SimplyRhino, London
February 12-14, 2020

Python Scripting for Rhino/Grasshopper

Long Nguyen

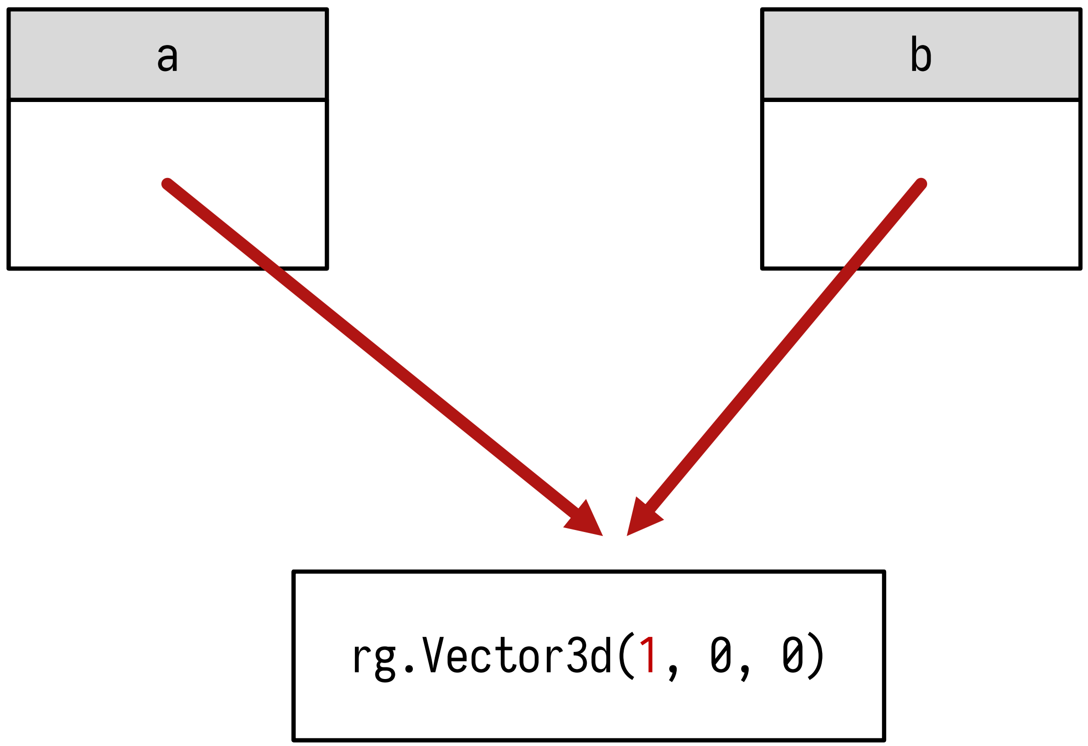
What is actually “stored” inside a variable

```
a = 2
b = a
b = 3
print a
```

```
a = rg.Vector3d(2, 0, 0)
b = a
b.Unitize()
print b
print a
```

What is actually “stored” inside a variable

```
a = rg.Vector3d(2, 0, 0)
b = a
b.Unitize()
print b
print a
```



What is actually "stored" inside a variable

```
a = ["Batman"]
b = a
b.append("Spiderman")
print a
```

How arguments are passed into a function:

```
import Rhino.Geometry as rg

def DoSomething(someVector):
    someVector.Unitize()

myVector = rg.Vector3d(2, 0, 0)
DoSomething(myVector)
print myVector
```

Tuples

What is a tuple?

- A tuple let us pack related data into a single item in a structurally fixed way

Without using tuple

```
firstName = "John"
lastName = "Clarks"
birthYear = 1988
profession = "Architecture"
```

Using tuple

```
john = ("John", "Clarks", 1988, "Architecture")
```

- Here we have a quadruple (a tuple of 4 elements)
- Obviously we can also have pairs, triples, quintuple, sextuple, etc…

Retrieve elements from a tuple

```
john = (“John”, “Clarks”, 1988, “Architecture”)

# Retrive a single element, using the index operator
birthYear = john[2]
print birthYear
print john[-1]

# Retrive a range of elements as a (smaller) tuple, using the slice operator
name = john[0:2]
print name
```

```
1988
“Architecture”
(“John”, “Clarks”)
```

Nested Tuples

Defining tuple-within-tuple

- A pair of quadruples

```
hadid = ("Zaha", "Hadid", 1950, "Architecture")
sanders = ("Bernie", "Sanders", 1941, "Politics")

couple = (sanders, hadid)
```

```
print couple[0][1] # This will print "Sanders"
```

Using tuples as function inputs

```
hadid = ("Zaha", "Hadid", 1950, "Architecture")
sanders = ("Bernie", "Sanders", 1941, "Politics")
swift = ("Taylor", "Swift", 1989, "Music")

def GetOlderPerson(personA, personB):
    if (personA[2] < personB[2]):
        return personA
    else:
        return personB

result1 = GetOlderPerson(hadid, sanders)
result2 = GetOlderPerson(sanders, swift)
```

Lists vs. Tuples

- A list usually stores items of the same type (e.g. strings)
- And the items have equivalent “meaning” (e.g. all are names of superheroes)

```
heroes = ["Batman", "Wolverine", "Superman"]
```

- A tuple can store multiple items of the same or different types
- These items “meaningfully” belong to the same entity (e.g. information related to John)
- but they often don’t have equivalent meaning (e.g. name has different “meaning” from birthyear)

```
john = (“John”, “Clarks”, 1988, “Architecture”)
```

Python in Rhino

Creating Rhino objects

Creating a point object

```
import Rhino.Geometry as rg
import Rhino

myPoint = rg.Point3d(10, 10, 2)

document = Rhino.RhinoDoc.ActiveDoc
document.Objects.AddPoint(myPoint)


document.Views.Redraw() # Force the viewport to redraw immediately
```

Saving the GUID of the Rhino object for later use

```
import Rhino.Geometry as rg
import Rhino

myPoint = rg.Point3d(10, 10, 2)

document = Rhino.RhinoDoc.ActiveDoc
pointObjectGuid = document.Objects.AddPoint(myPoint)


document.Views.Redraw() # Force the viewport to redraw immediately
```

Creating a line object

```python
import Rhino.Geometry as rg
import Rhino

myPoint = rg.Point3d(10, 10, 2)

document = Rhino.RhinoDoc.ActiveDoc
pointObjectGuid = document.Objects.AddPoint(myPoint)
lineObjectGuid = document.Objects.AddLine(myPoint, rg.Point3d(20, 20, 0))

document.Views.Redraw() # Force the viewport to redraw immediately
```

RhinoScriptSyntax

Creating a point object, using RhinoScriptSyntax library

Previously, …

```
import Rhino.Geometry as rg
import Rhino

myPoint = rg.Point3d(10, 10, 2)
document = Rhino.RhinoDoc.ActiveDoc
pointObjectGuid = document.Objects.AddPoint(myPoint)
document.Views.Redraw()
```

Now, using the RhinoScriptSyntax library instead

```
import rhinoscriptsyntax as rs

pointObjectGuid = rs.AddPoint(10, 10, 2)
```

Asking for user’s input

Randomness

Generate random integers

```
import random

myRandomNumber = random.randint(0, 5)
print myRandomNumber
```

Generate a random integers

```
import random

for i in range(0, 10):
    myRandomNumber = random.randint(0, 5)
    print myRandomNumber
```

Generate 10 random integers

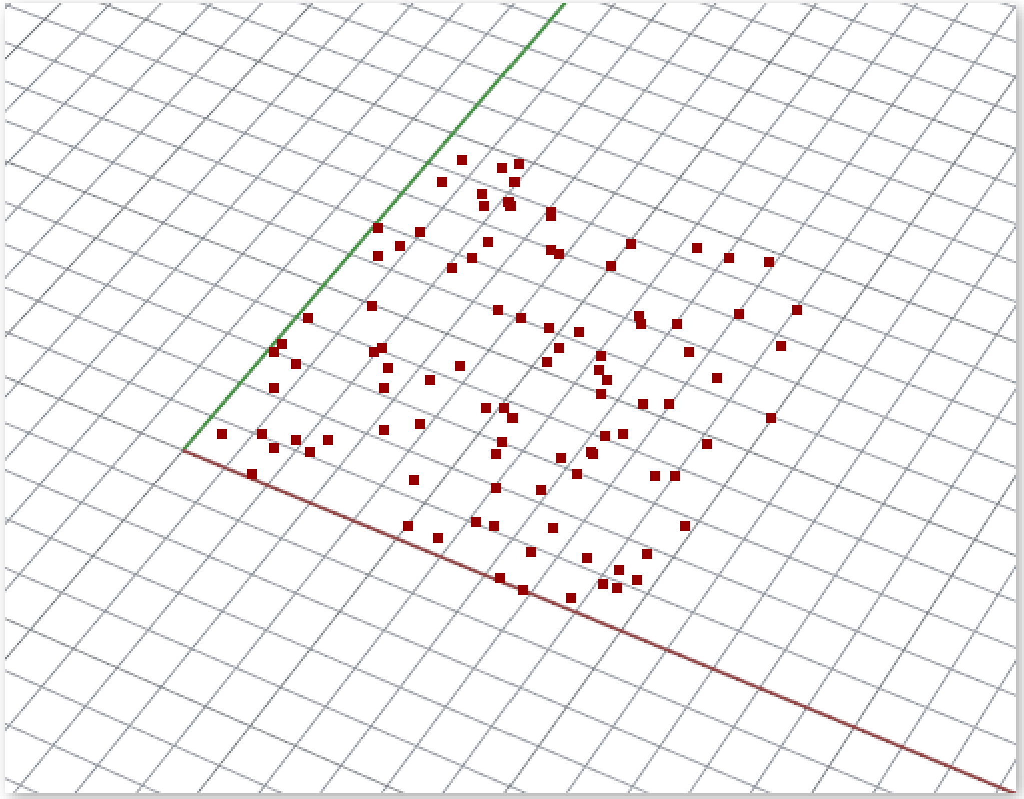
```
import random

for i in range(0, 10):
    myRandomNumber = random.uniform(1.2, 5.0)
    print myRandomNumber
```

Generate 10 random floats between 1.2 and 5.0

Generate points with random x, y coordinates

```
INPUTS
iCount: int

OUTPUTS:
oGeometry
```

```
import Rhino.Geometry as rg
import random

points = []

for i in range(0, iCount):
    x = random.uniform(0.0, 10.0)
    y = random.uniform(0.0, 10.0)
    point = rg.Point3d(x, y, 0.0)
    points.append(point)

oGeometry = points
```



Randomness does not have to be completely chaotic.

