge. This will form a quad even if the two adjacent faces are defined by different polygons. [see image]

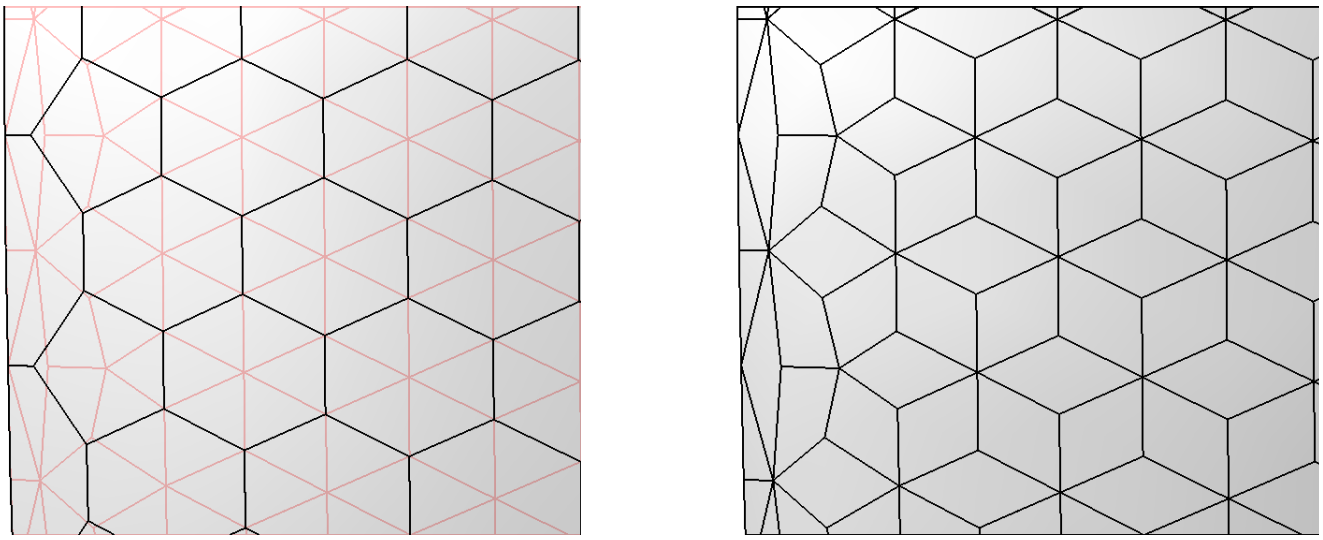
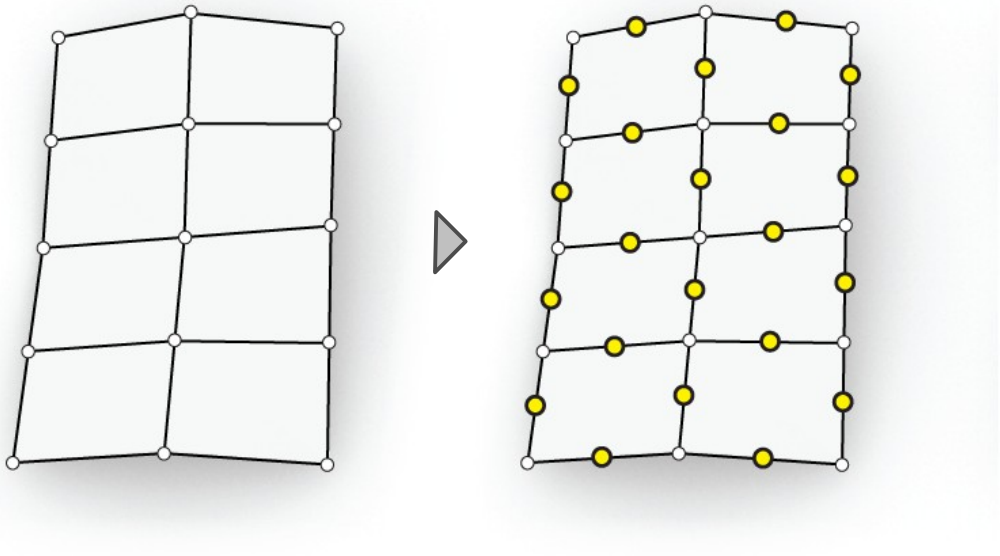


Fig. 8 *– Left: N-gon mesh with Diagonalize overlaid in red. Right: The Diagonalize result from the n-gon mesh.*

7.2.5 Edge Split «»

Edgesplit is a bit of a black-sheep in the Subdivision family. It doesn't add new faces or change the connectivity of existing vertices. Instead, it simply adds new vertices at the midpoint of each edge. In other words, every edge in the mesh is replaced by a polyline with two segments.
[see image]



These extra vertices may allow the mesh to better approximate the target surface. Additionally, adding these vertices and increasing the number of edge segments will drastically effect the results of subsequent subdivisions. Using this algorithm in combination with the other subdivision routines can lead to some very interesting panel patterns. For more see Example 7 – Patterning with Combinations, later in this chapter.

7.2.6 Identity «»

This subdivision routine doesn't perform any subdivision at all. In fact, it only makes a copy of the original mesh. At first this may be a bit confusing. Why not just use Rhino's copy command if you need a copy of the mesh? Why is this command even listed under the subdivision if there is no division going on?

The answer to both questions is the same: using the Identity subdivision tool retains the logical connection between the parent mesh and the copy. This is useful if you want to perform local subdivision or other mesh edits (i.e. deleting faces, loop cuts, etc.) on a mesh but keep the original mesh to use as a control surface You'll find out more about this connection in Chapter 9.2.

7.2.7 Dual and Dual with Boundary

The primary functionality of the Dual and Dual with Boundary subdivision algorithms is to convert between triangle and hex meshes. However, they will accept any mesh as input and can be used in patterning combinations to good effect. The basic process of the Dual subdivision is the addition of vertices in the face centers and then the connection of these added vertices across adjacent edges in the original mesh. For a quad-mesh, this results in a new quad mesh, but for a triangular mesh this results in a hexagonal mesh. See section 8.5.2 in Chapter 8 for the creation and optimization of a hex mesh.

Dual with Boundary differs in that it fills in the mesh to the boundary edge with n-gon faces.

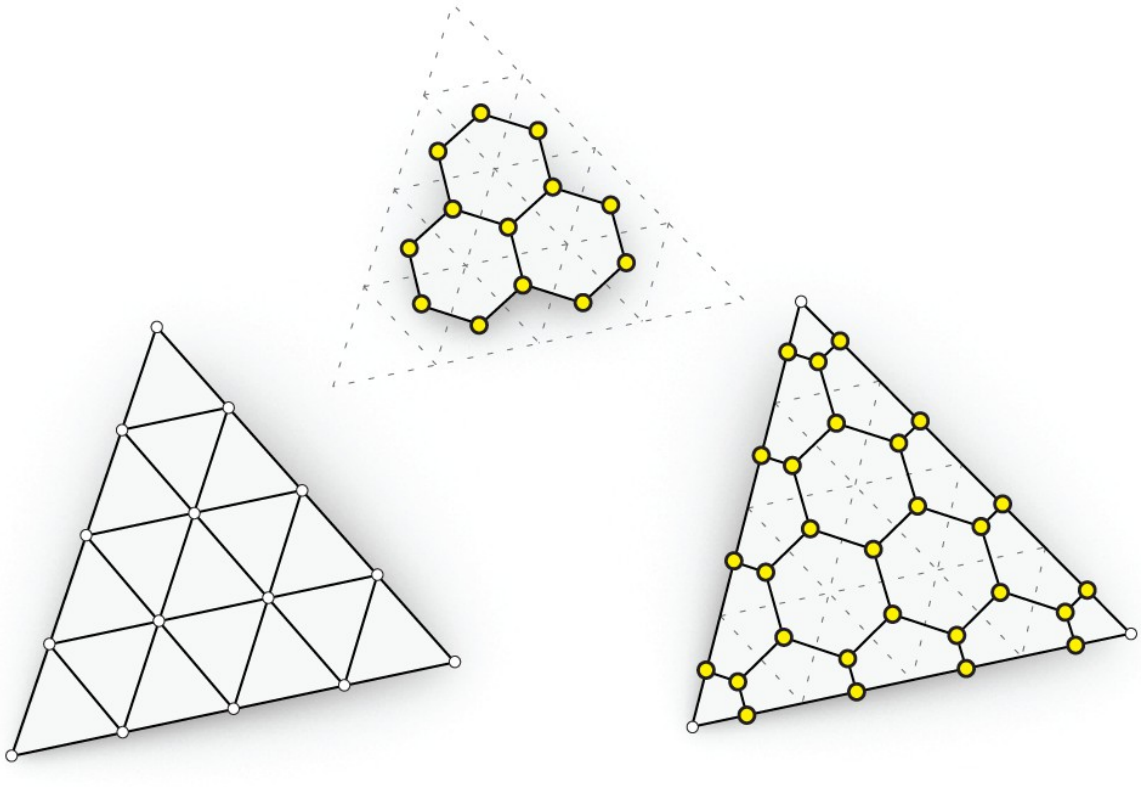


Fig. 9 *– A dual and dual with boundary subdivision.* ***Above:*** *Dual subdivision on a triangle mesh produces a mesh composed only of hexagons.* ***Lower Right:*** *Dual with Boundary fills to the boundary with various and sundry ngons.*

7.2.8 Dual Edge

Despite a similar name, Dual Edge shouldn't be confused with the other Duals. It has some unique and very interesting properties. The most useful of which is when applied to a quad mesh, half the resulting quads in the mesh will be completely planar. This occurs without any optimization. It is a purely geometric property. The remaining, non-planar quads can be triangulated for a quick and easy panel layout with only planar elements. [see Example 6 – Planar Panels the Quick and Easy Way later in this chapter]

The Dual Edge algorithm begins by adding vertices at the midpoint of each edge, like many of the other subdivision methods. Then, using the new points, it creates a new face inscribed within the original face polygon for each face in the mesh. The original edges are removed and the resulting gaps between the new faces are filled in with n-gons.

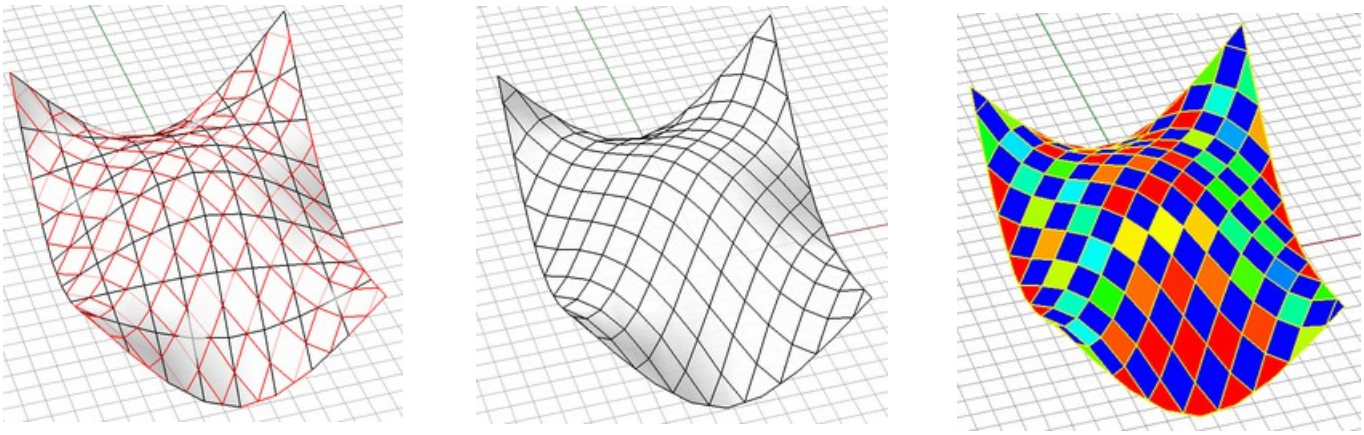
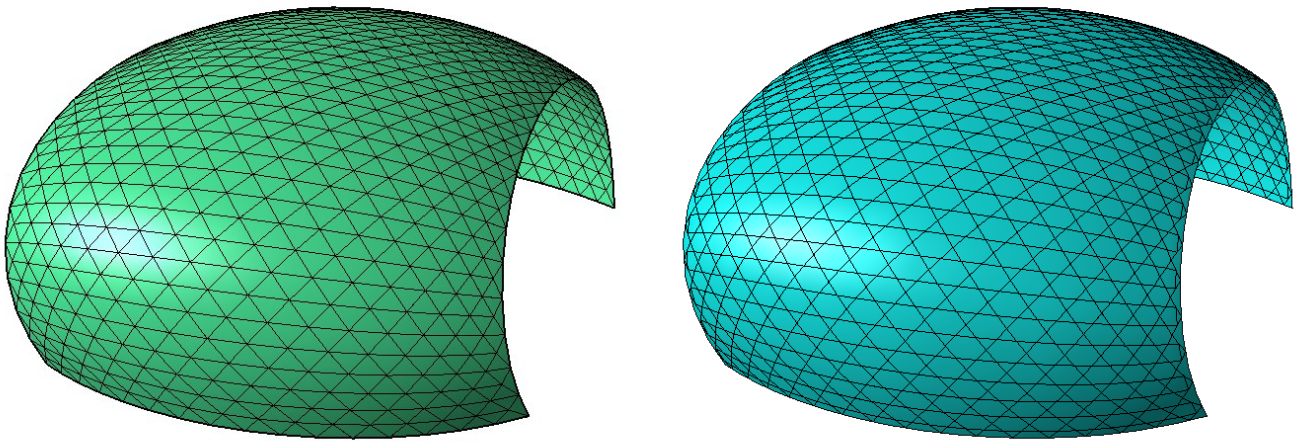


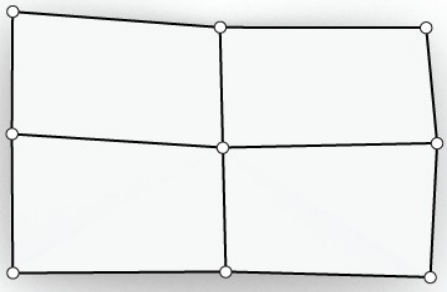
***Fig. 10**– A Dual Edge subdivision. **Left:** The original mesh with the subdivision mesh overlaid in red. **Center:** The resulting subdivision mesh. **Right:** Planarity analysis of the subdivision mesh. The dark blue faces are completely planar.*

When the mesh is triangular, then the result will be a rather nice tiling of hexagons and triangles.

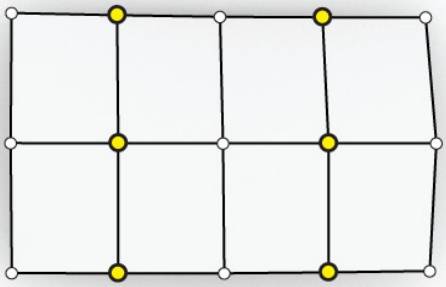


This is only available in EvoluteTools Pro.