We can control randomness in many ways

Randomly offsetting 2D gridpoints

This is the usual 2D grid of points

```
import Rhino.Geometry as rg
import math
import random

points = []

for i in range(0, 60):
    for j in range(0, 30):
        x = i
        y = j
        points.append(rg.Point3d(x , y, 0))

oGeometry = points
```

```
INPUTS
iVariation: float

OUTPUTS:
oGeometry
```

Randomly offsetting grid points

… now let's add some small random offset from their regular positions

```
import Rhino.Geometry as rg
import math
import random

points = []

for i in range(0, 60):
    for j in range(0, 30):
        x = i + random.uniform(-0.1, 0.1)
        y = j + random.uniform(-0.1, 0.1)
        points.append(rg.Point3d(x , y, 0))

oGeometry = points
```

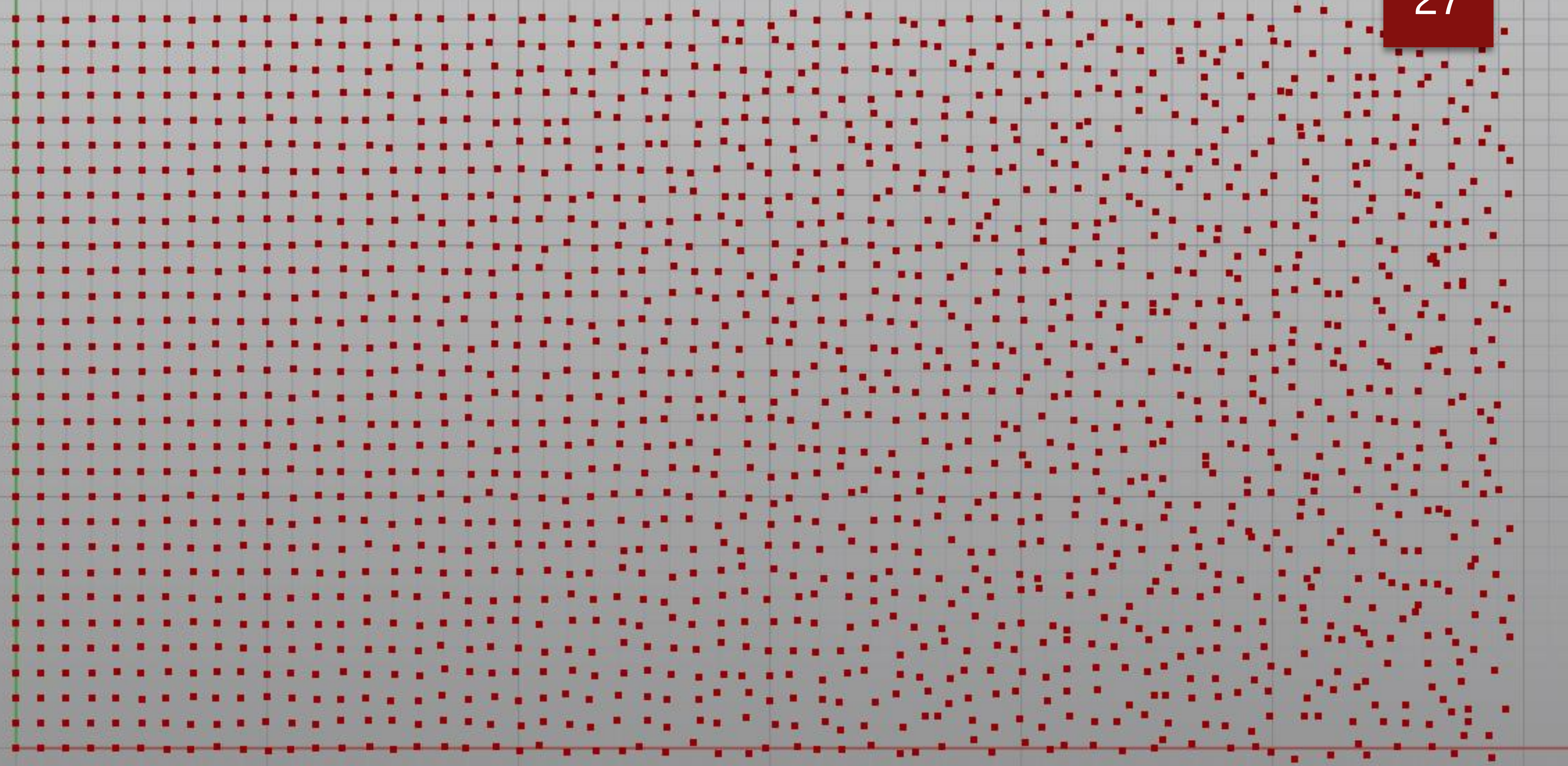
```
INPUTS
iVariation: float

OUTPUTS:
oGeometry
```

Randomly offsetting grid points

… next, let's change the amount of offsetting by in external input parameter

```
import Rhino.Geometry as rg
import math
import random

points = []

for i in range(0, 60):
    for j in range(0, 30):
        x = i + iVariation * random.uniform(-1.0, 1.0)
        y = j + iVariation * random.uniform(-1.0, 1.0)
        points.append(rg.Point3d(x , y, 0))

oGeometry = points
```

```
INPUTS
iVariation: float

OUTPUTS:
oGeometry
```



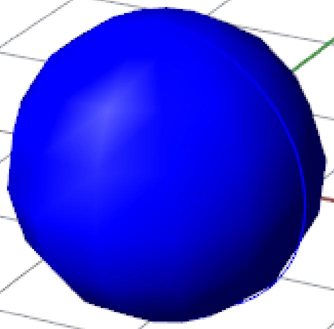
How about this: Transition from no variation to maximum variation

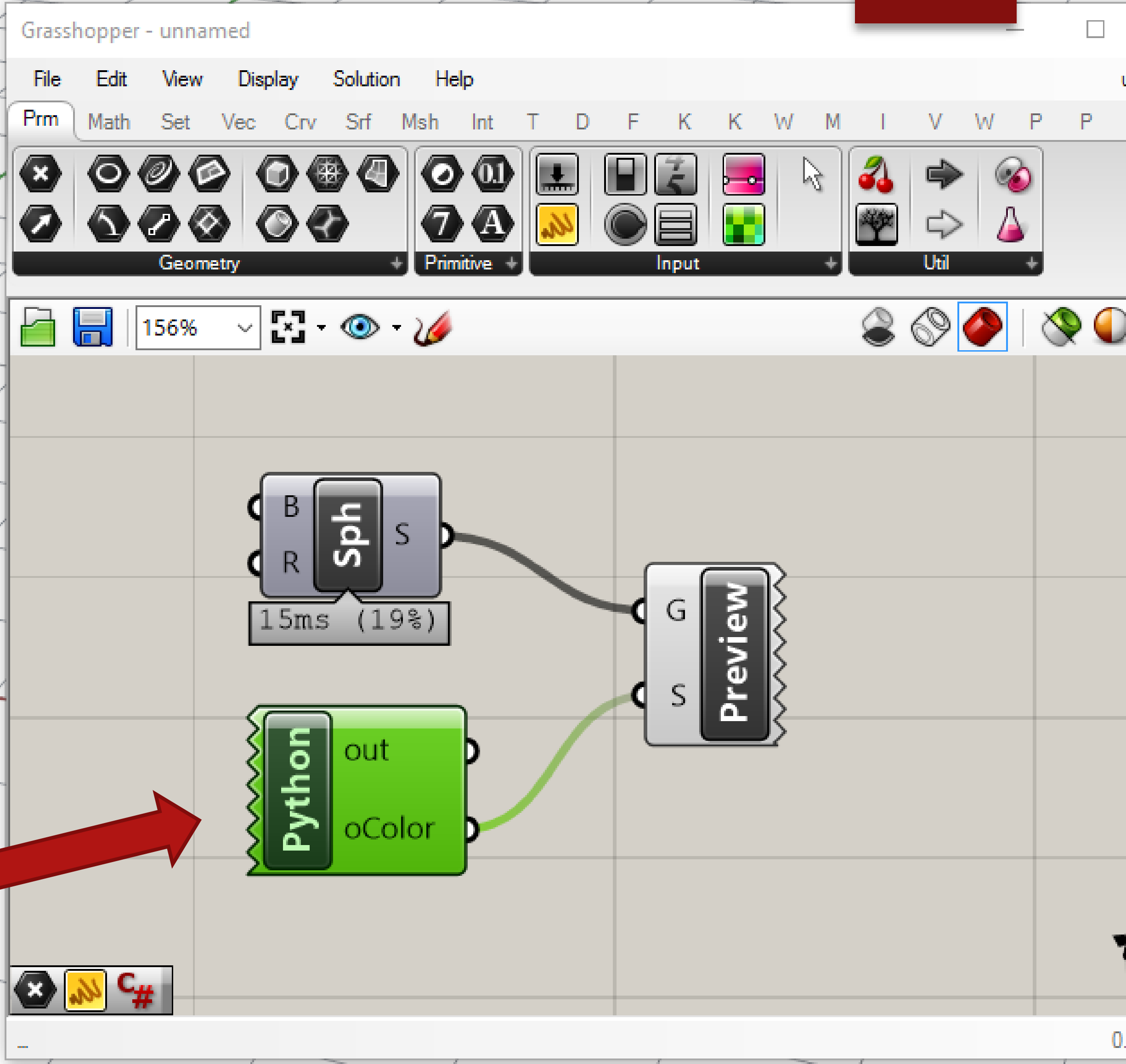
Transition from “regular” to “random”

```
import Rhino.Geometry as rg
import math
import random

points = []

for i in range(0, 60):
    for j in range(0, 30):
        variation = 0.5 * (i / 59)
        x = i + variation * random.uniform(-1.0, 1.0)
        y = j + variation * random.uniform(-1.0, 1.0)
        points.append(rg.Point3d(x , y, 0))

oGeometry = points
```