7.2.9 Strips «»

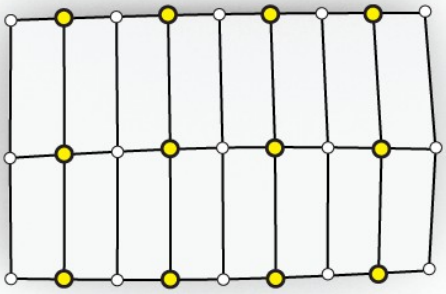


Strip Subdivision will accept **only** a quad mesh as an input. This subdivision will subdivide each face by cutting it in half in a particular direction. Repeated use of this command will approximate strips of continuous ruled surfaces. [Note: not necessarily developable] In order to set the direction of the rulings (i.e. the direction of the cuts), you must first use another tool called,



enigmatically, *'_etSetRulingDirection'* or this button  in the EvoluteTools toolbar. When you run this command it will ask you to select an edge on the mesh that is parallel to the ruling direction you desire.

Tip: When creating a coarse mesh or editing an existing mesh with the end goal of strip subdivision all vertices in the mesh should have valence 2, 4, or 6.



This is only available in EvoluteTools Pro.

7.2.10 TriHex «»

TriHex only accepts triangular meshes as inputs. The result of the TriHex is a tiling of hexagonal and triangular faces similar to that of Dual Edge when applied to a triangular mesh. However, if you test this on an arbitrary triangular mesh, the result will look „wobbly“ compared to Dual Edge. This is a result of how the algorithm selects the new point on the existing mesh edges. Rather than select the mid point like Dual Edge, TriHex uses the ballpacking property of the mesh [see Chapter 8.2.7 - Ballpacking «»] to determine the location of the new vertices.

This will only produce a “clean-looking“ mesh when it used on a mesh that has already been optimized for ballpacking.

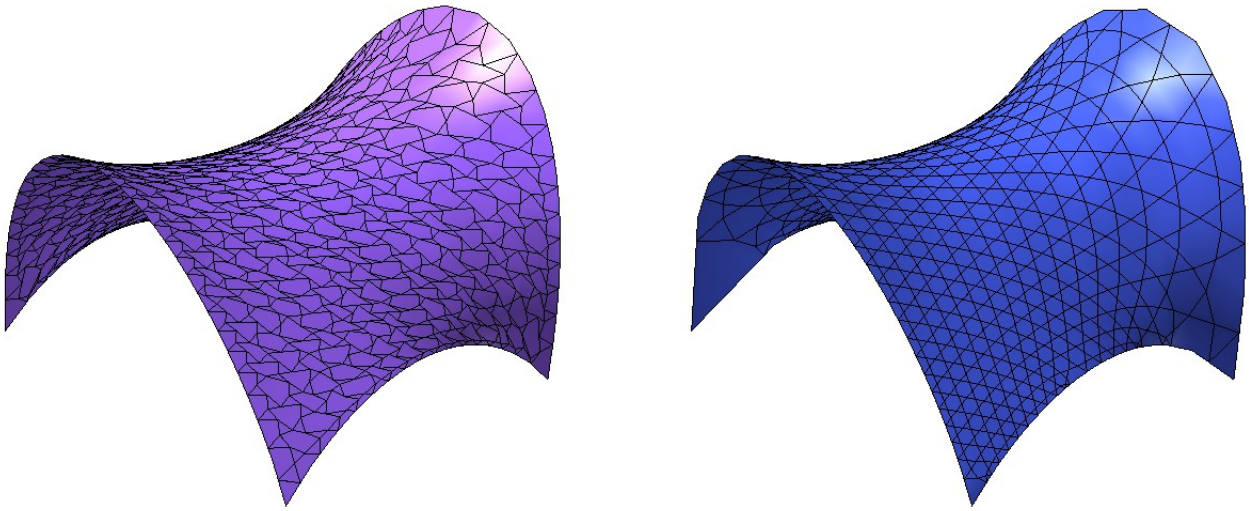
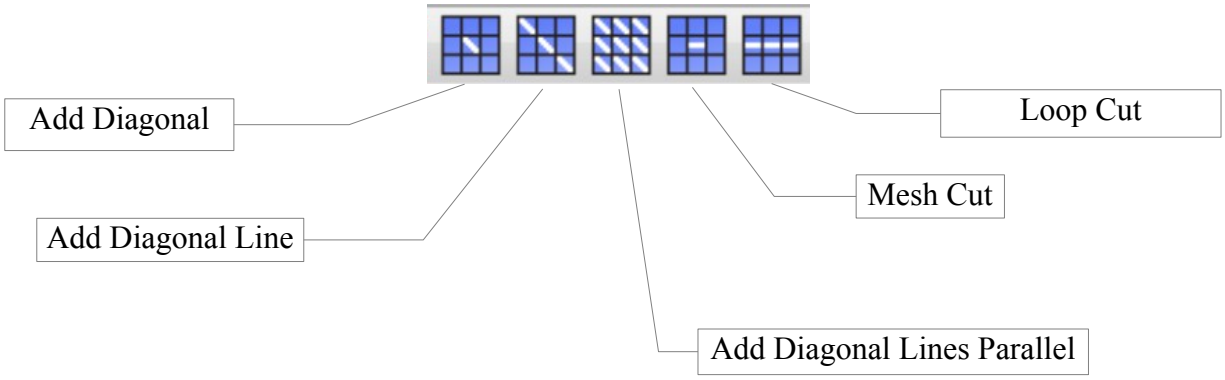


***Fig. 11** – TriHex subdivision results. **Left:** TriHex result without first optimizing parent mesh for Ballpacking is often wobbly, this is an extreme case. **Right:** The parent mesh has been optimized for the Ballpacking property and the resulting TriHex mesh is much more regular.*

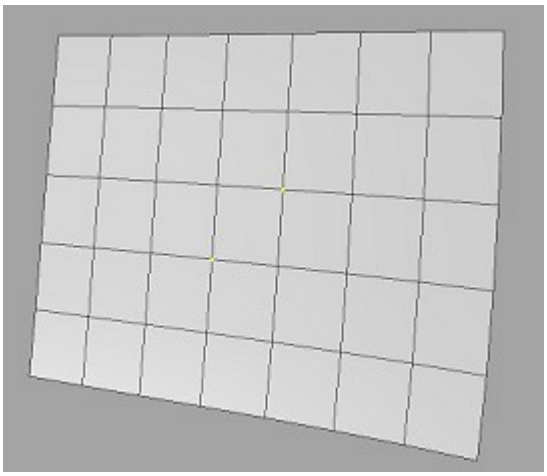
7.3 Local Subdivision Tools

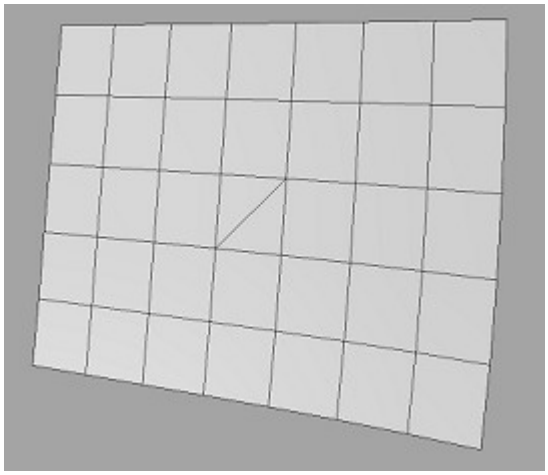
In addition to the global subdivision algorithms, there are also 5 local subdivision tools.



7.3.1 AddDiagonal

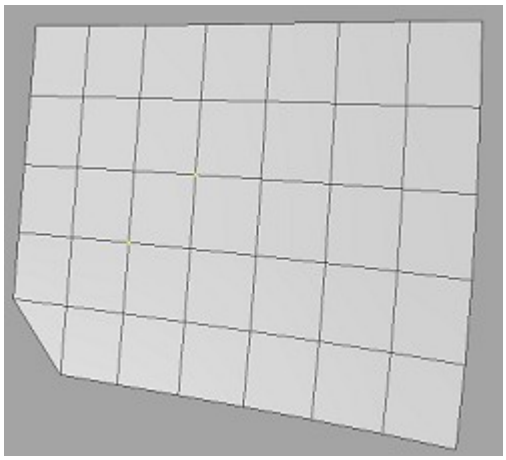
'_etMeshAddDiagonal' cuts a single quad face into two triangular faces by adding an edge between two vertices. The command prompts for the user to select of the two desired vertices. If they are opposite vertices of a single mesh quad face, then an edge is added between them.

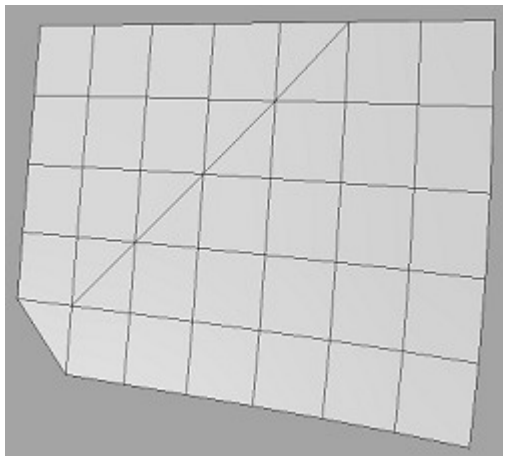




7.3.2 AddDiagonalLine

'_etMeshAddDiagonalLine' performs the same action as the Add Diagonal tool, but if possible, the diagonal is continued in both directions until either a non-quad face or a mesh boundary is encountered. Thus creating a new polyline within the mesh.





7.3.3 AddDiagonalLinesParallel

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

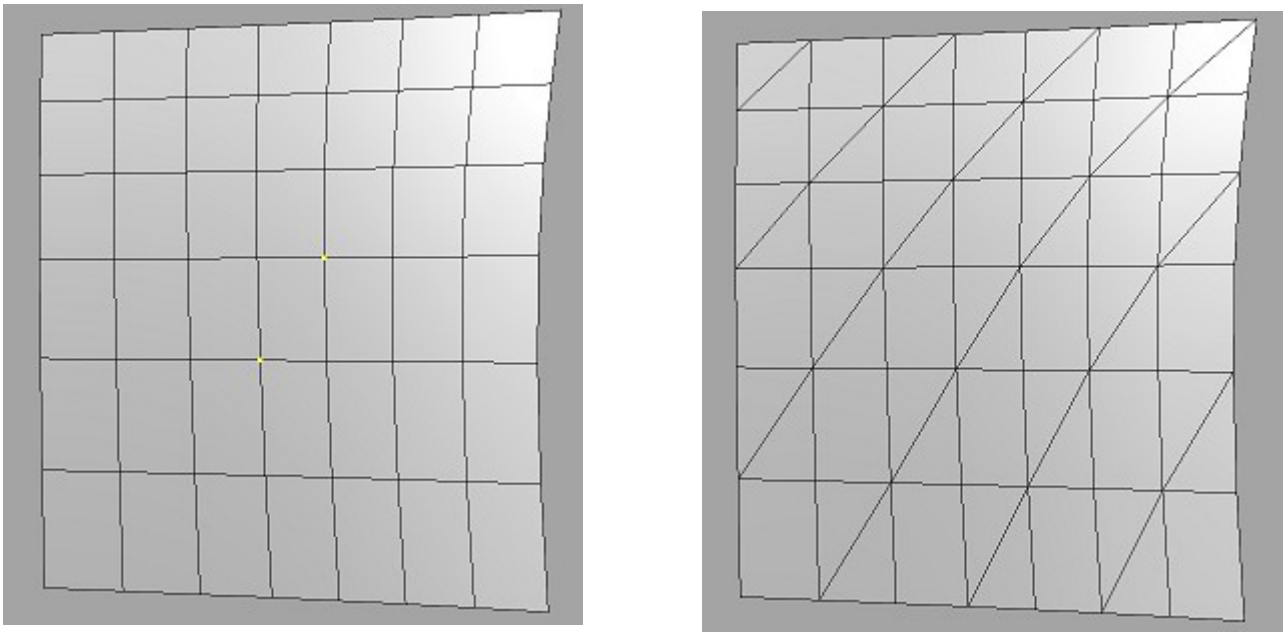


Fig. 12 *– An example of Add Diagonal Lines Parallel with Interval of 1.*

7.3.4 MeshCut

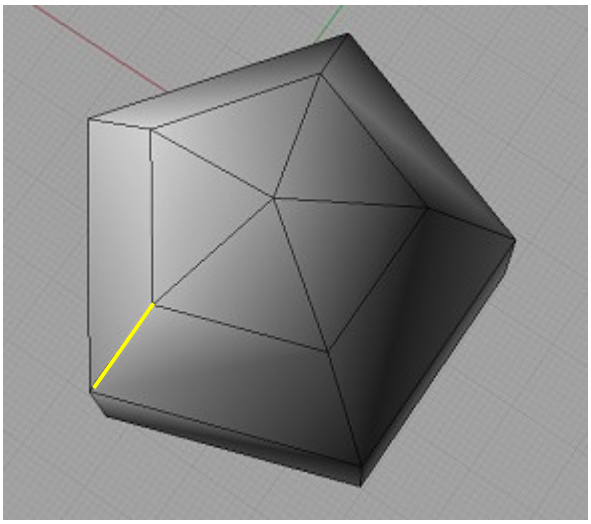
Rather than split a face from a vertex to a vertex like the tools mentioned above, *'_etMeshCut'* divides a face or row of faces in half from the midpoint of an edge to the midpoint of the edge opposite. This command takes two mesh edges as input. If a row of quadrilateral mesh faces connects these edges, the faces along the row are cut in half by inserting new edges starting from the midpoints of the input edges. The command has two options for operation:

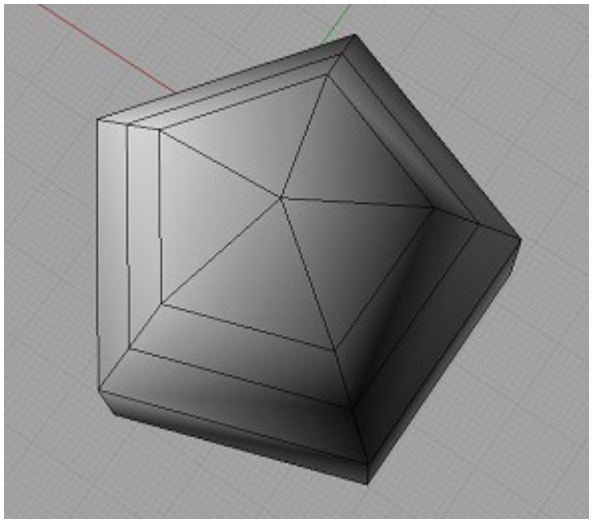
By default, the option AcceptNgons is set to '**false**'. This causes EvoluteTools to split the quadrilateral mesh faces at both ends of the newly inserted cut into triangles. This is done because otherwise these quad faces would contain 5 mesh vertices and therefore, technically, become pentagons. As Rhino does not support mesh faces with more than 4 vertices, this is the only way to facilitate such a mesh cut and still create a standard Rhino mesh object. EvoluteTools supports the editing, creation and manipulation of Polymeshes, however, if a Rhino file containing a polymesh would be loaded on a machine without EvoluteTools installed, the polymesh will then be displayed triangulated.