Colors

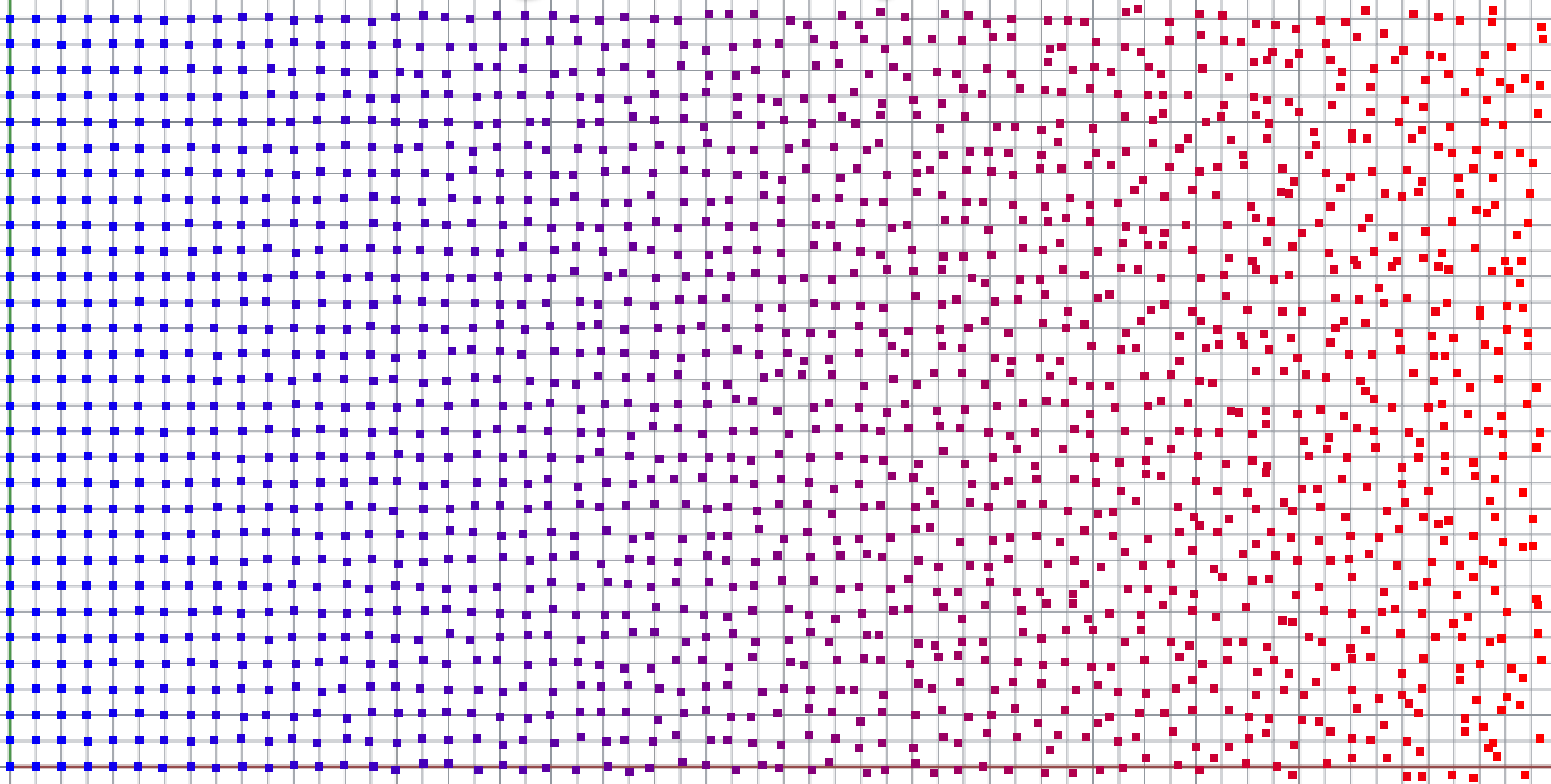
Creating Color using Python






```python
import System.Drawing as drawing

oColor = drawing.Color.FromArgb(0, 0, 255)
```

Control colors algorithmically



Control colors algorithmically

```
import Rhino.Geometry as rg
import math
import random
import System.Drawing as drawing

points = []
colors = []

for i in range(0, 60):
    for j in range(0, 30):
        variation = 0.5 * (i / 59)
        x = i + variation * random.uniform(-1.0, 1.0)
        y = j + variation * random.uniform(-1.0, 1.0)
        points.append(rg.Point3d(x , y, 0))

        red = 255 * i / 59
        blue = 255 - red
        color = drawing.Color.FromArgb(red, 0, blue)
        colors.append(color)

oGeometry = points
oColor = colors
```

Parametric Curves

Creating curves

Creating NurbsCurve: Interpolated Curve

```
INPUTS
iPoints: List of Point3d

OUTPUTS:
oCurve
```

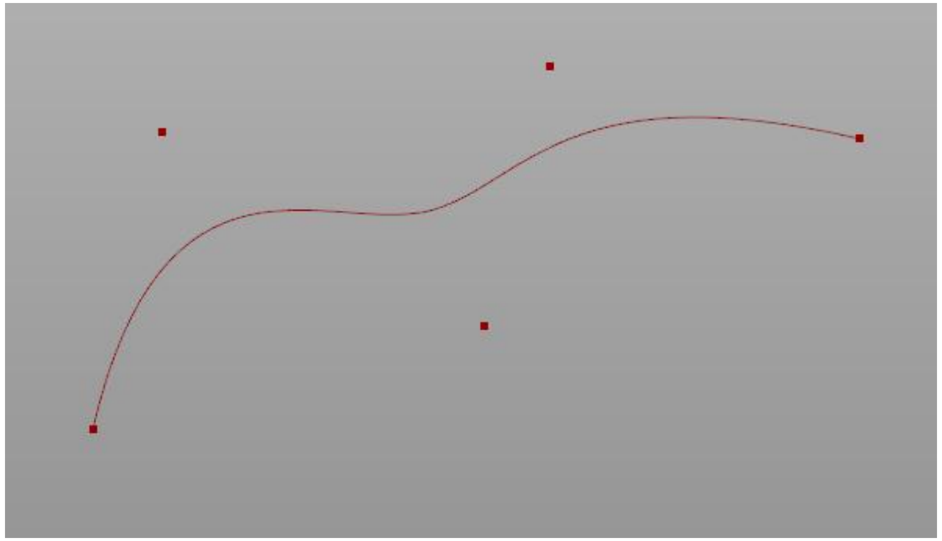
```python
import Rhino.Geometry as rg

oCurve = rg.Curve.CreateInterpolatedCurve(iPoints, 3)
```

Creating NurbsCurve: Control Point Curve

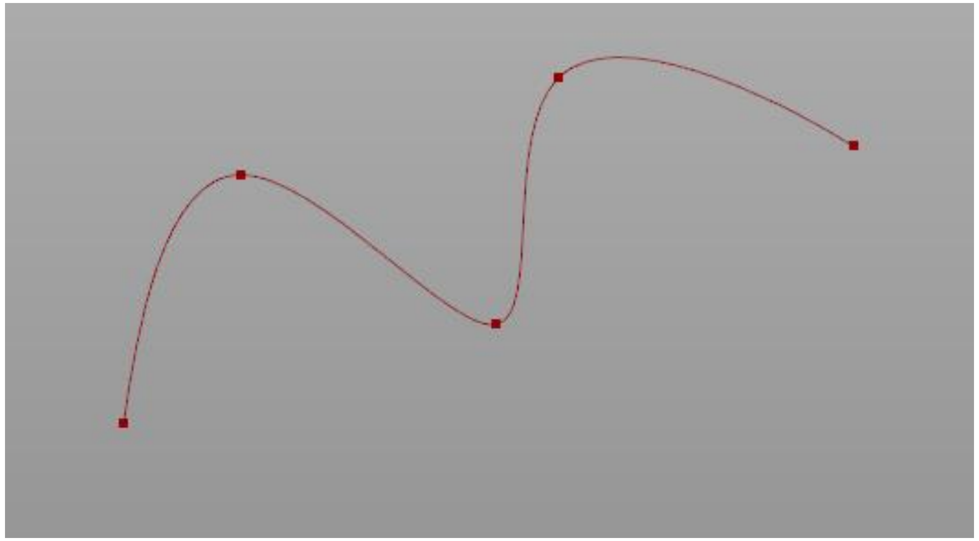
```
INPUTS
iPoints: List of Point3d

OUTPUTS:
oCurve
```

```python
import Rhino.Geometry as rg

oCurve = rg.Curve.CreateControlPointCurve(iPoints, 3)
```