If the option AcceptNgons is set to '**true**', the triangle split is not performed. This usually leads to the creation of a polymesh object containing pentagons. These objects are not supported by Rhino and not fully functional in this version of EvoluteTools for Rhino.

7.3.5 Loop Cut

The *'_etMeshLoopCut'* command functions very similar to the MeshCut command. However, rather than selecting two edges to cut between, you only need to select one edge. Then the cut is continued in both directions until a non-quad face or the mesh boundary is reached. This may lack the precision of MeshCut, but it avoids creating ngons and all of the associated problems. Personally, I use LoopCut quite frequently to add face density and detail to meshes. It's a very handy tool!



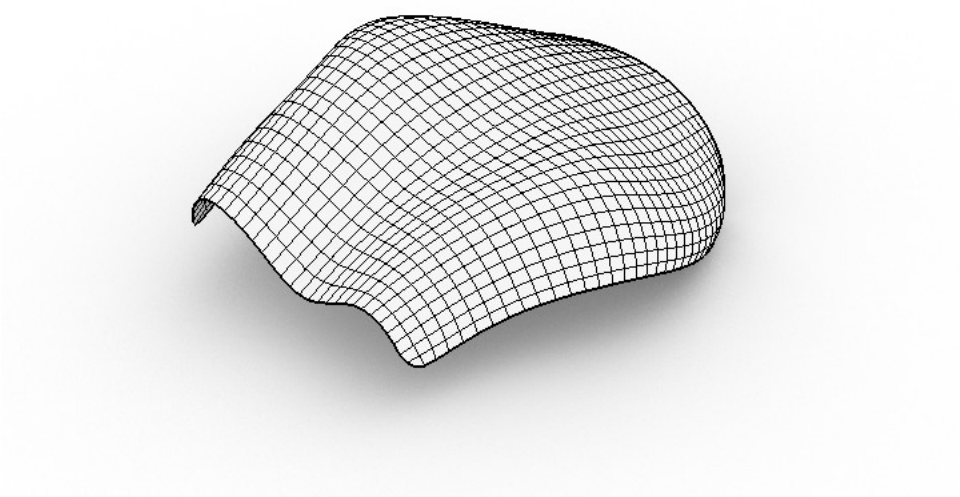


7.4 Subdivision Examples

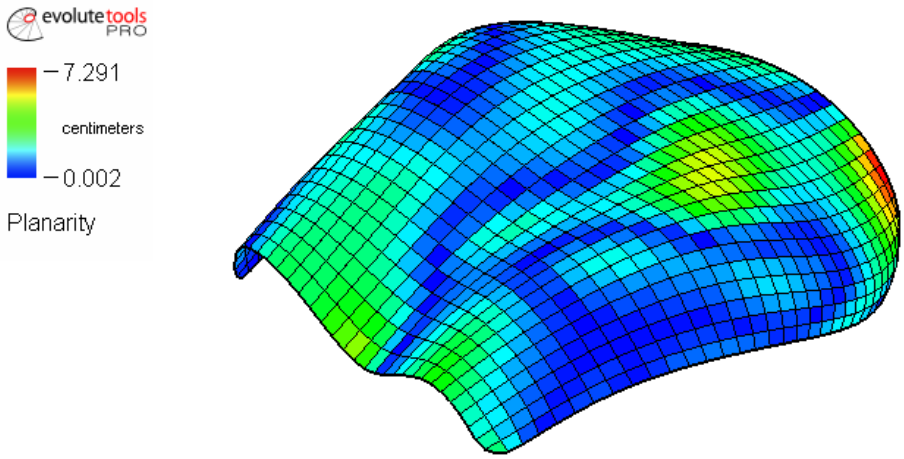
Now that that is out of the way, it is more than past time to get hands-on with some examples. In the first example, you will use the Dual Edge subdivision and the Add Parallel Diagonals tool to create a planar panel layout from a quad mesh without any optimization step. In the next example, we're going to get creative and use combinations of subdivision routines to make interesting patterns in a panel layout. Let's get started!

7.4.1 Example 6 – Planar Panels the Quick and Easy Way

1. Open the file called Primer_Example6. This file contains a familiar shape. It is quad mesh that approximates the shape from Example 1. If you have saved your result from Example 1, you can use that to complete this example.



In addition to providing new ways to create and modify meshes, EvoluteTools also provides new ways of looking at them. We are going to use one of those tools now. Select this button:  or type *'_etAnalyzePlanarity'* in the command line. If you didn't have the mesh selected before, the command line prompt will ask you to select a mesh. Select the mesh.



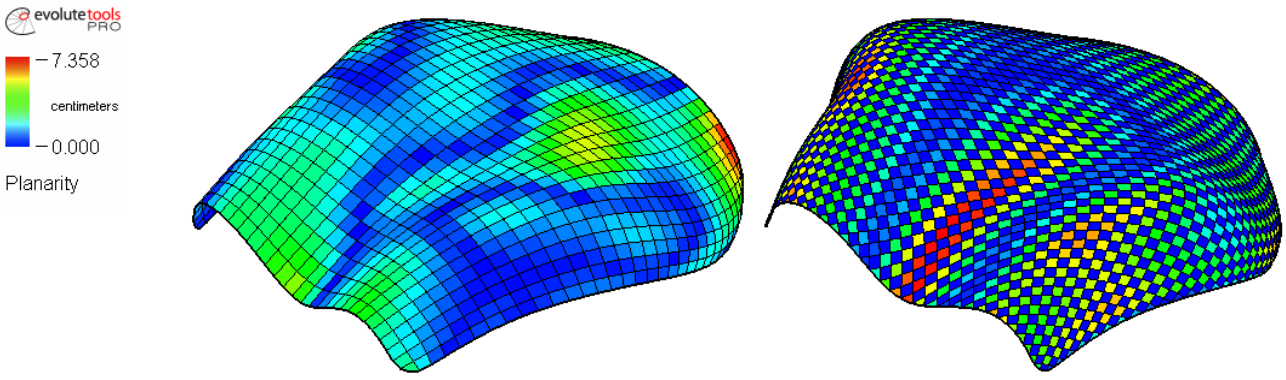
You will now see something like the image above. EvoluteTools has calculated how planar each face is [see Chapter 8.2.9 for details] and colored the faces according to the scale provided in the pop-up window. In general, the more red a face is, the more out of plane it is and the more difficult and expensive it will be to manufacture a panel to fit in that space. All-in-all, this mesh is not too bad, but there are still a lot of areas with non-planar faces.

Let's see if the Dual Edge subdivision can help us out.

2. Next, select the mesh again and then click the Subdivide button  . Then make sure you change the rule to "DualEdge" and hit enter. This will create a new mesh, but it will be overlaid with the original and not yet added to the Analyze Planarity visualization.

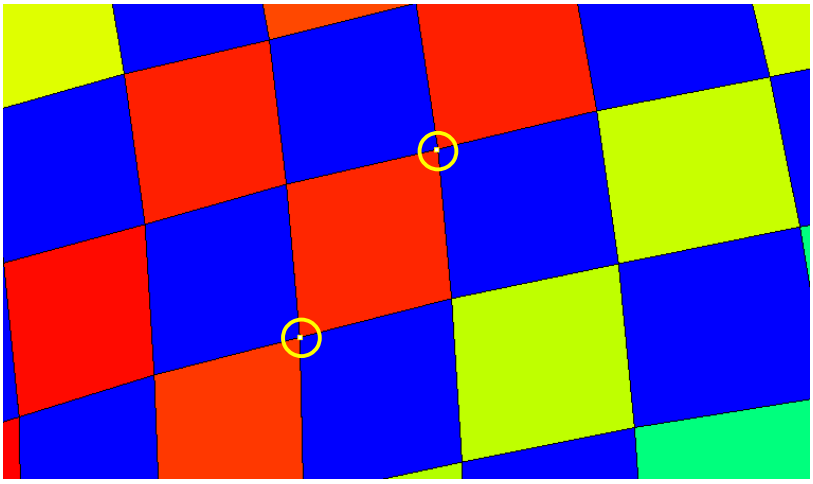
3. To compare the difference in the two meshes, drag the selected original mesh to the side.

4. Then select the new subdivision mesh and click the "Add Mesh" button in the Mesh Analysis dialogue. Now both meshes should be colored as below:

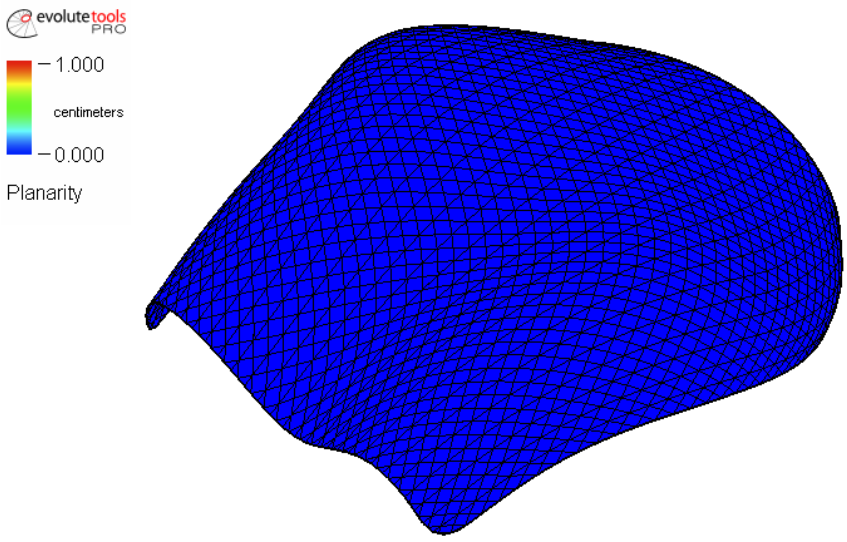


Notice how there is an alternating pattern of dark blue faces on the new mesh? These panels are perfectly planar. Sure some of the others are less planar than the original, but we will fix that in the next step.

5. To “planarize” the other faces you will divide them into triangles and since every triangular face is planar (3 non-linear points define a plane) the resulting mesh will be entirely planar. To do this you will use the *‘_etMeshAddDiagonalLinesParallel’* command, button  . Then select any two opposite vertices on a face that isn’t dark blue.



6. The result should look like the image below:



7. Just like that your entire mesh is made of a completely planar quads and triangles! Now, you may think that using triangular panels is a bit like cheating. What if you want a planar mesh that is only made from quads? Or hexagons? In the next chapter we will cover how to use the Optimization tool to make any kind of panel planar(ish).

7.4.2 Example 7 – Patterning with Combinations

1. Open the file called Primer_Example7. This should appear empty. Whatever you do, do NOT reveal the layer called Patterns!!

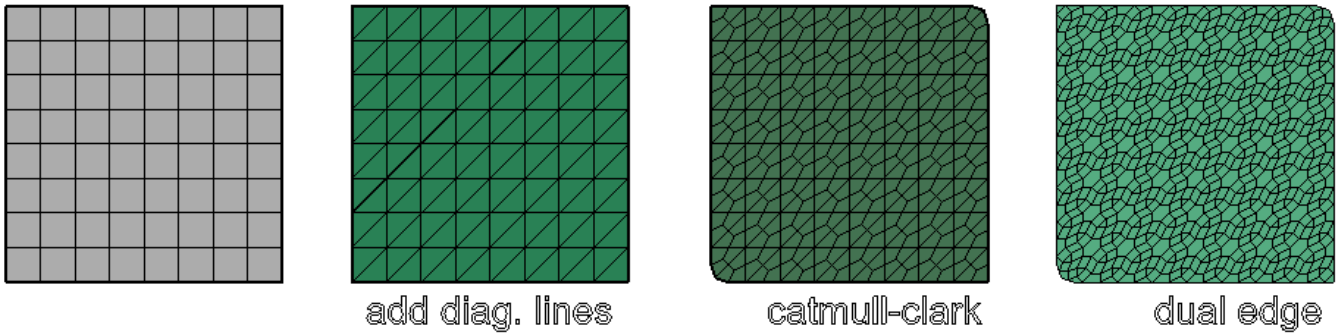
You did it, didn't you? Well, I'm not surprised. This collection of flat meshes show just a few of the patterns you can make by combining the various subdivisions. Now we will create another to add to this collection.

2. Start by copying the 8x8 quad mesh (on sublayer Starting Grid) to a location somewhere above it.

3. Now turn it into a diagonal mesh by using the *'_etMeshAddDiagonalLinesParallel'* command with an interval of '0'. For this example, it doesn't matter which direction you create the diagonals. It will only mirror the pattern.

4. Your mesh won't stay triangular for long. Use the 'Catmull-Clark' subdivision to turn in back into a quad mesh.

5. Interesting, but not quite done. Run a 'Dual Edge' as your last subdivision step. Now you should have a really cool tiling of quads, triangles, and hexagons.



Important Disclaimer: At this point, I would encourage you to go crazy and start experimenting with wild subdivision patterns. And you should, **but** do so with this word of caution. Rhino currently does not support poly-meshes (i.e. meshes composed of polygons other than quadrilaterals or triangles). EvoluteTools gives you the ability to create and manipulate these meshes within Rhino. However, certain very complicated poly-meshes or those with many sided polygons will cause Rhino to crash upon opening the file. This means everything will appear fine until you try to reopen the document. So, if you are planning to use complicated mesh patterns, please save your file before subdivision and save the file with a **different name** afterwards. This should be resolved in a future release, but at the moment it is better to be safe than sorry.

8. The Optimization Chapter

You've made it at last (or just skipped ahead) to the Optimization chapter. In the very first example, way back in Chapter 4, you saw EvoluteTools work its „magic." The truth is, of course, that there is nothing magic about it. EvoluteTools contains an optimization engine that iteratively alters the geometry of a mesh, compares the result to your target geometry and the measurable constraints you set, and then depending on this comparison approves the changes and continues making alterations.

What makes the optimization approach of EvoluteTools so powerful is that you can specify as many constraints as you want and let the optimizer sort out the solution. Additional constraints can be added at any time without having to redesign the layout. By controlling the importance of each constraint you can tailor the optimization to the specific requirements of your project.