Control Point Curve vs. Interpolated Curve



Control Point Curve
(Approximated Curve)



Interpolated Curve

(Live demonstration of twisting issues and tangent line up)

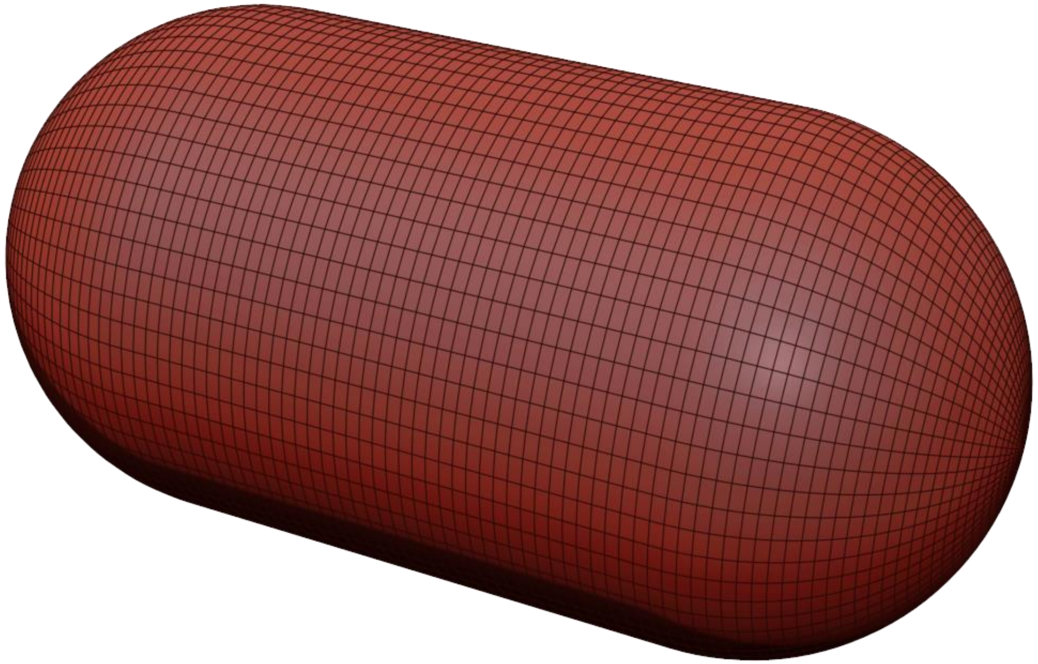
Curve Degree

- Higher curve degree gives smoother curve
- Higher curve degree means each point on the curve is influenced by more control points
- Why degree 3 is usually the default value?

(Live demonstration)

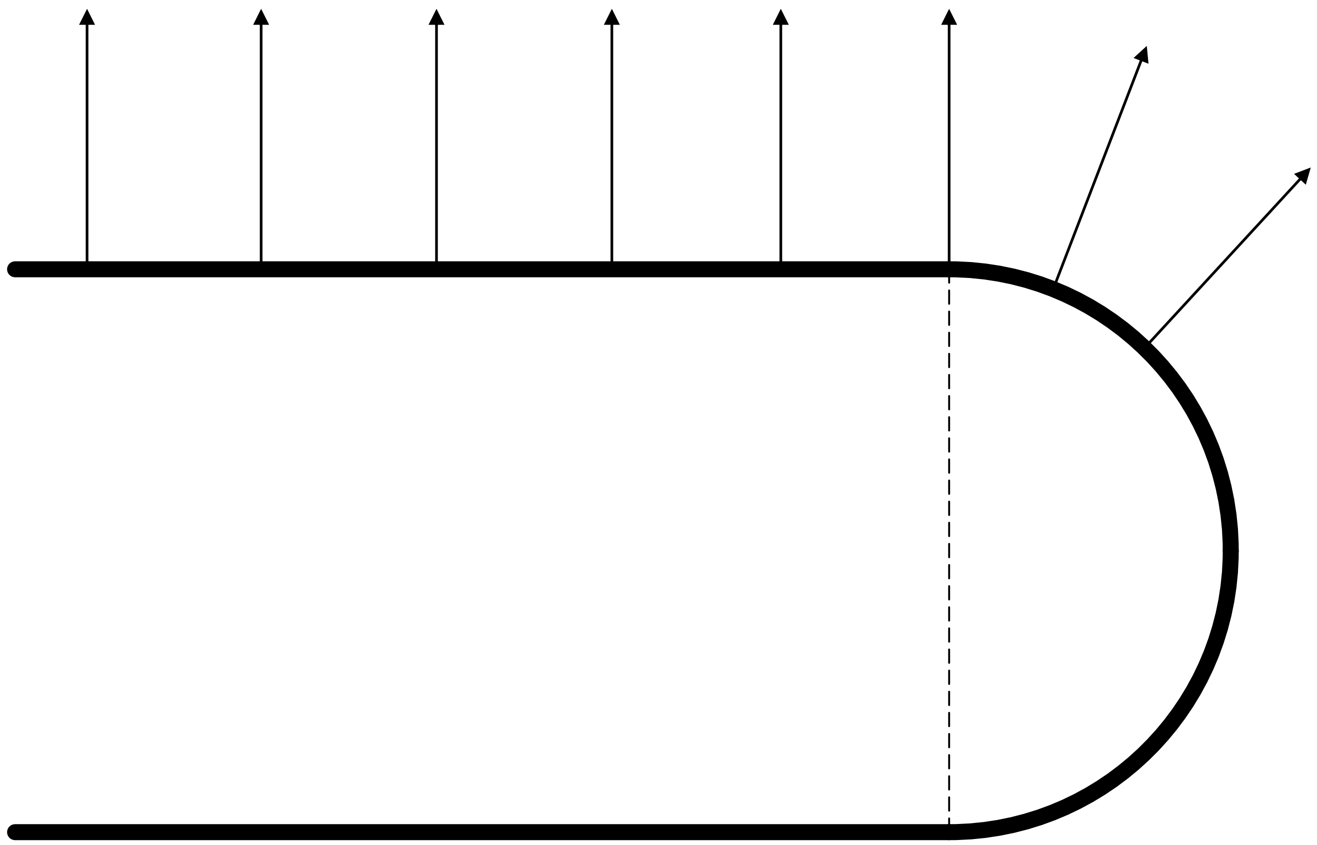
Why degree 3 is usually the default?

Degree 2 is smooth, but not smooth enough



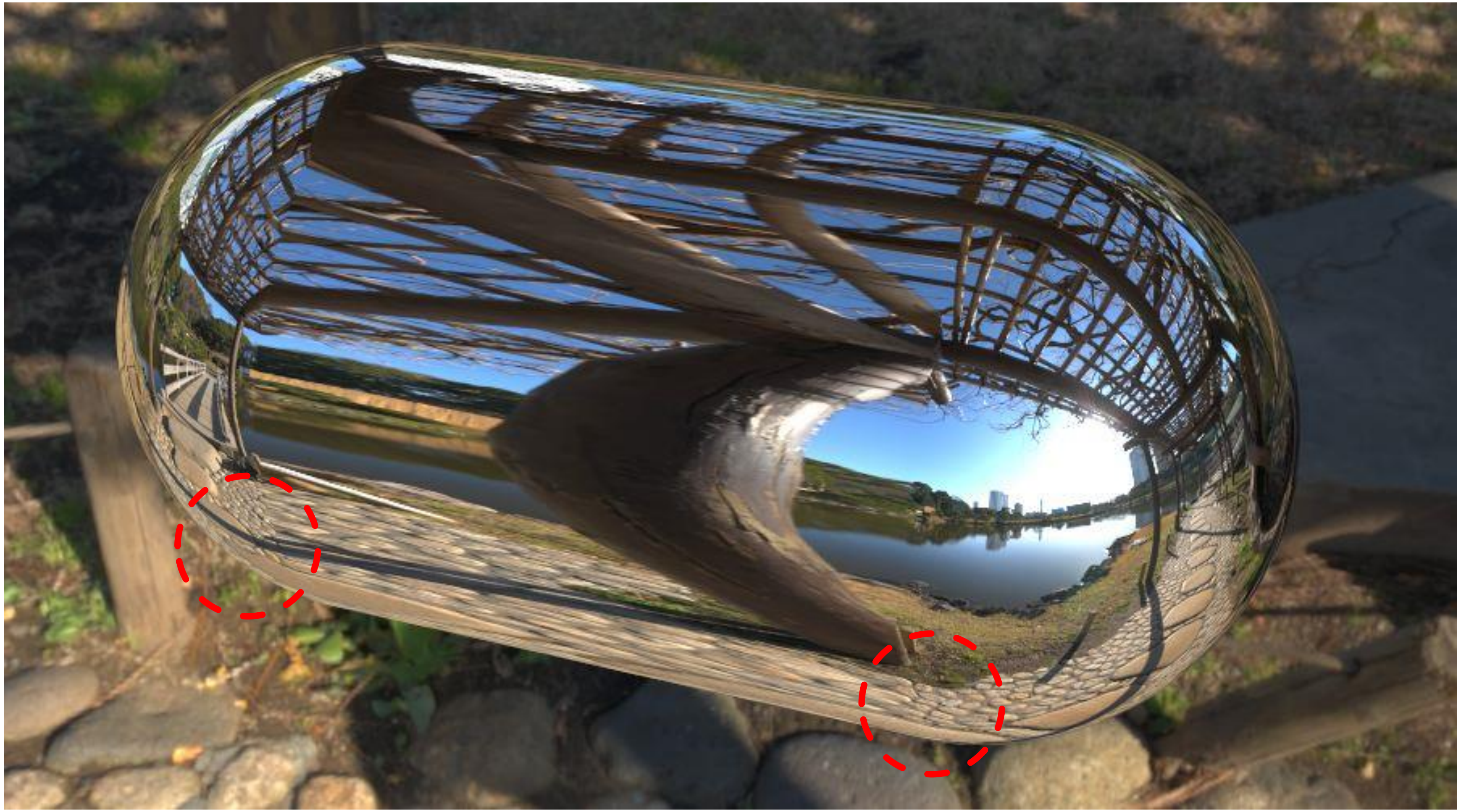
Why degree 3 is usually the default?

Degree 2 is smooth, but not smooth enough



Why degree 3 is usually the default?

Degree 2 is smooth, but not smooth enough



Creating Curves: LineCurve

```
import Rhino.Geometry as rg

myLineCurve = rg.LineCurve(rg.Point3d(0,0,0), rg.Point3d(3, 4, 5))
```

Why Rhino has Line and also LineCurve?
What’s the difference?

Curves

Curve operations

Evaluate points along a curve

Each point on a parametric curve is uniquely correspondent to a number, known as “curve parameter”

```
point = iCurve.PointAt(iCurveParameter)
oGeometry = point
```

```
INPUTS
iCurve: Curve
iCurveParameter: float

OUTPUTS:
oGeometry
```

Other common curve operations

```
# Get the point on the curve at the specified distance from the  starting point of the curve
PointAtLength(float) -> Point3d

# Get the local coordinate frame associated with the point at the specified curve parameter
FrameAt(float) -> (bool, Plane)

# Get the closest point on the curve relative to the input point
ClosestPoint(Point3d) -> (bool, float)
```

C# Functions → Python functions

In a C#, if a function needs to return more than one outputs, it will store the additional outputs in the “out” parameters

C#

```
ClosestPoint(Point3d, out double) → bool
```

In Python, if a function needs to return more than one outputs, it will group all output into a tuple and return that tuple

Python

```
ClosestPoint(Point3d) → (bool, float)
```

Parametric Surfaces

Operations on parametric surfaces

Parametric surface

- Parametric surface is a collection of points, each has its own (u, v) coordinates
- Naturally, an (untrimmed) parametric surface has 4 boundary sides
- Parametric surface has a “positive” side and “negative” side

(Live Demonstration)

Evaluate points on the surface

Each point on a parametric surface is uniquely correspondent to a pair of numbers (u, v)

```
import Rhino.Geometry as rg

iSurface.SetDomain(0, rg.Interval(0.0, 1.0))
iSurface.SetDomain(1, rg.Interval(0.0, 1.0))

point = iSurface.PointAt(iU , iV)
oGeometry = point
```

```
INPUTS
iSurface: Surface
iU: float
iV: float

OUTPUTS:
oGeometry
```

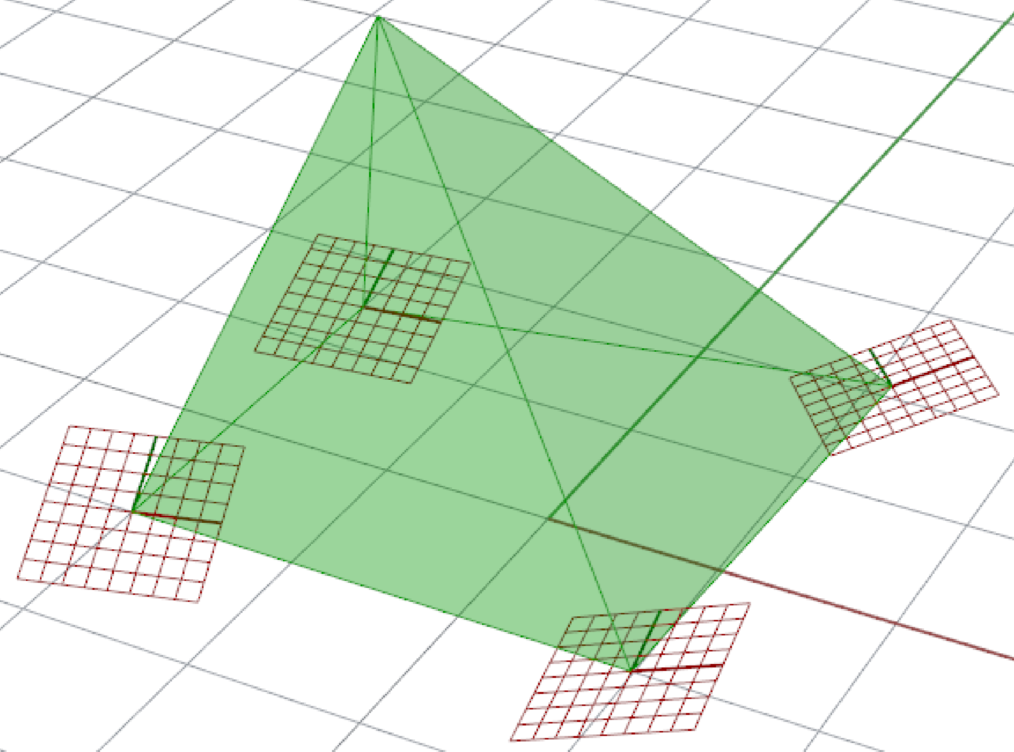
Other common operations

```
# Get the local coordinate frame associated with the point at the specified (u, v) coordinates
FrameAt(float, float) -> Plane

# Get the closest point on the curve relative to the input point
ClosestPoint(Point3d) -> (bool, float, float)

# IMPORTANT: Compute an iso-curve of the surface
IsoCurve(int, float) -> Curve
```

Live Exercise

Parametric pyramid

- Input: 4 planes (coordinate frames)
- Output: a pyramid (as 4 parametric surfaces) whose top is pointing in the direction that is the average of the Z axes of 4 planes