You can think of the optimization process as a multi-directional game of tug-of-war. Each constraint is a contestant pulling on a rope and their strength is defined by the importance value that you, the user, assign. At the center of this struggle is your poor mesh being pulled this way and that until the optimization engine finds the solution that best satisfies all the constraints according to their importance. The real art of using EvoluteTools is not to understand how each constraint will affect the mesh individually, but how to find the appropriate balance between them to achieve the result you want. For some more detail and musings on this, see Chapter 9 - Getting the Most Out of EvoluteTools .

There are three groups of tools that will affect how the optimization is carried out: Optimization Options Importance or what I like to call „the global constraints", optimization toggles, and the local constraints. Below I will discuss each in turn, however in practice these constraints are almost always used together. This is a **long** list of options with odd names and occasionally convoluted functionality. I've tried to add context to keep things interesting, but this section is probably better used as a reference than read from top to bottom.

If you are more interested in action rather than theory or just have a short attention span, I'd suggest you read the next section to give yourself a general idea of what's going on and then skip to the Optimization Examples at the end of this chapter.

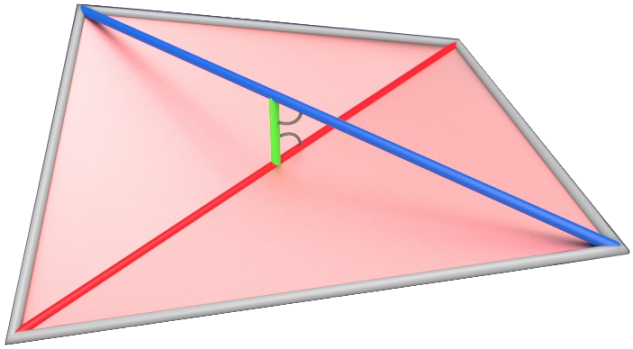
8.1 A Brief Note on Optimization

Moving forward, it will be good to have a more specific idea about what we mean when we are talking about optimization. To that end, I will rough out a short, general definition and then discuss what this has to do with EvoluteTools. Optimization is, in general, the systematic process of maximizing or minimizing a function. Good. Clear as mud? Let's break that down a bit.

First, *"systematic process".* This is the repetitive techno-wizardry that the software handles. In terms of EvoluteTools, it only means that there is some nifty logic to how the mesh is altered between iterations. There is an algorithm that takes the mesh and finds a "slightly better" variation. It does this repeatedly, in a very clever way, until a solution is found. The specifics aren't *so* important for an end user. The really important question is what does it mean for my mesh to get "slightly better"? In what way?

This brings us to the next bit: *"maximizing or minimizing a function."* In terms of EvoluteTools, the function is a mathematical way of describing the aspect (or aspects) of your mesh that you care about. For example, if you are interested in finding the shortest person in the room, you are only interested in one aspect of a person: their height. This doesn't describe an entire person, but it is enough to find the shortest person. It is important that the aspect you are trying to optimize be **measurable and quantifiable**. For instance, there is no mathematical way of describing the niceness of a person. Finding the nicest person in the room is not possible with such computational methods (but the answer is usually: me).

Each of the settings in the Optimization Options Importance dialogue represents a single measurable and quantifiable aspect of your mesh. The mathematical representation of this aspect is a function. Let's get concrete. Planarity of a mesh face is measured by the shortest distance between its diagonals [see image right].



This distance is measurable and quantifiable. The function for the total planarity, $\mathbf{P}_{total}$, of a mesh with some number of faces, N, can be represented as the sum of this distance for each face in the mesh.

$$\boldsymbol{P}_{total} = P \text{ of face } 1 + P \text{ of face } 2 + P \text{ of face } 3 + \ldots + P \text{ of face } N$$

The *lower* the number is the more planar the faces in a mesh. For this function, "slightly better" means a slightly lower value for $\mathbf{P}_{total}$. An optimization of this function would be trying to minimize it.

As I mentioned earlier, optimization in EvoluteTools works a bit like a game of tug-of-war. That's because you will generally be interested in more than one aspect. Therefore, you will have more than one function. How does the optimizer handle all these functions simultaneously? It multiplies the outcome of each individual function by the importance weight you set and then adds them all together. This is called a weighted function.

*Overall Optimization Function = [weight 1] * [function 1] + [weight 2] * [function 2] + … + [weight n] * [function n]*

When you set an option importance weight to zero you are telling the optimizer to disregard that aspect entirely. The higher the importance weight you set, the more that element contributes to the eventual solution.

In the following sections, I will predominately be describing the effect that a given constraint has on a mesh during optimization. However, where it is particularly enlightening or at least convenient for me, I will explain in general terms the function that EvoluteTools uses in its calculations.

8.2 Optimization Options Importance – Global Constraints

Below, I will describe the functionality of each Optimization Option individually, but you should read with the understanding that these options are meant to be used together. Moreover, it will not always be clear what values you need to set. This will come through experimentation and experience.

To access the Optimization Options Importance, click on this button:  in the EvoluteTools toolbar or type '*_etOptionsImportance'* in the command line.

8.2.1 Surface Closeness

In the vast majority of cases, this will be a standard setting and by default it is set to 1. This constraint will try to keep the vertices of the mesh as close as possible to the reference surface. The actual function in the algorithm sums the measured distance from all the mesh vertices to the target geometry and then tries to minimize this value.

This is an important constraint, because it gives a basic underlying order for the location of the vertices in space. If this importance becomes to low with respect to some of the others, the mesh may develop unwanted ridges or fold in on itself. If matching the precise geometry of reference surface is not as important to you as other constraints, I would suggest raising the other constraints gradually as you optimize rather than lowering this value.

8.2.2 Curve Closeness

This optimization setting seeks to minimize the distance of boundary vertices to the nearest boundary of the reference surface. Similar to the Surface Closeness function, the algorithm measures the distances between the edge vertices and the edge of the surface and then adds them all up. Then this value is minimized. By default, this setting is preset to 1.

By default, the optimizer automatically determines the boundaries of the mesh and the reference surface [See 8.3.6 - DefaultBoundaryCloseness]. However, this will affect only the boundary vertices of the mesh that are within a certain distance of the reference boundary [see 8.3.8 – CutOffCurveDistance «»].

Optimizations that have more rigid restrictions on the geometry of the panels (e.g. Ballpacking, Ideal Edge Length) may benefit from relaxing this term or turning it off altogether. It will allow the mesh to pull away from or overflow the boundaries to fulfill the other requirements. After these have been met, additional faces can be added or trimmed to match the boundary better.

With EvoluteTools PRO for Rhino the user can designate additional mesh vertices as curve points using *'_etSetVertexCurvepoint'*, and reference curves using *'_etSetReference'*. This can be used to align interior mesh vertices along predefined curves.

8.2.3 Original Closeness «»

Original Closeness minimizes the distance of vertices to their original position. In other words, the optimizer minimizes the change of vertex coordinates.

This is a term that you should reserve for final adjustments as it will allow only small changes to the mesh. This could also be useful in a form-finding process where you have no reference surface.

Caution: When using the Original Closeness term the current mesh that defines the original position of the vertices is the mesh selected when the *'_etOptimize'* command is run. Therefore, it is important to increase the number of iterations [see section 8.3.5 below] so that the optimization runs until it converges. Otherwise, the goal will be continually shifting and the result may drift from what you intended.

8.2.4 Fairness Springs

This is an interesting constraint that tries to minimize the edge lengths. In a sense, it mimics what would happen if all edges of the mesh were springs. This has the general effect of “tightening” the mesh. It can be useful when the other fairness terms are causing mesh polylines to bunch together. It has a very strong effect on the mesh, so I would advise you to increase the importance value in small increments. When Surface Closeness is set to 0 or there is no reference surface, this will produce a result approaching a minimal surface.

8.2.5 Fairness Curvature

Fairness Curvature is another setting that you will find yourself using fairly regularly. It is used to minimize the visible kinks in the mesh and giving it a smooth appearance. If a mesh is viewed as network of polylines, this constraint tries to make them as straight as possible. To put it simply, in a lot of cases this just makes your mesh look better!

Imagine you have a lumpy, curvy surface and a length of string that you are holding at either end. Your task is to lay this piece of string on the surface and at the same time keep it as straight as possible. Where will you lay it? To start with, you would probably quickly learn to avoid peaks and valleys, as well as areas of high curvature. The fairness curvature term is trying to achieve the same task with all the polylines of the mesh. This can lead to a lot of movement in your vertices and polylines, which is an excellent segue into the next optimization setting.

8.2.6 Fairness Curvature Variation

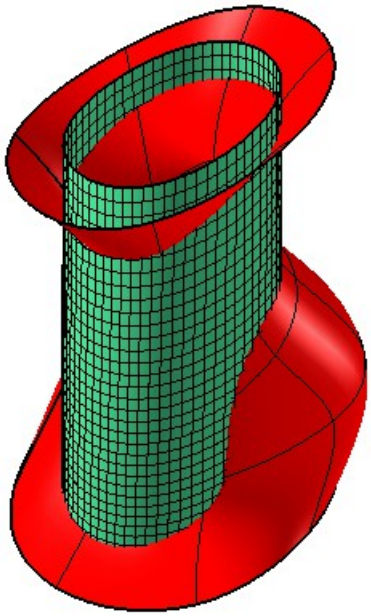
Instead of trying to straighten the polylines as the Fairness Curvature setting does (by reducing their curvature to 0), this term tries to keep their curvature constant - ideally circular. “Ok,” you may being saying to yourself, “that’s nice, but when do I need this?” The short answer is when you have a geometry that isn’t well suited to Fairness Curvature, which of course begs another question: “what kind of geometry is that?”

As mentioned previously, Fairness Curvature will avoid areas of high curvature. In geometries that have a generally cylindrical form, there is always one direction that has a significant curvature. In these cases, Fairness Curvature may begin to move the mesh polylines in undesirable ways as it attempts to straighten them. Fairness Curvature **Variation**, on the other hand, doesn’t care if a polyline is curved, it only tries to even out that curvature so that it is more uniform (like a circle). Using this term instead should result in less twisting, torquing and movement of the mesh.

Let’s run a quick comparison:

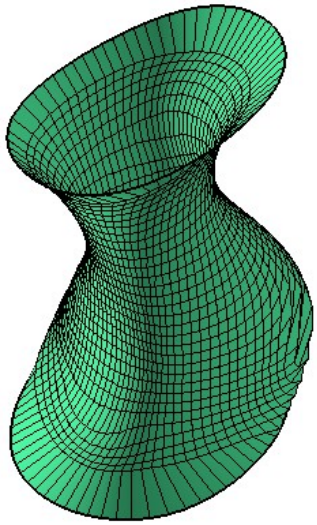
8.2.6.1 Example 8 – Fairness Curvature vs. Fairness Curvature Variation

Open the file called Primer_Example8. This file contains the cylindrical reference shape from Chapter 6.1.2-Example 3 and a subdivided mesh that is just itching to be optimized. In this example we will take a closer look at why you used the settings you did in that previous example.

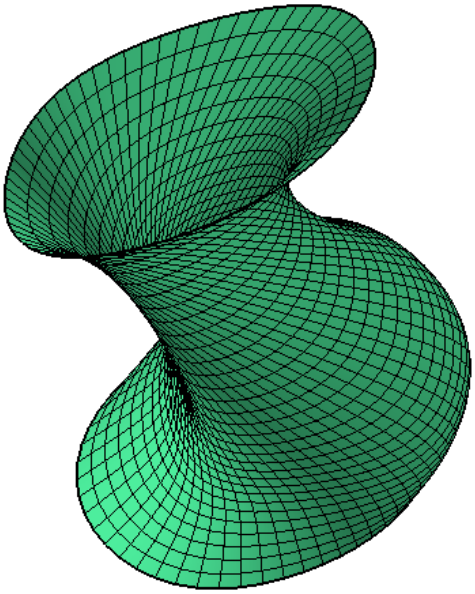


1. To begin, make sure that you settings are restored to the default by clicking  or typing '*_etOptionsReset*'.

2. Next, click optimize once. The mesh will snap to the reference surface, but not in such a nice way. You should also notice that the lines in the mesh are visibly kinked and wobbly. [see image] This is because we haven't used any Fairness terms yet to smooth this out.



3. Try to fix this by setting the Fairness Curvature to '1' and running the optimization a few times. Note: this is higher than you would normally set a fairness term, but it will make the result more noticeable.



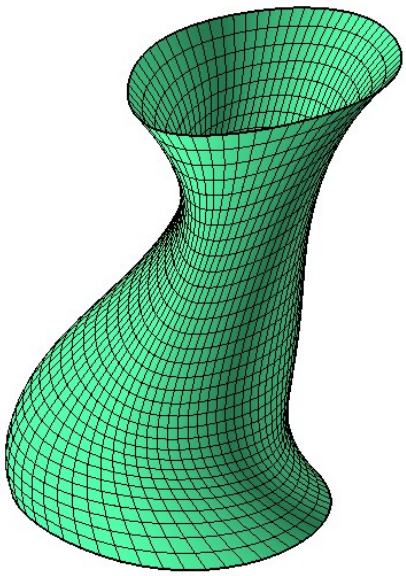
4. After a while you should notice that the vertical polylines in the mesh begin to twist around the shape [see image left]. For those of you familiar with hyperboloid structures, the reason for the twist should be clear. If not, I strongly encourage you to look them up and while you are at it Russian engineer Vladimir Shukhov.

5. If this twist is not desirable, you need another solution. Undo the optimizations you ran, or open the file again without saving.

6. Make sure you restore the settings to default and then this time set the Fairness Curvature Variation to '1' and run the optimization.

7. Your result should look like the image below. Notice that this time the kinks are removed but there is no twist. However there is some significant curving towards the base and there are some skewed panels. [image below]

8. Now add in a bit Fairness Springs optimization. However, if you set Fairness Springs to '1', it will pull the mesh back from the edges significantly. Try something like '0.4'.



Note: In practice, the question of which fairness term to use is not cut and dry. It will depend greatly on the properties of your surface. In the end, some combination of the 3 may be required. For example the mix you used in Example 3. Three tips: *Experiment. Experiment. Experiment.*

8.2.7 Ballpacking «»

And now for something completely different: Ballpacking. This term is meant primarily for use on triangle meshes, however it does work on other types of meshes as well. In this description, I will be referring to the triangular case.

First, I need to explain the geometric concept of an incircle. An incircle or inscribed circle of a triangle is the largest circle that can fit inside a triangle. It is tangent to all three sides. [see fig. 13 below]

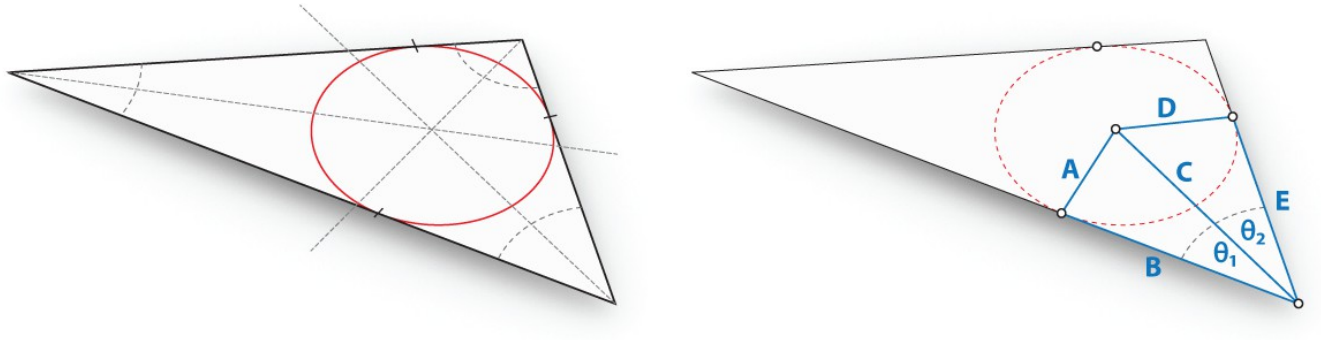


Fig. 13 *– Incircle Diagrams.* ***Left:*** *The center of the incircle lies at the intersection of the 3 angle bisectors.* ***Right:*** *The distance from a vertex to the tangent points on its edges is constant. Edges A & D are both radii of the incircle, thus A = D. Edge C is shared and bisects the vertex angle. Therefore,* $\vartheta_1 = \vartheta_2$ *and B = E.*

When the Ballpacking constrain is set, triangle meshes are optimized such that the incircles of neighboring triangles touch tangentially. As you can see in the diagram above, the distance from a vertex to the tangent point of the incircle is constant. When the triangular face is part of a ballpacked mesh, this distance from the vertex is constant for all the faces intersecting that vertex. In the image below: $\mathbf{r}_1 = ||V_1 - t_{14}|| = ||V_1 - t_{13}|| = ||V_1 - t_{12}||$ Viewing $\mathbf{r}_1$ as radius of a sphere centered on the vertex, we arrive at a sphere packing: Adjacent spheres $\mathbf{S}_i$, $\mathbf{S}_j$ touch at $\mathbf{t}_{ij}$. This is also called a ballpacking [see fig. 15 below]. This results in visually balanced meshes. In addition, this condition can lead to an important architectural property: namely, torsion-free nodes.

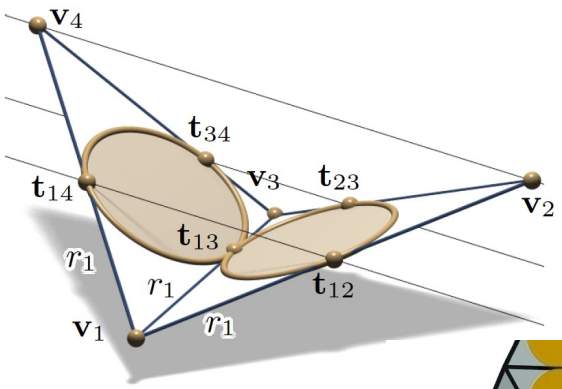
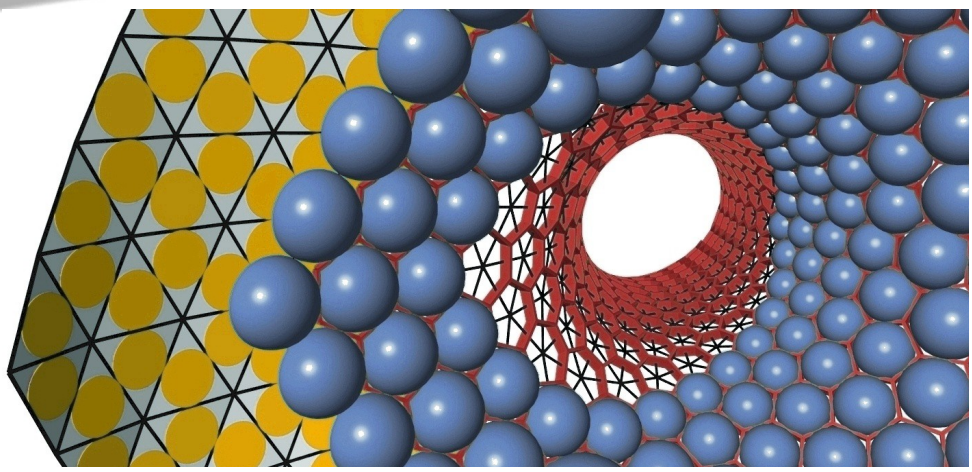


Fig. 14 *– Adjacent incircles in a ballpacking mesh.*

Fig. 15 *– An example of a ballpacked mesh with incircles (yellow), the resulting spheres (blue) and a hexagonal grid derived from connecting the circle centers (red).*



If we physically realize a mesh and let beams follow its edges, the intersection of beams near vertices is an important issue. It is desirable that symmetry planes of beams nicely intersect in a common node axis that passes through a vertex. Such beams are called a support structure with torsion-free nodes.

This is easy to accomplish if we move from a regular ballpacked triangle mesh to a hexagonal mesh with a Dual subdivision and is exactly what we'll do in the example in Chapter 8.5.2. In this new mesh, node axes are the axes of the previous incircles. Moreover, neighboring incircle axes lie in a common plane, orthogonal to the common tangent of those incircles. Thus, that common plane corresponds to the plane of symmetry of a beam running along the edge connecting those vertices and the nodes are torsion-free. See the scientific article "Packing Circles and Spheres on Surfaces" on Evolute's homepage for details.

Tip: You can extract the balls from a ballpacking mesh with the command *'_etBallPackingMeshExtractBalls'*. You may find these two commands helpful immediately afterward: *'_SelLast'* and *'_SelDot'.*

Bonus Tip: As an alternative to the Dual Subdivision, a ballpacked triangle mesh is also a great opportunity for the TriHex subdivision.

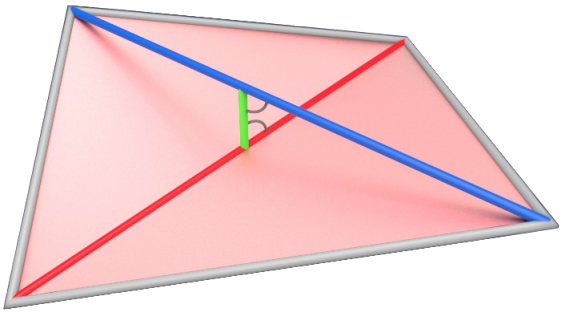
Pro-Tip: Included within the EvoluteTools directory there is a folder called "ScriptingExamples. In that folder there is a RhinoScript file called "Incircles". Running this script and selecting a mesh will extract all the incircles onto a new layer. If you want to find out more about how RhinoScript can be used in conjunction with EvoluteTools check out Marko's EvoluteTools Scripting Primer!

8.2.8 Coplanarity

This is the setting for the importance of the local optimization constraint for coplanarity. See Chapter 8.4.7-SetVerticesCoplanar for details.

8.2.9 Planarity

We've already discussed this constraint in section 8.1 – A Brief Note on Optimization. To summarize again, it optimizes the mesh for planarity of its faces. The planarity is measured as the shortest distance between the diagonals of a face. This will be a very important term for most projects as planar components are much less expensive to produce.



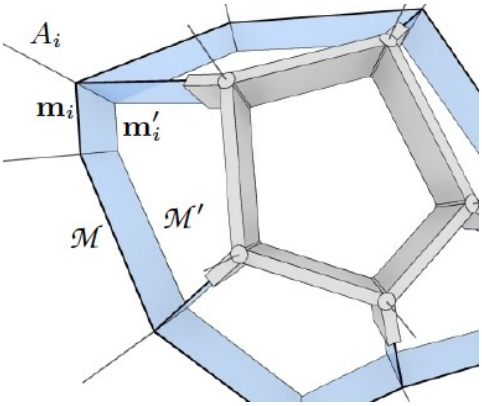
8.2.10 Conical and Circular Optimizations «»

When the Conical term is given an importance value, the mesh is optimized to become a conical mesh and when the Circular term is bumped up, the mesh is optimized to become circular. Good. Moving on!

No? Are you sure you want to know more? Ok. Time for another healthy dose of geometry, but you asked for it this time. All kidding aside, while the concept of a conical or circular mesh maybe difficult to wrap your head around at first, its implications for architectural construction are well worth appreciating. Stick with me for minute and you'll be glad you did.

The primary purpose of each of these terms is to create clean parallel mesh offsets. This is of crucial importance to architecture, because structures are multilayered. A mesh by itself is only an abstract mathematical construct that, at best, represents a single layer of actual construction. In order to design real structures from abstract meshes, the offset is a necessity.

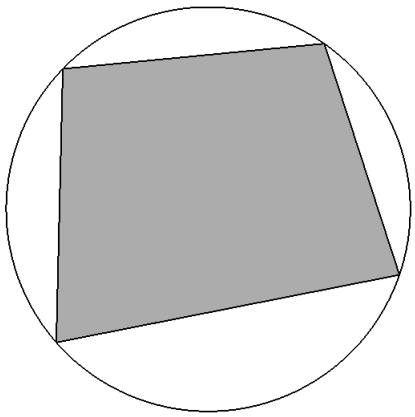
In the realization of a mesh, nearly planar panels (for lower cost) are held by beams (with some standard cross sectional shape) following the edges of the mesh. Beams are generally symmetric around a central plane. This central plane intersects the edge of the mesh and, ideally, the node axis [i.e. torsion-free nodes – see also section 8.2.7-Ballpacking «»] . The node axis is a line that connects corresponding vertices on a mesh and its offset. [see the image right]



The definition of the offset when dealing with a smooth surface is pretty intuitive really: the resulting surface that is at a constant distance along the surface normal of each point. Unfortunately, it is not that straight forward for a mesh. There are several meaningful definitions of a mesh offset and they don't all have the same properties. There are two that are relevant to EvoluteTools:

- **Vertex Offset:** The distance of corresponding vertices between the mesh and its offset has a constant value. Meshes with this property are called “circular.”
- **Face Offset:** The distance between corresponding faces is constant. Meshes with this property are called “conical.” Conical meshes also have torsion-free nodes!

Not every parallel mesh offset of an arbitrary mesh will these properties. These two are of particular importance, because you can use EvoluteTools to optimize your mesh so that it will have these properties. I won't discuss the other type of mesh offset, Edge Offsets, here, but for more detail see this fascinating paper by Helmutt Pottman, et al..



It is outside the scope of this primer to discuss the exact reasons that a circular mesh has the vertex offset property (it has to do with discrete Gaussian images and a nice concise description can be found in Architectural Geometry*).* It may help to understand the basic property that is being optimized for, however. A quad mesh is called circular if each of its quads possesses a circum-circle [see the quad in the diagram left].

Conical meshes are called conical when all of the faces that meet at a vertex are tangent to a cone of revolution through the axis of that vertex [fig 16a]. As you can see in Fig. 16b below, the axes of two connected vertices in a conical mesh lay in a common plane. Therefore, any beam along that edge will have a plane of symmetry that coincides with those axes and a construction derived from this mesh offset pair will have torsion-free nodes. Conical meshes are pretty slick!

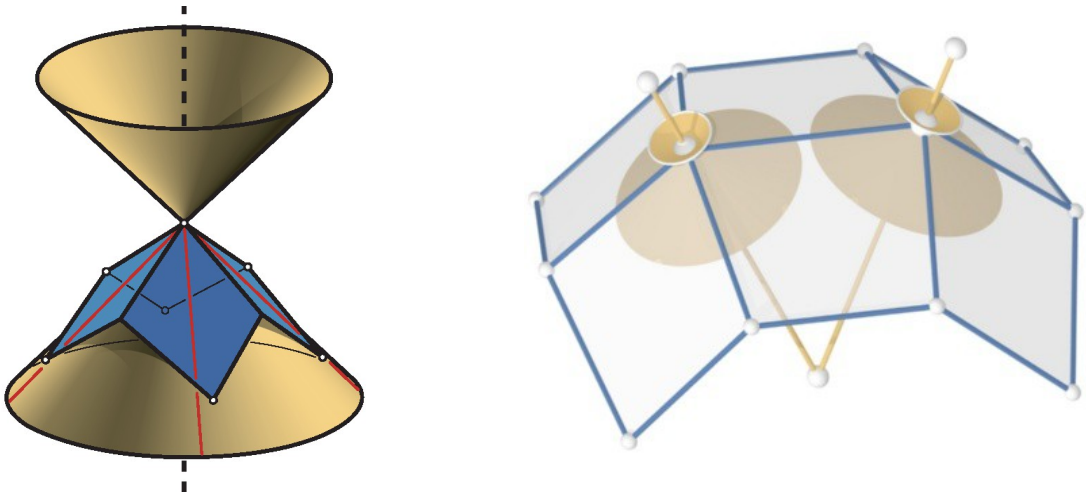


Fig 16 *– a)* ***Left:*** *4 faces tangent to a cone of revolution through their shared vertex. b)* ***Right:*** *Two neighboring vertices in a conical mesh have axes that lay on a common plane.*

8.2.11 Normal Closeness «»

This optimization term minimizes the deviation of vertices from their current vertex normal. In other words, it tries to restrict vertex movement to moving along the vertex normals. This is not necessarily a global optimization constraint. You must first select the vertices you want to restrict with the *'_etSetVertexRestrictedToNormal'* command,  . If you want this to affect the entire mesh globally, you must simply select all the vertices.

This is a useful tool for controlling the amount of movement of certain vertices. Sometimes it is too restrictive to fix a large group of vertices with *'_etSetVertexFixed*'. This will not allow them to move at all. With the normal closeness term you can allow the optimizer to adjust points into and out of the mesh without moving them across the surface.

It is only available in EvoluteTools PRO for Rhino.

8.2.12 Ideal Edge Length «»

Similar to the Normal Closeness parameter, Ideal Edge Length does not necessarily affect the entire mesh. The optimization will ONLY affect the edges that you have flagged with the *'_etSetEdgeLengthOptimization'* command,  . You can, of course, select all the edges if you wish.

It optimizes the edge lengths to be equal to the factor of the same name, set in *'_etOptionsToggles'* [see section 8.3.9]. Alternatively, if no value is set in the Toggles, the algorithm will try to make all of the selected lines as equal as possible. The latter is useful if you have no preset length requirement, because the optimizer will find the best fitting length for your given mesh and target geometry.

It is only available in EvoluteTools PRO for Rhino.

8.2.13 Ideal Panel «»

This is a rather "delicate" optimization term that applies to triangle and quad faces. In triangular meshes, the shape of the faces are optimized towards being equilateral triangles with side lengths close to the value set in the *IdealEdgeLength* parameter in *'_etOptionsToggles'* [see section 8.3.9]. Quad faces are optimized towards squares of the same side length.

I use the term “delicate” because having a surface paneled with equally sized ideal panels is only possible in special cases. Giving too much importance weight to this term compared to others can cause unexpected results.