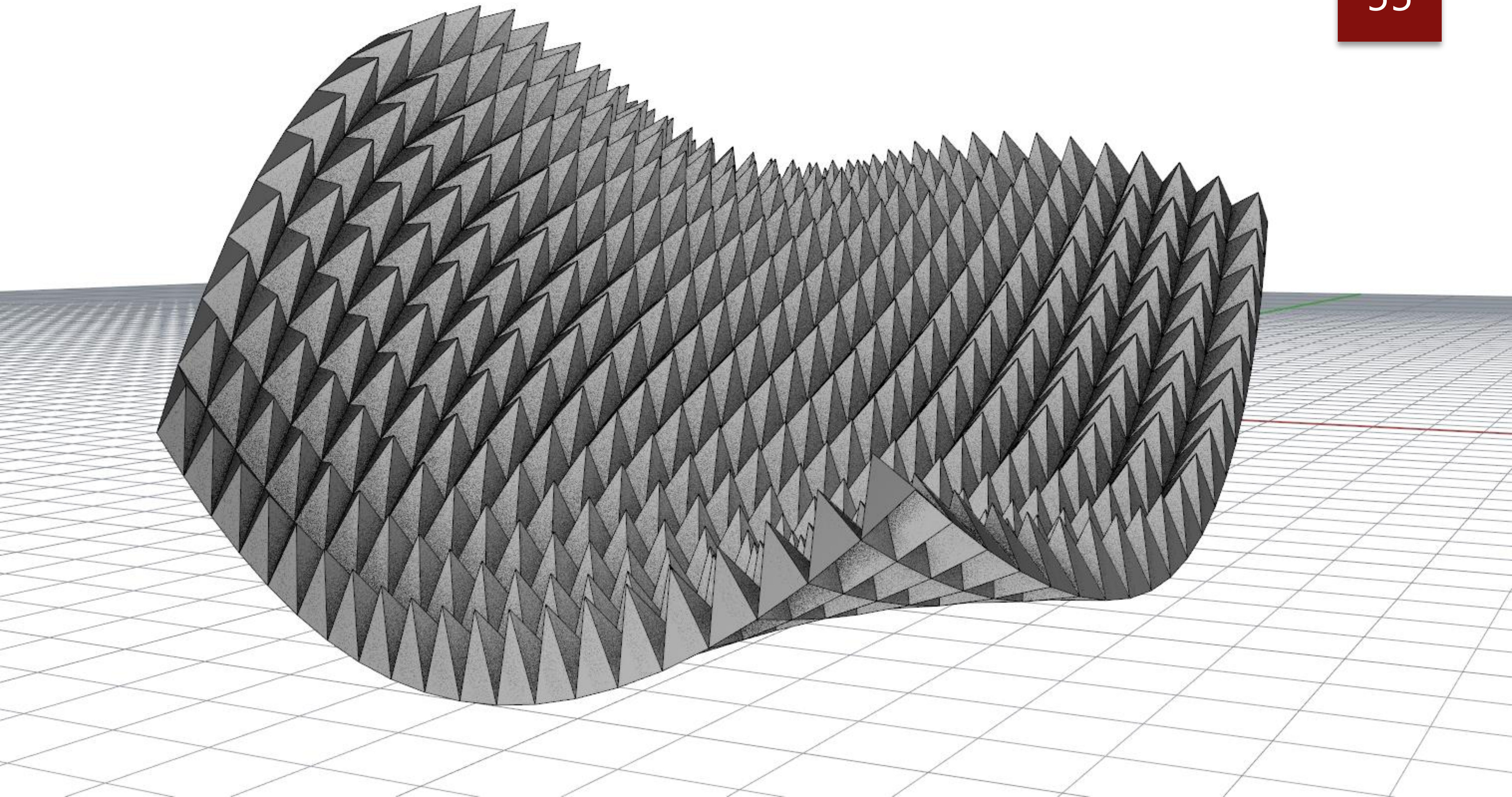
A utitlity function: Create a surface from three points


```
import Rhino.Geometry as rg

def CreateTriangleSurface(A, B, C):
    CB = rg.LineCurve(C, B)
    BA = rg.LineCurve(B, A)
    AC = rg.LineCurve(A, C)
    return rg.Brep.CreateEdgeSurface([CB, BA, AC])

oGeometry = CreateTriangleSurface(iA, iB, iC)
```

```python
import Rhino.Geometry as rg

def CreateTriangleSurface(A, B, C):
    CB = rg.LineCurve(C, B)
    BA = rg.LineCurve(B, A)
    AC = rg.LineCurve(A, C)
    return rg.Brep.CreateEdgeSurface([CB, BA, AC])

def CreateComponentSurfaces(frameA, frameB, frameC, frameD, height):
    z = frameA.ZAxis + frameB.ZAxis + frameC.ZAxis + frameD.ZAxis
    z.Unitize()
    centerBottom = 0.25 * (frameA.Origin + frameB.Origin + frameC.Origin + frameD.Origin)
    top = centerBottom + height * z

    surface1 = CreateTriangleSurface(frameA.Origin, frameB.Origin, top)
    surface2 = CreateTriangleSurface(frameB.Origin, frameC.Origin, top)
    surface3 = CreateTriangleSurface(frameC.Origin, frameD.Origin, top)
    surface4 = CreateTriangleSurface(frameD.Origin, frameA.Origin, top)
    return [surface1, surface2, surface3, surface4]

oGeometry = CreateComponentSurfaces(iFrameA, iFrameB, iFrameC, iFrameD, 3.0)
```

Live Exercise

Adaptive geometry along a uv-surface



```python
import Rhino.Geometry as rg

def CreateTriangleSurface(A, B, C):
    ...

def CreateComponentSurfaces(frameA, frameB, frameC, frameD, height):
    ...

iSurface.SetDomain(0, rg.Interval(0.0, 1.0))
iSurface.SetDomain(1, rg.Interval(0.0, 1.0))

allSurfaces = []

for i in range(0, iUDivision):
    for j in range(0, iVDivision):
        frameA = iSurface.FrameAt(i / iUDivision, j / iVDivision)[1]
        frameB = iSurface.FrameAt((i + 1) / iUDivision, j / iVDivision)[1]
        frameC = iSurface.FrameAt((i + 1) / iUDivision, (j + 1) / iVDivision)[1]
        frameD = iSurface.FrameAt(i / iUDivision, (j + 1) / iVDivision)[1]
        componentSurfaces = CreateComponentSurfaces(frameA, frameB, frameC, frameD, 1.5)
        allSurfaces.extend(componentSurfaces)

oGeometry = allSurfaces
```

Breps

Brep

- Brep is a collection of (optionally trimmed) surfaces

```
# Assume that myBrep is a variable that stores an object of data type Brep

# Get the number of surfaces that contained inside the brep object
print myBrep.Surfaces.Count

# Retrieve a specific surface from the brep
mySurface = myBrep.Surfaces[0]
```

Brep: Create loft surfaces

- Brep is a collection of (optionally trimmed) surfaces

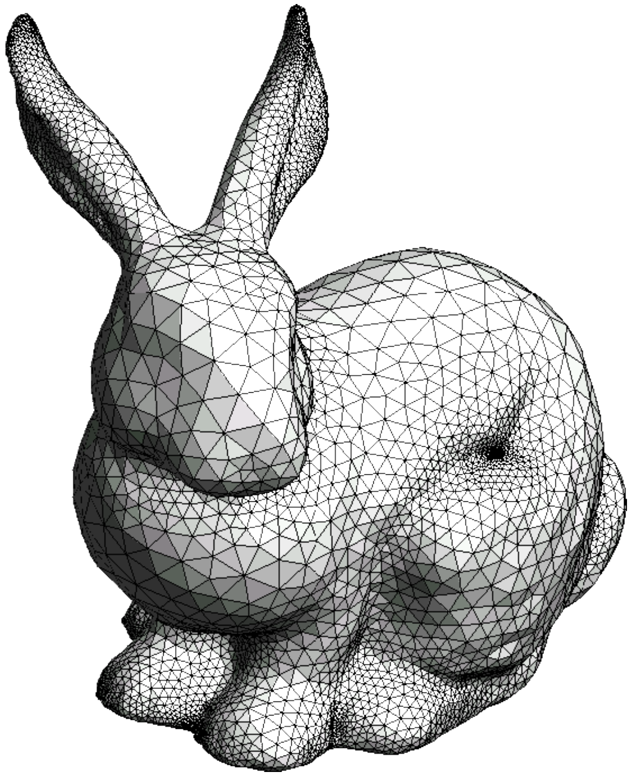
```
import Rhino.Geometry as rg

# Assume that myCurves is a list of objects of type Curve

breps = rg.Brep.CreateFromLoft(myCurves, rg.Point3d.Unset, rg.Point3d.Unset, \
rg.LoftType.Normal, False)

# The static method CreateFromLoft actually returns an array of Brep objects
# If we are confident that we will get only one brep, which contains only one surface, ...
# ... then we can retrieve that surface using the square-bracket operator

myLoftSurface = breps[0].Surfaces[0]
```

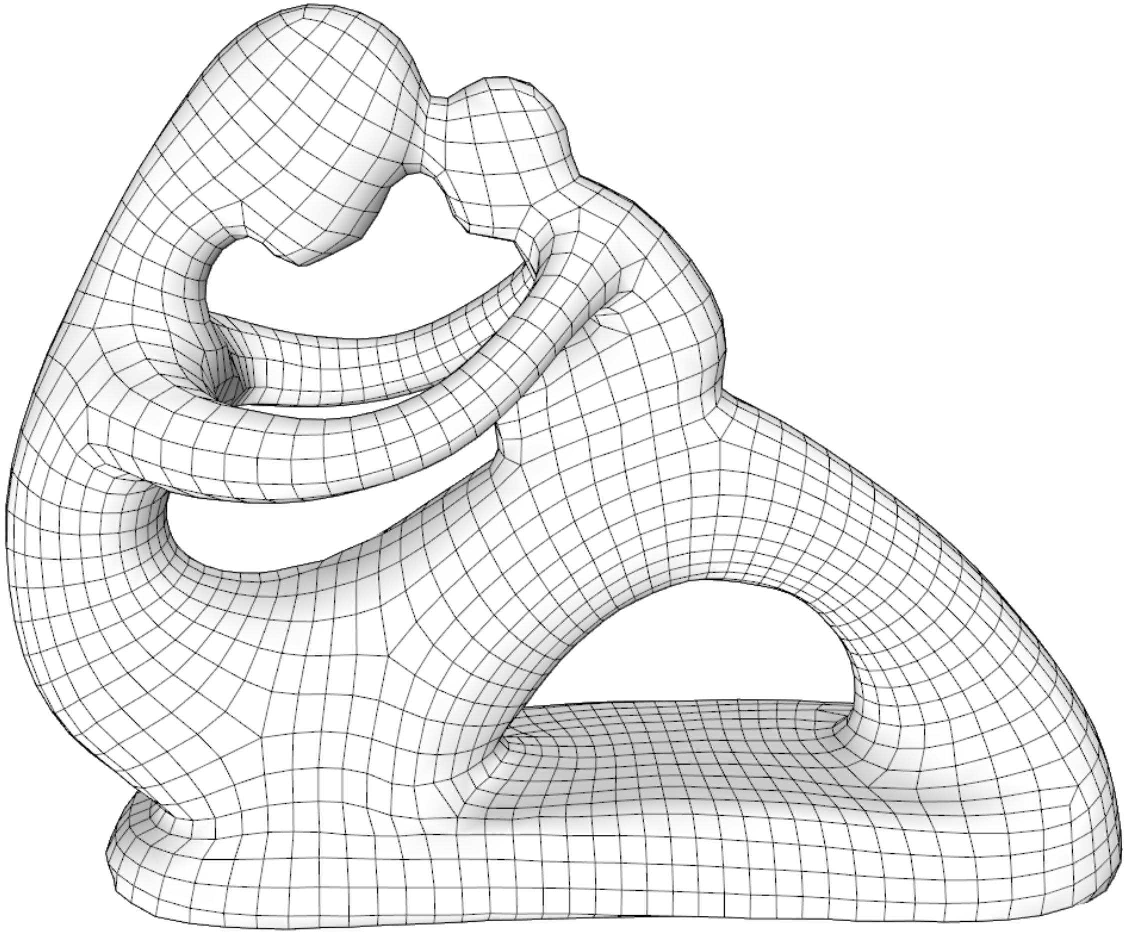
Meshes

What is a mesh?