It is only available in EvoluteTools PRO for Rhino.

8.3 Optimization Toggles

Rather than change the relative importance of the optimization terms like the Options Importance settings, the choices in the Optimization Toggles dialogue modify the behavior of the optimizer itself. You can access these settings with the command *'_etOptionsToggles'*,  can be reset to their default values using *,_etOptionsReset'*,  .

8.3.1 Fairing

This options toggles between two settings that effect the calculation of the *'FairnessCurvature'* and *'FairnessCurvatureVariation'* optimization terms. The default is **'absolute'** and in this mode the fairness terms function as described in the previous section. *'FairnessCurvature'* minimizes curvature (i.e. mesh polylines are straightened) and *'FairnessCurvatureVariation'* minimizes curvature variation.

The other option is called **'relative.'** With this option selected the state of the current mesh is used as the reference for the fairing terms. In other words, when *'FairnessCurvature'* is set the optimizer will now attempt to minimize difference of curvature to curvature of *current* mesh rather than reducing it to 0. Likewise, when *'FairnessCurvatureVariation'* is set, the optimizer will minimize difference of curvature variation to curvature variation of current mesh.

This is useful for situations where you’ve got the smoothness and curvature of the mesh just the way you want it, but there are still other aspects you want to optimize, such as planarity. This will keep the general form of your mesh while giving the optimizer more freedom than the global Original Closeness constraint [see section 8.2.3].

Caution: When using the RelativeFairing toggle the current mesh that is used for reference is the mesh selected when the *'_etOptimize'* command is run. This means that the reference is *reset each time* you run the command. Therefore, it is important to increase the number of iterations [see section 8.3.5 below] such that the optimization algorithm runs until it converges. Otherwise, the goal will be continually shifting and the result may drift from what you intended.

8.3.2 FairingMeasureScaleInvariant «»

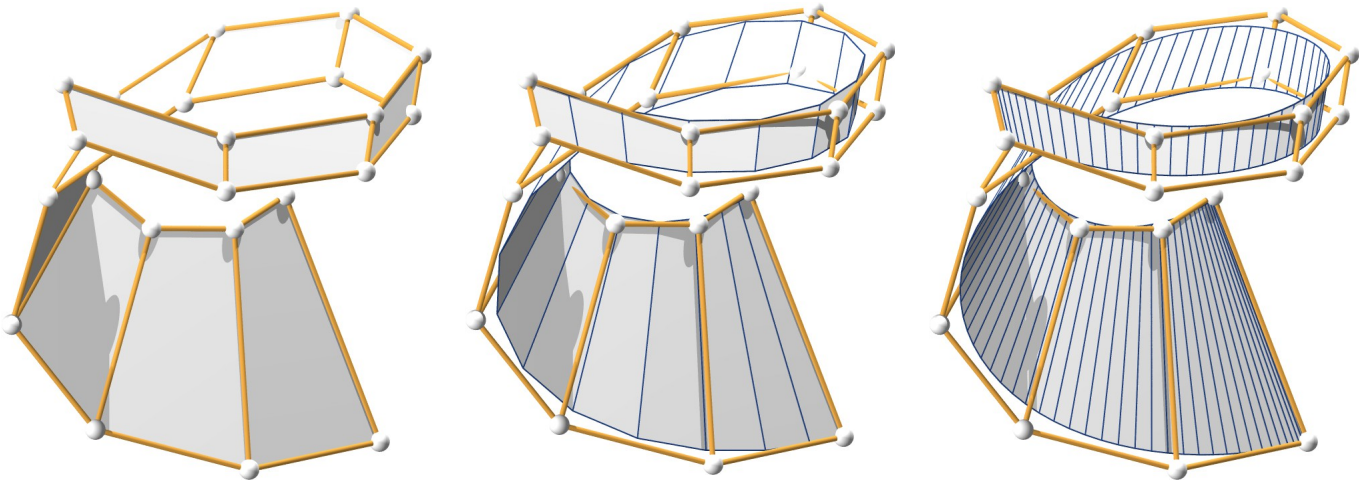
Like the Fairing toggle, this setting affects the *'FairnessCurvature'* optimization term. One side effect of the fairing terms is that they tend to concentrate vertices and edges in areas of higher curvature. When this toggle is turned on, this effect is somewhat reduced.

The default value is **'off'**. When this setting is off, the optimizer uses a scale-variant measure, which causes smaller edge lengths in highly curved regions.

When it set to **'on'** it uses a scale-invariant curvature measure. This causes more equally distributed edge lengths.

8.3.3 FairingMeasurePerEdgeScaling «»

This setting applies to fairing on strip meshes and should generally be switched on. As a strip is subdivided there are more rulings added, which overall maintain a similar average length. As they are added, the mesh polylines that define the strip are subdivided into smaller and smaller lengths. The length ratio of the rulings to the strip edges gets larger and larger. [see image below]



If every edge were to be treated equally by the fairness terms, more and more weight would be given along the ruling direction because of this increasing difference in their length.

When it is **'off'**, every edge is treated equally, using the general fairing importances. If it is **'on',** the optimizer uses individual scaling of the fairness term for single edges. This is used by several commands, among them by '*_etSubdivide*' for its strip subdivision rule.

This is only available in EvoluteTools PRO.

8.3.4 PlanarityMeasureScaleInvariant «»

This toggle applies to the Planarity optimization term. The default value is **'off'** and when this is set the optimizer will minimize diagonal distance of mesh faces as discussed previously. If this toggle is set to **'on'**, the optimization tool will minimize diagonal distance divided by mean diagonal length of mesh faces. In other words, as the face size increases so does the tolerance for deviation from planarity. This reflects the real physical property that large sheets of material can tolerate more deformation over their larger area.

This is only available in EvoluteTools PRO.

8.3.5 Iterations

This setting defines how many iterations will be done by the optimizer when *'_etOptimize'* command is run. Normally you will want to keep this value low so that you can judge the effect of your settings on the mesh and make adjustments if necessary. The lower the value, the more control you have over the process. However, as mentioned earlier, if you are using any optimization term or toggle that uses the current mesh state as an additional reference, you will want to increase this value because the current mesh state will be reset each time the command is run.

Caution: Increasing this value will cause *'_etOptimize'* to take longer before returning control to you, but overall optimization time until convergence will decrease.

8.3.6 DefaultBoundaryCloseness

This toggle affects the *'CurveCloseness'* optimization term. It is set to **'on'** by default and in most cases should be left on. When activated, all mesh boundary vertices will be automatically flagged as curve points [see also Section 8.4.1 - SetVertexCurvePoint]. This will cause boundary vertices to snap to their *closest* reference curve [see 8.2.2 - Curve Closeness]. Furthermore, the optimizer will automatically detect and add all reference surface boundaries as reference curves. This takes care of a lot of tedious work for you, but also takes away some amount of control.

When set to **'off'** the boundary vertices will not be automatically flagged and you must use *'_etSetVertexCurvePoint'* to configure which vertices should snap to the closest reference curve. Additionally, reference surface boundaries will NOT automatically be added as reference curves.This setting allows you to define the boundaries as you require. This can be particularly useful when working with ballpacking meshes.

8.3.7 CutOffSurfaceDistance «»

This setting defines the distance within which the *'SurfaceCloseness'* optimization term functions. Closeness optimization will only be done for vertices that are within this distance from one of the reference surfaces. Beyond this distance a vertex will no longer be considered in this calculation, although it will still be considered in any other active optimization terms.

By default the value is 1e+154, or 1 x 10^154. This is a very large number and unless things go seriously pear shaped, all vertices will be within this range. Normally, you will want to leave this alone. However, you may want to lower it if you are trying to blend a mesh between multiple reference surfaces or only using them as guides in a form-finding process.

8.3.8 CutOffCurveDistance «»

This option functions in a similar way to *'CutOffSurfaceDistance'* except it applies to the *'CurveCloseness'* term. It defines the distance from the reference curve within which curve closeness optimization applied to vertices marked as boundaries. If those vertices are beyond that distance, they will not be considered in the *'CurveCloseness'* optimization.

Similar to the previous toggle, the default is an extremely large number and should be left this way in most cases. However, if you only want to vertices to snap to an edge if it is within some range, this is setting you need to change.

This is only available in EvoluteTools Pro.

8.3.9 IdealEdgeLength

This value is used in conjunction with the optimization importance of the same name. When this setting is given a value larger than zero and the *'IdealEdgeLength'* optimization term [see section 8.2.12] is given some importance, selected edges will have their length optimized to be close to this value.

If this setting is left at the default of '0', the optimizer will determine a target edge length that best fits the mesh. I recommend this option if you have no particular panel size requirements. For instance, it is ideal if you are only trying to even out the edge lengths at an earlier stage of mesh subdivision.

Note: this only applies to edges selected with the *'_etSetEdgeLengthOptimization'* command,  .
[see section 8.4.6 below]

This is only available in EvoluteTools Pro.

8.3.10 StripFairingScalesRatio «»

This is a setting that is used in conjunction with the strip subdivision rule [see section 7.2.9]. The fairing importance strips (along the ruling direction) on a mesh edited with the strip subdivision rule is scaled down by this value, if 'FairingMeasurePerEdgeScaling' is switched on. If this value is zero, a scaling ratio is automatically computed using the average lengths of ruling and non-ruling edges.

8.4 Local Constraints

The local constraints, as opposed to the global constraints addressed above, affect only the components of the mesh that you select. In general, they are used for fine tuning a mesh and giving the user a higher degree of control over how the mesh will be modified.

General Tip: Any of the following commands that accept vertices as an input (i.e. any command name starting with ,_etSetVertex') will accept preselected vertices. This means that you can use any of Rhino's or EvoluteTools' selection methods (e.g. *,_etSelectMeshBoundary'*) to grab the vertices you want and then run the command. Alternatively, if nothing is selected, you will be prompted to select the vertices, but you will be limited to basic mouse selection.

General Tip #2: If you need to unflag vertices previously flagged vertices, run the command again and select vertices while holding CTRL (i.e. the typical Rhino selection/deselection mode).

8.4.1 SetVertexCurvePoint

The command *'_etSetVertexCurvePoint'*,  designates mesh vertices as curve points for optimization.Then during optimization, distances between mesh vertices flagged using this command and the nearest reference curve will be minimized, if the *'CurveCloseness'* optimization importance value is set. In other words, the selected will move towards the closest boundary curve. If *'DefaultBoundaryCloseness'* is switched on, all boundary vertices

will automatically be designated as curve points and save you the trouble of doing it yourself.

Additionally, if you have EvoluteTools PRO, this can be used to align interior mesh vertices along curves set with *'_etSetReference'*. This is rather useful if you want to align the panel layout with some predefined structural grid for your building.

8.4.2 SetVertexFixed

I've already mentioned a few ways to constrain how much a vertex can move around on your mesh, but if you really don't want that vertex to move at all you need to use *'_etSetVertexFixed'*,  . This command allows you to designate mesh vertices as fixed for optimization. Mesh vertices flagged using this command will not be changed at all by *'_etOptimize'*.

8.4.3 SetVertexCorner

The command *'_etSetVertexCorner'*,  designates mesh vertices as corners for optimization. During optimization, boundary mesh vertices flagged using this command will be excluded from all fairing optimization terms. This will prevent the optimizer from straightening the boundary edges around them and causing a bulging or rounded corner. If a corner should also remain at its exact coordinates during optimization, *'_etSetVertexFixed'* should be used in conjunction with this command.

The command has two options:

AutoDetect: If true, EvoluteTools automatically selects the edge vertices along the boundaries according to the kink angle between the boundary edges at those vertices.

AutoDetectKinkAngle: If AutoDetect is used, all boundary vertices with a kink angle higher than the value of this option will be selected.

8.4.4 SetVertexRestrictedToNormal «»

'_etSetVertexRestrictedToNormal',  restricts vertex movement to its normal. When invoked, it highlights the previously selected vertices.Vertices selected with this command will be constrained to move only along their approximated vertex normals during optimization, the strength of this constraint being defined by the importance setting *'NormalCloseness'*. This is useful if you have a mesh that already has the desired appearance, but its panels do not yet fully meet your planarity requirements. Setting *'NormalCloseness'* instead of fairing constraints and combining it with the planarity constraint will possibly improve the planarity of panels without significantly changing the appearance of the mesh.

This is only available in EvoluteTools PRO.

8.4.5 SetEdgeFeature «»

So far most of the tools discussed have been useful for smooth, flowing free-form designs. But what if your target geometry is a bit edgier? Or has an angular crease? That's where the set edge feature command comes in handy.

You can use the command *'_etSetEdgeFeature'*,  to designate mesh edges as creases. Using this command, you can flag mesh edges to be treated as creases during optimization. This prevents any fairing across the flagged edge. In other words, this permits sharp features within an otherwise smooth mesh. When invoked, the command highlights previously defined feature edges. These can be selected and unselected using several options:

SelectByVertices: This toggle allows selection of Polymesh edges by picking vertices one at a time.

SingleEdge allows you to flag a single edge by selecting it.

Polyline flags all edges in a polyline by selecting one edge.

ClearAll unflags all edges.

AddAll flags all edges.

Autodetect prompts for an angle and automatically flags all edges whose two adjacent faces intersect with a kink angle larger than the specified value. All other edges are unflagged.

Simply turning the fairing off across the crease does not mean that those respective vertices will automatically snap to your actual edge crease, in order to do that, it is necessary to define those vertices as additional curve points and define the creased edge int eh reference as an additional curve reference.

This is only available in EvoluteTools PRO.

8.4.6 SetEdgeLengthOptimization

You use the command *'_etSetEdgeLengthOptimization'*,  to set the mesh edges that you want to be optimized for Ideal Edge length [see section 8.2.12]. When you run this command, it will highlight any previously flagged edges, which can be selected and unselected using several options:

SelectByVertices this toggle allows selection of mesh edges by picking vertices one at a time.

SingleEdge allows you to flag a single edge by selecting it.

Polyline flags all edges in a polyline by selecting one edge.

ClearAll unflags all edges.

AddAll flags all edges.

Invert flips the current selection. Flagged edges become unflagged and vice versa.

This is only available in EvoluteTools PRO.

8.4.7 SetVerticesCoplanar

'_etSetVerticesCoplanar',  is a very useful tool for controlling the movement of vertices in your mesh. It is of particular interest to architectural field because it can be used to align groups of vertices with planar structural elements such as floor slabs. Generally, this tool allows you to flag sets of vertices that you want to become coplanar to a specified plane.

When run, the command shows previously defined coplanarity sets, if any, and allows you to browse through them. If there are no defined sets, it will prompt you to select at least 2 vertices to create a new set.The command has the following options:

Add allows you to add another set of coplanar vertices.

Remove purges the currently displayed set from the list.

Previous lets you switch to the previous set of vertices.

Next switches to the next set.

Planetype lets you specify constraints on the common plane:

GeneralPlane will not add any constraints. The optimizer will determine the best fitting plane.