- Mesh is a general way to describe arbitrary 3D shape using ONLY polygons
- A mesh can contains only triangular faces, quad faces, or polygonal faces, or a mixture of these things

Note: smooth shading can be applied on a mesh so that it appears to be smooth when being displayed/rendered. But the underlying geometry is **still polygonal** (which is usually evident when looking at the silhouette of the model)

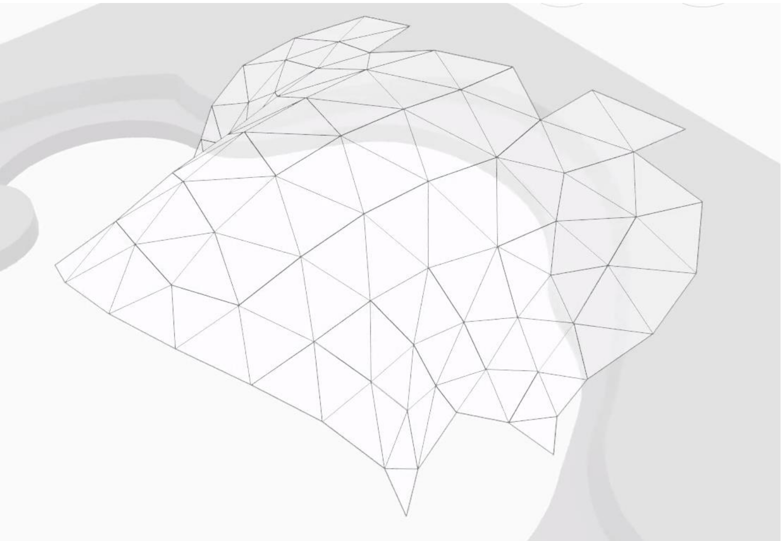


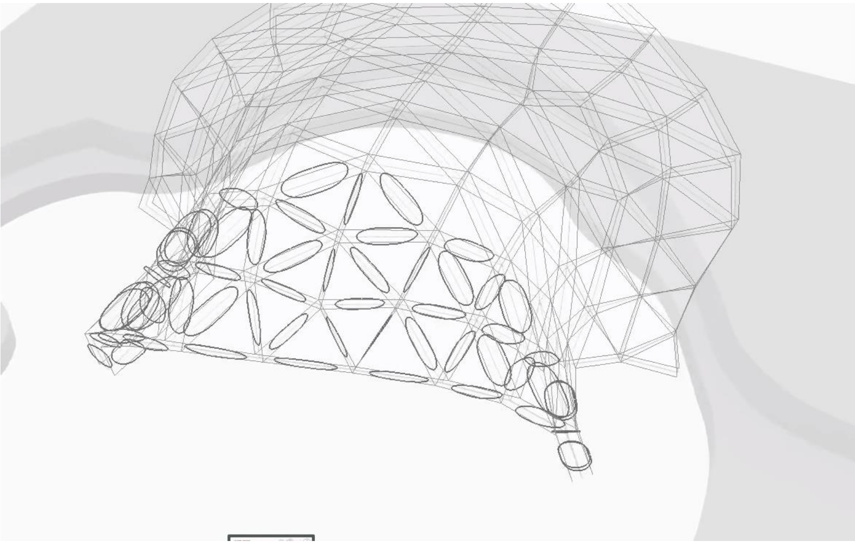


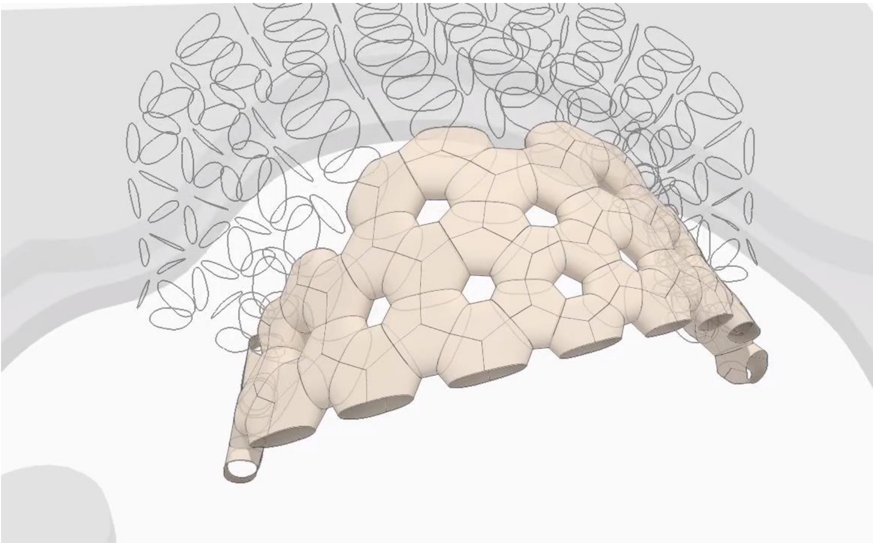
Mesh in computational design

- In computational design, meshes are useful not only because they can describe arbitrary shapes. But also because they explicitly contains topological information (which vertex connects to which vertex), which makes them a good tool for certain kinds of computational design logic (e.g. panels, truss, plates, etc.)

ICD/ITKE Research Pavilion 2015-2016







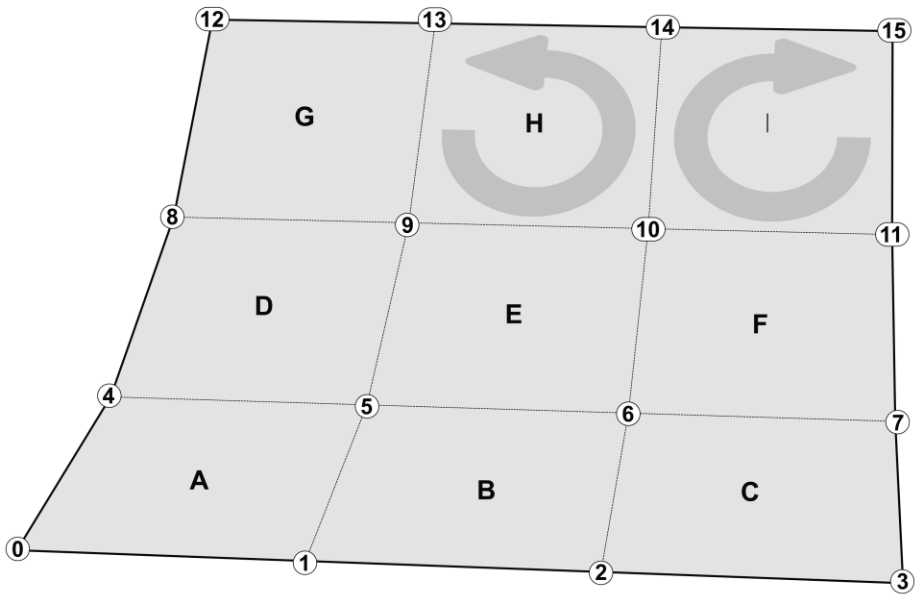
Triangular mesh → Generate intermediate geometries based on the triangular faces and their inter-relationship → Generate final architectural design geometries

Meshes in Rhino

- In Computer Graphics, there exist many methods and variations to describe a mesh. Rhino uses one of these.
- A (Rhino) mesh contains:
  - A list of vertices (each of which is a Point3f object)
  - A list of MeshFace objects, each of which contains three (or four) integers that specify the indices of the vertices that make up the triangular (or quad) face.
  - The sidedness of each face on the order of the vertices (clockwise vs. counter-clockwise)

Vertex array	Face array	Resulting mesh

```
 0 = {0.0, 0.0, 0.0}    A = { 0, 1, 5, 4}
 1 = {1.0, 0.0, 0.0}    B = { 1, 2, 6, 5}
 2 = {2.0, 0.0, 0.0}    C = { 2, 3, 7, 6}
 3 = {3.0, 0.0, 0.0}    D = { 4, 5, 9, 8}
 4 = {0.0, 0.8, 0.3}    E = { 5, 6,10, 9}
 5 = {1.0, 0.8, 0.3}    F = { 6, 7,11,10}
 6 = {2.0, 0.8, 0.3}    G = { 8, 9,13,12}
 7 = {3.0, 0.8, 0.3}    H = { 9,10,14,13}
 8 = {0.0, 1.4, 0.9}    I = {11,10,14,15}
 9 = {1.0, 1.4, 0.9}
10 = {2.0, 1.4, 0.9}
11 = {3.0, 1.4, 0.9}
12 = {0.0, 2.0, 1.5}
13 = {1.0, 2.0, 1.5}
14 = {2.0, 2.0, 1.5}
15 = {3.0, 2.0, 1.5}
```