ParallelToYZPlane, ParallelToXZPlane and **ParallelToXYPlane**, will optimize the vertices towards a plane parallel to the respective coordinate plane.

NormalToXYPlane, NormalToXZPlane and **NormalToYZPlane** will optimize the vertices towards a plane normal to the respective coordinate plane.

FixedPlane lets you define more detailed constraints on the plane:

SelectPlane asks you to select a planar surface and will fix the plane to that planar surface.

SelectPlaneNormal asks you to select a planar surface and will fix the plane's normal to the normal of the selected planar surface (translations of the plane will still be allowed).

HorizontalWithFixedZCoordinate will constrain the plane to be horizontal, and asks you to specify the height of the plane. This is a shortcut for SelectPlane using a horizontal plane.

Importance lets you specify the optimization importance value for the current coplanarity constraint. This importance value is multiplied by the importance value for Coplanarity.

ShowPlane adds a planar surface patch to the document, showing the current plane used for optimization.

ShowDistancesFromPlane adds text dots specifying the distances of the respective vertices from the current plane.

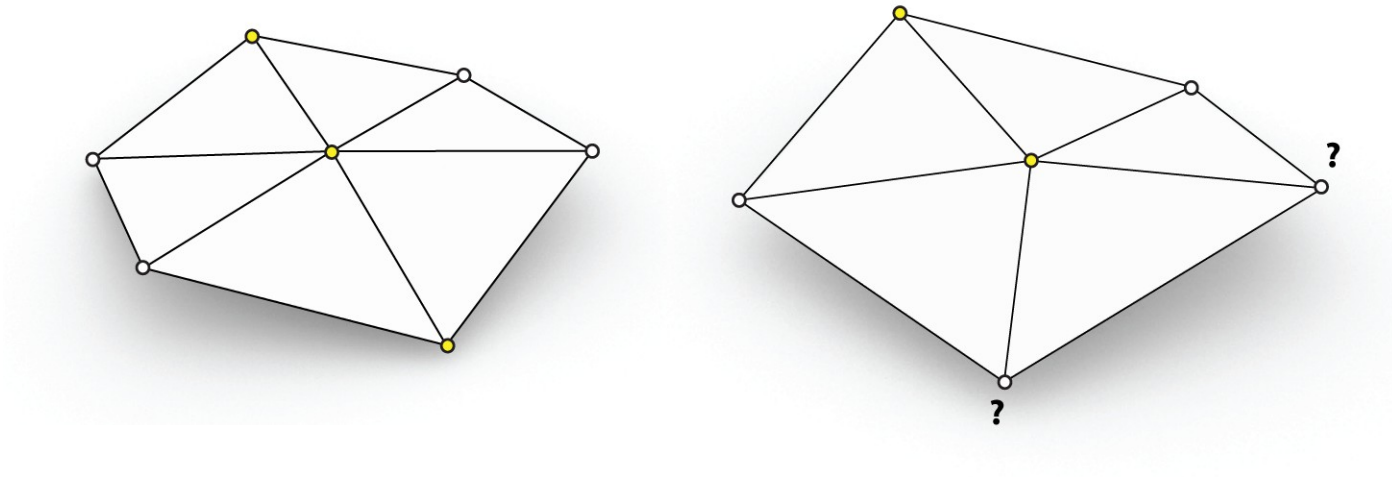
Vertex list coplanarity constraints will be preserved during changes to the connectivity of the mesh if possible. If a list of coplanarity-constrained vertices contains less than the number required vertices (varies depending on plane definition) after changes to the mesh connectivity, it will be removed.

This is only available in EvoluteTools PRO.

8.4.8 SetVerticesFairing «»

The command *'_etSetVerticesFairing'*,  allows you to select mesh vertices for additional fairness optimization (minimization of kinks). Mastery of this command will give you a very fine grained control over the fairing in your mesh and can be useful in a number of situations.

To better understand how to use this tool it is helpful to think of fairing has being calculated across groups of 3 vertices and the angle (i.e. kink) that they define. Let's also consider only a single central vertex at some number of intersecting edges. At vertex intersections with an even valence (i.e. an even number of incoming edges) the fairing optimization examines that central vertex combined with pairs vertices from the ends of opposite edges.



However, at vertices with an odd valence, the optimizer can't define opposite pairs and these are ignored in the fairing algorithm. [see fig above] However, you can use the *'_etSetVerticesFairing'* to manually select vertices. Additionally, at boundary vertices only other boundary vertices are grouped for fairing. If you wanted to smooth an interior mesh polyline into a boundary edge, you would need to use this command.

It can also be used to fair vertices along “imaginary” mesh polylines. I know, I know, but bear with me for a moment. You can select vertices that are not connected by actual mesh edges and the fairing optimization will act as if there is a polyline between them and remove the kinks.

Caution: the picking order of the vertices always matters. When you are selecting vertices imagine that you are drawing the polyline that you want to fair along.

When invoked, the command shows previously defined lists of vertices, if any, or prompts to select at least 3 vertices to create a new list. It has the following options:

Add allows you to add another list of vertices for additional fairness optimization.

Remove purges the currently displayed list.

Previous lets you switch to the previous list of vertices.

Next switches to the next list.

Importance lets you specify the optimization importance value for the current list of vertices. This importance

value is multiplied by the importance value for FairnessCurvature.

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

This is only available in EvoluteTools PRO.

8.5 Optimization Examples

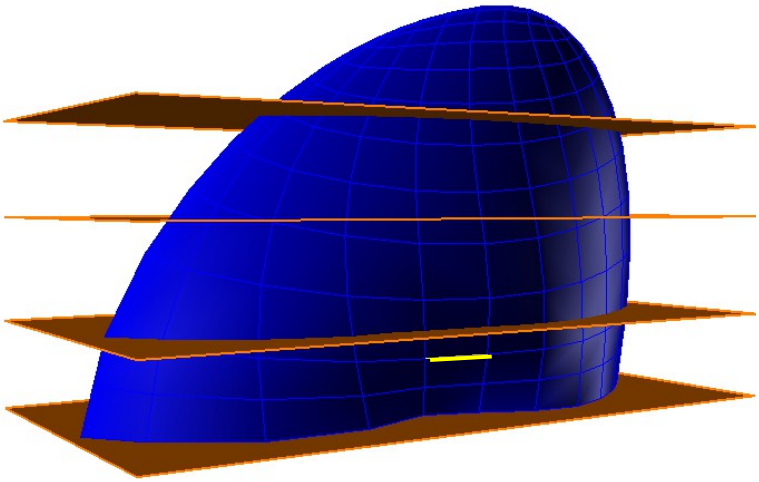
Without further ado, the optimization examples:

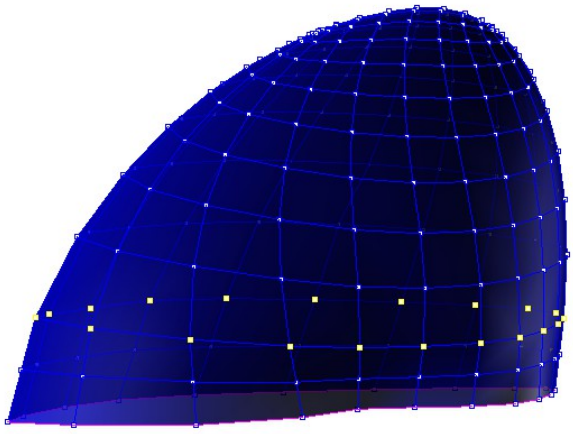
8.5.1 Example 9 - Planarity, Coplanarity, Fairness

In this example we will start with an initial mesh grid for the facade of a building and then optimize it for planarity while at the same time aligning the panel layout with the building's floor slabs.

1. Open the file called Primer_Example9. This file contains a reference surface, the initial mesh layout and a group of planes that you will use to align the panels.

2. The first order of business is to set groups of the vertices to be coplanar using the *'_etSetVerticesCoplanar'* command,  . But rather than select all your vertices by hand once you've run that command, you will preselect each group with the *'_etSelectMeshPolyline'* tool,  . Select that tool and then select a horizontal edge on a mesh polyline close to the 1st floor slab plane. [see image below]





3. This will select all the vertices along that polyline. Pretty convenient! Now all you need to do is run the coplanar tool,  . The dialogue in the command line will prompt you for “Planetype” and “Importance”. You should leave the importance set to 1, but you will need to change the plane type. Select the “Planetype” option.

4. Now the dialogue will give you a long list of plane options. Section 8.4.7 has more details. For now, select “Fixedplane” and press [Enter].

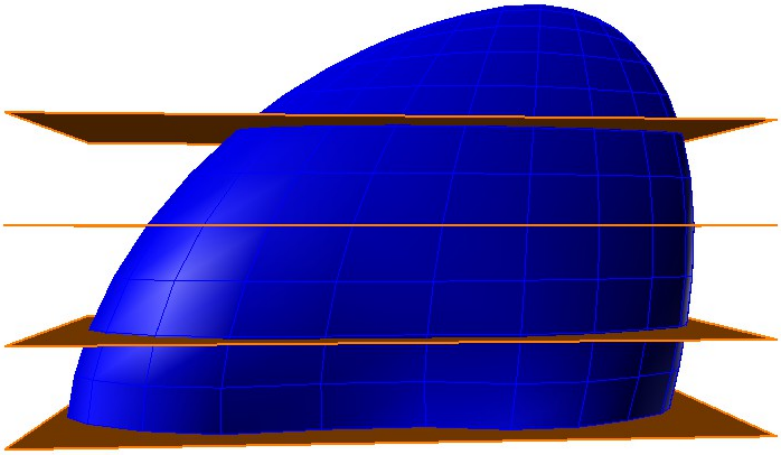
5. Next you are given 3 options for defining the fixed plane. Since you already have the planes in the drawing, you can simply choose “SelectPlane”.

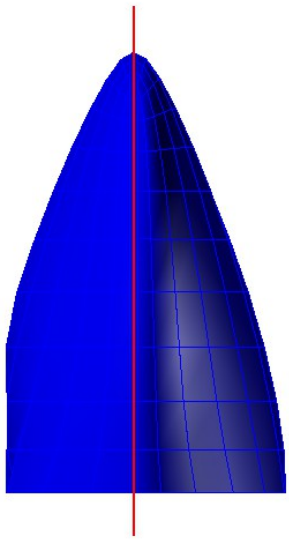
6. Select the first floor plane and look for the confirmation in the command line: “Preselected grips set to become coplanar.”

7. Now repeat this process for the remaining floor planes and the central symmetry plane.

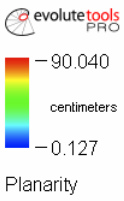
8. Make sure your optimization settings are reset,  . Then set *FairnessCurvature* to '0.2', *FairnessCurvatureVariation* to '0.4' and *Coplanarity* to '0.6'. **Note:** Either fairness term alone would cause the mesh polylines to become either too straight and twisted or too curved. This combination attempts to balance these effects.

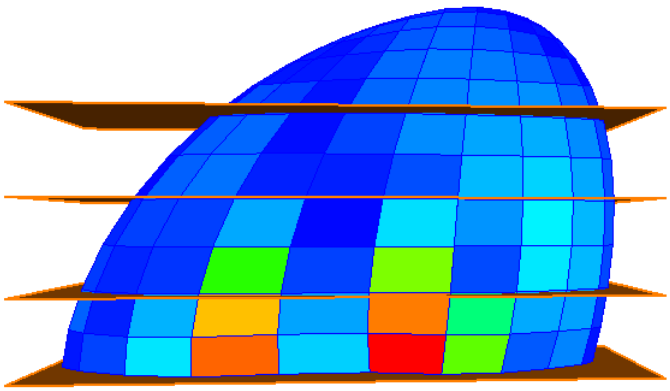
9. Run the optimizer several times. Your mesh should now be nicely aligned with the central and floor planes.



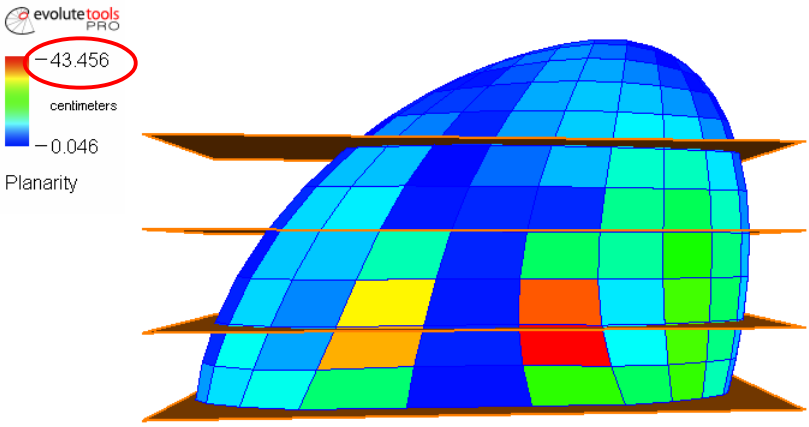


10. The grid layout is looking much better, but remember that we wanted the panels to be more planar as well. Let's take a look at the planarity by using the *'_etAnalyzePlanarity'* command,  .



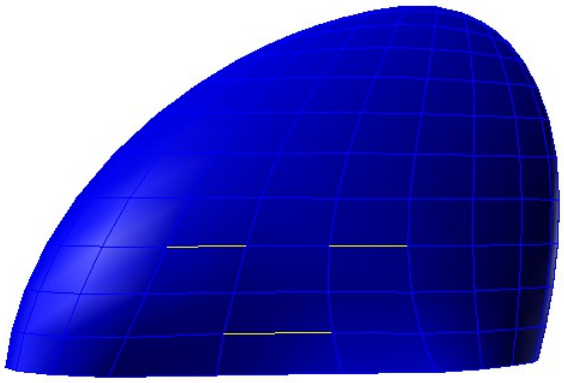


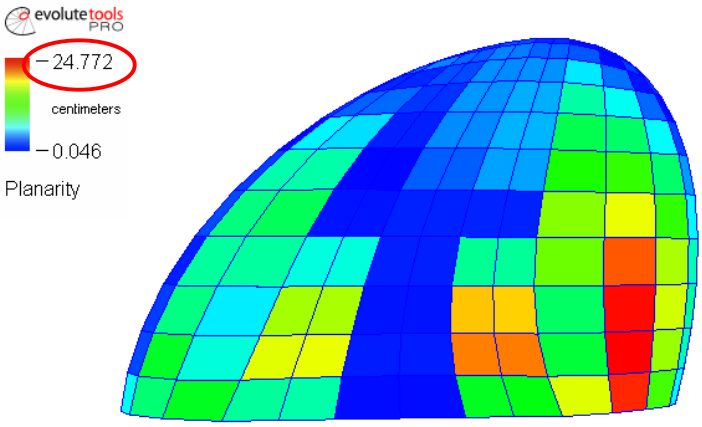
11. There are some definite problem areas. Let's bump the *Planarity* optimization importance up to '1'. Now optimize. (make sure to reset the scale by clicking the “Min-Max Range” button in the Mesh Analysis window)



12. You should see a marked improvement. The maximum deviation from planarity is still too high, but it has been cut in half. In reality, the panels are too large for approximating the target geometry.

13. To proceed, use the Mesh Loop Cuts [section 7.3] to divide the 3 center sections by selecting the 3 edges shown to the right. This step alone, without any additional optimization will improve the planarity.



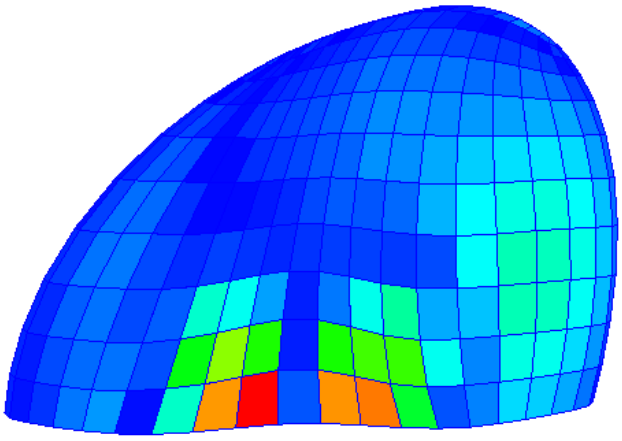


14. Before optimizing again, you should switch the FairingMeasureScaleInvariant «» toggle to **'on'**. This shape has a high degree of curvature along the central spine compared to the relatively flat sides. Without this toggle, the optimizer will concentrate the smaller edges there. This would move the detail we just added too far away from the problem area.

15. Optimize some more.

16. Repeat steps 13 & 15 several more times. You should notice a couple things. First of all, there are still a few problem panels, but the vast majority of your mesh is becoming increasingly planar. Additionally, there is a bulge forming along the polyline you previously went to all the trouble to set as coplanar!

    This happens because the new vertices that you added with the loop cuts are not included in the coplanarity groups. You'll have to do this manually.



17. Select your mesh and then activate the coplanarity tool,  . This will display the first grouping of vertices and you should see the new vertices in the bulge are not selected. In the command line dialogue, click *'Add'* and then select the new vertices. Hit [enter] and then select the appropriate plane.
Note: this will not add them to the other group, but you will set them to be coplanar to the same plane and this will have the same effect.

18. Do this for the next group in the bulge as well. Then optimize. This should correct the bulge.

This mesh is clearly not ready to be turned into a finished product yet, but I hope this gives you an idea of how the optimization tools can be used in concert. If you want to continue on your own, try some horizontal loop cuts as well as vertical and perhaps increasing the planarity importance.

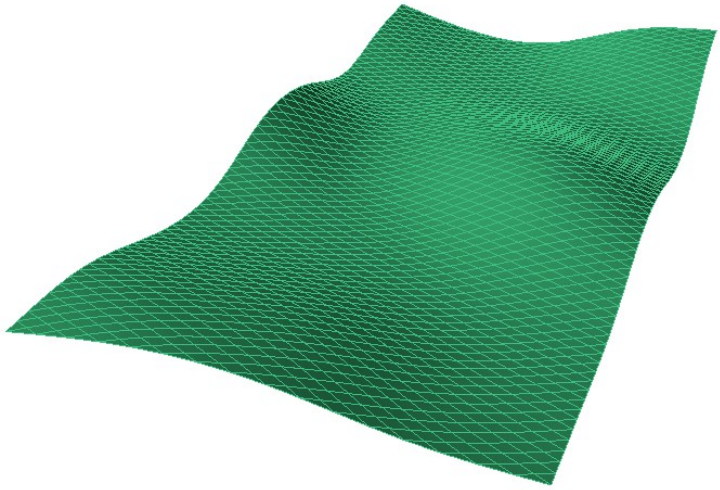
8.5.2 Hexploration – Ballpacking & Ideal Edge Length

If you are an audio-visual learner, I can't recommend too highly the video tutorial on Evolute's YouTube channel. This example will differ in a key aspect: here we will be more interested in getting an excellent ballpacking with very similar edge lengths without staying strictly within the reference boundary. However, it follows the same process more or less.

1. Open the file PrimerExample_10. This file contains the standard “Flying Carpet” reference surface and a quad mesh which is waiting to be optimized. It has its corner vertices flagged and even the optimization importance settings have been primed for you.

2. Open the importance settings and take a look. Just a dash of FairnessSprings is all you need. In this case, either of the other fairness terms causes the mesh polylines to move around a little bit too much. Don't believe me? Try them out for yourself.

3. Optimize a few times until convergence and then subdivide with the *'Diagonalize'* subdivision tool. Your mesh will look like this:



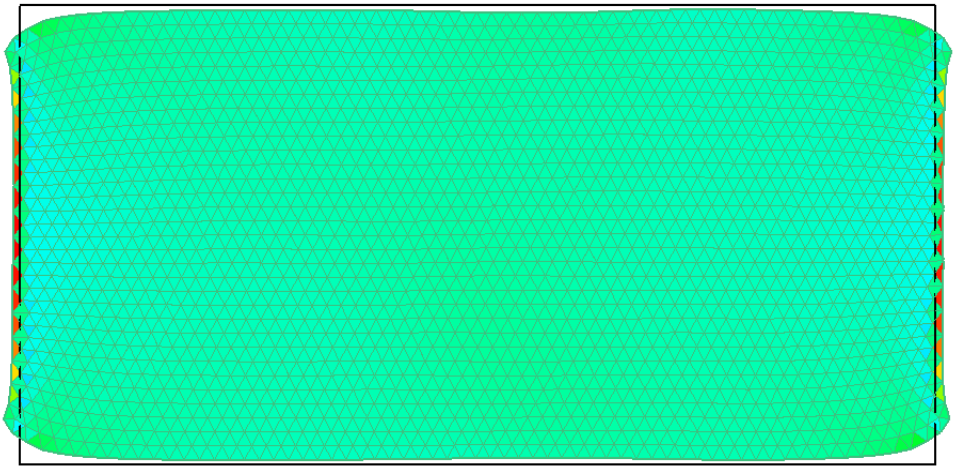
4. To get a hexagonal mesh you will first need a triangular mesh. To do that you will use this tool:  *'_etMeshAddDiagonalLinesParallel'* with an interval of '0'. Select two vertices in the long direction of the surface. Your mesh should now be a triangular mesh.

5. Now things are about to get a bit out of control, but in a good way. Set *CurveCloseness* to '0.5'. Set the *FairnessSprings* to '0', *Ballpacking* to '1' and *IdealEdgeLength* to '1'. Also, unflag the corner vertices as corners and fixed.

6. In order for the *IdealEdgeLength* term to have any effect, we must flag the edges you want to optimize. In this case, all of them! To do that, click on 

or run '_etSetEdgeLengthOptimization'. In the command line dialogue click on: 'SelectionMethod = SingleEdge' to change how you will select the desired edges. Next choose 'AddAll' since you want to optimize all the edges. Then hit [enter]

7. To get an idea if the IdealEdgeLength term worked, open the Mesh Analysis window (by selecting ) and then use the “Mode” drop down menu to select “Edgelengths”. This analysis view colors each face according to its longest edge. Your mesh should now look like the following image. The boundary of the target surface is shown for reference.

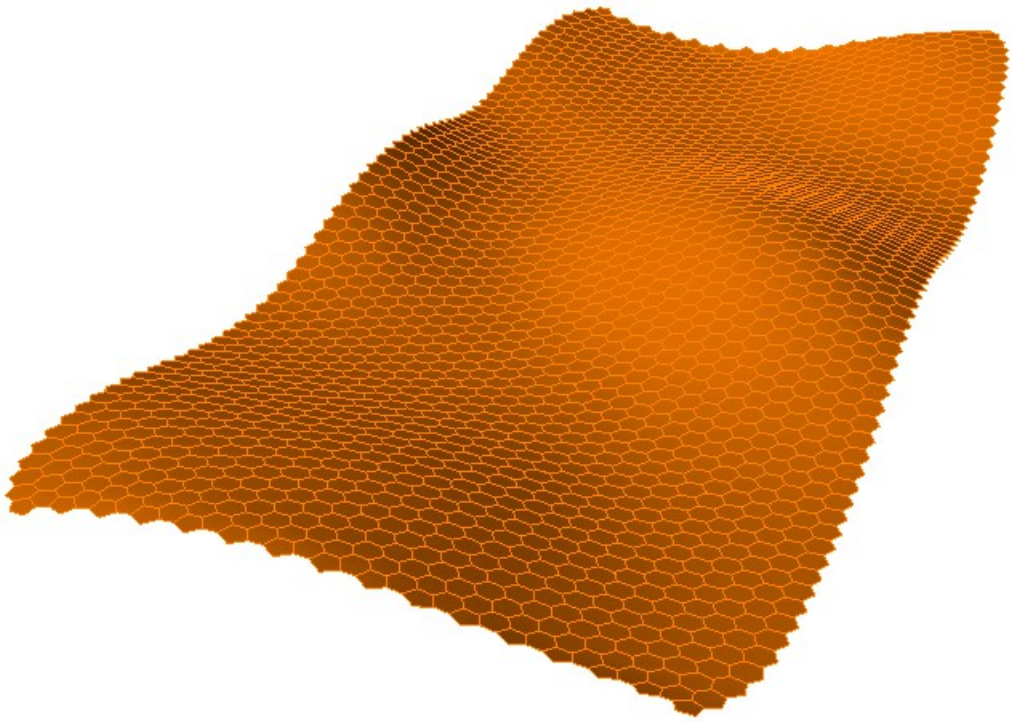


8. You can see from the coloring that aside from the faces at the narrow ends, the edges are very uniform. The mesh has also pulled in from the boundary along the long edges. This has to do with the aspect ratio of the rectangles we start with and how close the resulting triangles are to equilateral.

   Hint: For the best result start, with quad faces that have roughly a 1:1.73 ratio. Try going back to the beginning and using a different coarse mesh with these proportions!

   The triangles on the end are a problem, because they were not divided by the Add Parallel Diagonals tool. This can be solved in a number of ways. You can delete them and replace them with two triangular faces. This is a bit tedious to by hand and offers an excellent opportunity to write some RhinoScript code. If you are up for it, check out Marko's EvoluteTools Scripting Primer! For now we will just ignore them.

9. To convert this triangular mesh into a hexagonal mesh you only need to run a single Dual subdivision. [or if you're feeling a bit adventurous try a TriHex subdivison]



9. Getting the Most Out of EvoluteTools

The goal of this chapter is to provide you with additional Tips & Tricks for using EvoluteTools. However, rather than a list of case-specific instructions, these will be more general methods or best-practices.

9.1 An Iterative Process

The examples in this primer follow a mostly linear process and this is a bit misleading. It misses a crucial part of using EvoluteTools in the most effective way possible. That is, using it in iterative or cyclical process.

Discretizing a complex free-form surface is often not a straightforward task. Furthermore, it can be difficult to predict the outcome of a given optimization (and if it were easy, why bother with the optimization step?). The beauty of EvoluteTools is that it is flexible and fast. The flexibility allows you to tweak optimization settings or add constraints on the fly and observe the results. The speed with which you can generate outcomes allows for experimentation through trial and error. This is particularly true when you are first learning to use the software. Don't be afraid to go back to the beginning and try something new. That first coarse mesh didn't work? Go back and edit it.

So where did I lead you astray? Pretty much right from the beginning. Now don't get too angry. Remember that image of the typical workflow? Well, in practice you won't follow it in such a linear way. You'll often want to iterate between the subdivision and optimization steps. While I'm making confessions: You also shouldn't always delete the coarse mesh after you subdivide. The connection between a subdivided mesh and its parent can be very useful.

9.2 Logical Connections

In the following section, the term coarse mesh or parent is used for the original mesh selected for subdivision. The newly created mesh will be called the subdivided mesh or child.

After subdivision, these two meshes are *logically connected*.

What does that mean? Well, EvoluteTools maintains a link between the two meshes. The vertices of the subdivided mesh depend linearly on one or more vertices of the coarse mesh. If the parent mesh is deformed or otherwise edited, the child mesh will reproduce these alterations (according to the rules of the subdivision algorithm that was originally used to create it). This is very similar to the connection between a curve or surface and its control points.

By subdividing the subdivided mesh once again (using the same or any other algorithm), a subdivision chain can be created, enabling multi-scale subdivision modeling. Editing the connectivity of the subdivided mesh is only supported using the mesh editing tools provided in EvoluteTools. Deleting vertices or faces from a subdivided mesh using Rhino's standard commands will automatically decouple the mesh from the coarse mesh. Subdivision dependencies created by this command can be visualized using *'_etShowDependencies'*,  .

Important: Optimization only affects the vertices of the parent mesh, the others are created and positioned by the subdivision algorithm. This cuts both ways.

First, **the positive:** when you have a very large mesh with many small faces, the optimizer can begin to run very slowly as it must calculate for each vertex. If you maintain the connection between parent and child, the optimizer will run much more quickly as it has much fewer vertices to worry about.

It is also useful to keep the connection if you are using a subdivision that results in an “unstable” geometry. For instance, the planarity resulting from the Dual Edge subdivision. If you were to remove the connection and then optimize, this planarity would be lost as the optimizer began to shift the vertices. On the other hand, if this connection is preserved, the optimizer will only move the vertices of the parent mesh. Then the child mesh will still be a perfect Dual Edge subdivision of the altered parent mesh and will retain the resulting planarity.

There can be **negative** consequences to keeping the connection. If the parent mesh has too few vertices, the optimizer won’t have enough “material” to work with and won't be able to effectively approximate the target geometry. This is the reason that the examples have all called for deleting the original coarse mesh. Alternatively, the parent mesh can be kept and the connection can be severed with the *'_etDecoupleSubdivision'* tool,  .

Typically, you will want to dispose of the very first coarse mesh and use this connection between meshes later as your meshes become more complex.

Well faithful reader, that brings us to the end (for now). I think by now you've seen that EvoluteTools is an incredibly versatile piece of software that is loaded with functionality. Even at 60 pages, I've only just scratched the surface of what it can do and how its features can be used in concert to create and rationalize complex geometry. In the end, this is just a primer and by now you should consider yourself well and fully primed. In parting I will remind you of my 3 rules for success with EvoluteTools, specifically, and life, in general :

1. Experiment.
2. Experiment.
3. Experiment.

You'll be amazed at what you can accomplish with some dedicated messing around.

If you have additional questions or are looking for more EvoluteTools tutorials, make sure you check the Evolute website: http://www.evolute.at/ and YouTube channel : http://www.youtube.com/user/evolutegeometry

Or contact us via email at: support@evolute.at

Thanks for reading and enjoy using EvoluteTools !

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).

Revision 1.0

15th of November 2012