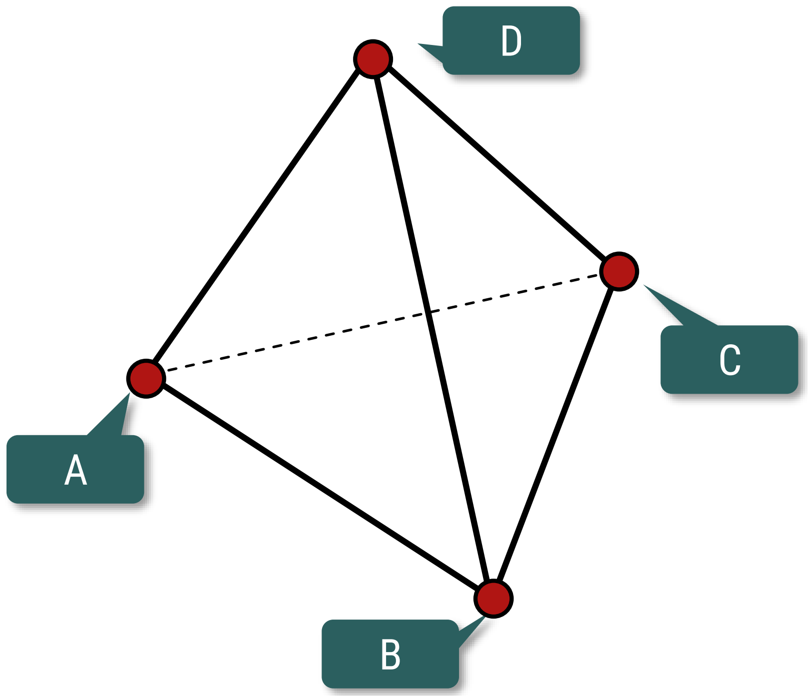


Example: Creating a mesh from scratch

Please open file *TetrahedronMesh.gh*

```
import Rhino.Geometry as rg

mesh = rg.Mesh() # Create an empty Mesh object

# Manually add the vertices
mesh.Vertices.Add(iA)
mesh.Vertices.Add(iB)
mesh.Vertices.Add(iC)
mesh.Vertices.Add(iD)

# Manually add the faces
mesh.Faces.AddFace(rg.MeshFace(2, 1, 0)) # C, B, A
mesh.Faces.AddFace(rg.MeshFace(0, 1, 3)) # A, B, D
mesh.Faces.AddFace(rg.MeshFace(1, 2, 3)) # B, C, D
mesh.Faces.AddFace(rg.MeshFace(2, 0, 3)) # C, A, D

oMesh = mesh
```

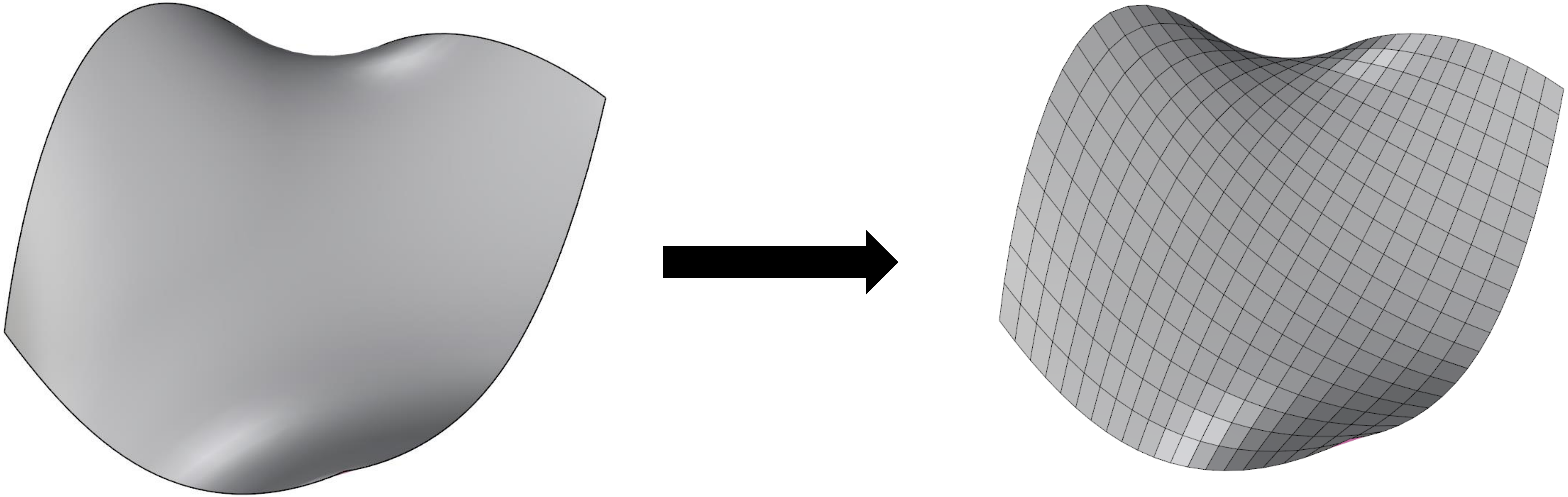
```
INPUTS:
iA, iB, iC, iD: Point3d

OUTPUTS:
oMesh
```



Example: Create a mesh based on a UV surface

- We can build a mesh from a UV surface
- The mesh vertices are based on a 2D grid, on the UV-coordinates of the surface

(We did this routine in the previous lecture)



Tip: Turn on Flat Shading to clearly see the polygonal faces of the result quad mesh!

Example: Create a mesh based on a UV surface

Please open file *UVSurfaceToMesh.gh*

```
import Rhino.Geometry as rg

iSurface.SetDomain(0, rg.Interval(0.0, 1.0))
iSurface.SetDomain(1, rg.Interval(0.0, 1.0))

mesh = rg.Mesh() # First, Create an empty mesh object

# Create vertices for the mesh
for i in range(0, iUDivision + 1):
    for j in range(0, iVDivision + 1):
        u = i / iUDivision
        v = j / iVDivision
        mesh.Vertices.Add(iSurface.PointAt(u, v))

# This utility function compute the vertex index knowing the row and column indices
def GerVertexIndex(i, j):
    return i * (iVDivision + 1) + j

# Create the mesh faces. For each face, we need to obtain the indices of the four
relevant vertices
for i in range(0, iUDivision):
    for j in range(0, iVDivision):
        v1 = GerVertexIndex(i, j)
        v2 = GerVertexIndex(i + 1, j)
        v3 = GerVertexIndex(i + 1, j + 1)
        v4 = GerVertexIndex(i, j + 1)
        mesh.Faces.AddFace(rg.MeshFace(v1, v2, v3, v4))

oGeometry = mesh # Finally, output the mesh, yay!!!
```

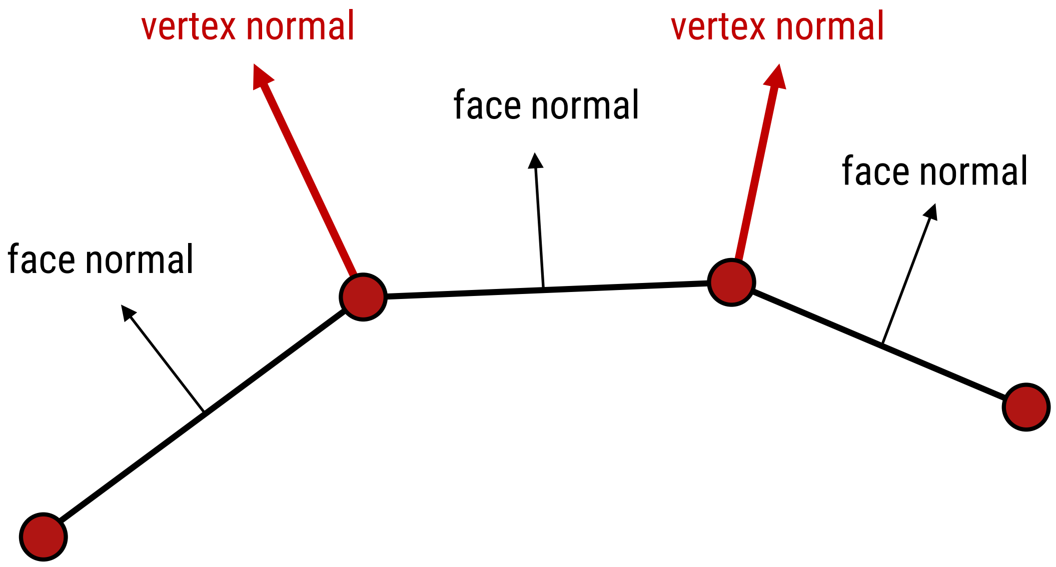
```
INPUTS:
iSurface: Surface
iUDivision: int
iVDivision: int

OUTPUTS:
oGeometry
```

The previous example shows how to create a quad mesh from the UV surface

Quick exercise to do in class:
Modify the Python codes so that it produce a triangular mesh instead
(Note: there are more than one way to generate triangular pattern on the surface)

Vertex Normals

Vertex normals



- Each mesh face has a normal vector, obviously
- But we can also define a normal vector for each vertex too
- A vertex normal is the average of the surrounding face normals
- A vertex normal approximates the orientation of the surface (recall that a mesh is a discreet way to represent a continuous, usually smooth surface)
- (Just FYI, The smooth shading algorithm uses the vertex normals to make the surface appear smooth when displayed)

Compute the vertex normals

- Usually, vertex normals are computed automatically
- If vertex normals are not available. We can simply invoke the function `ComputeNormals` once

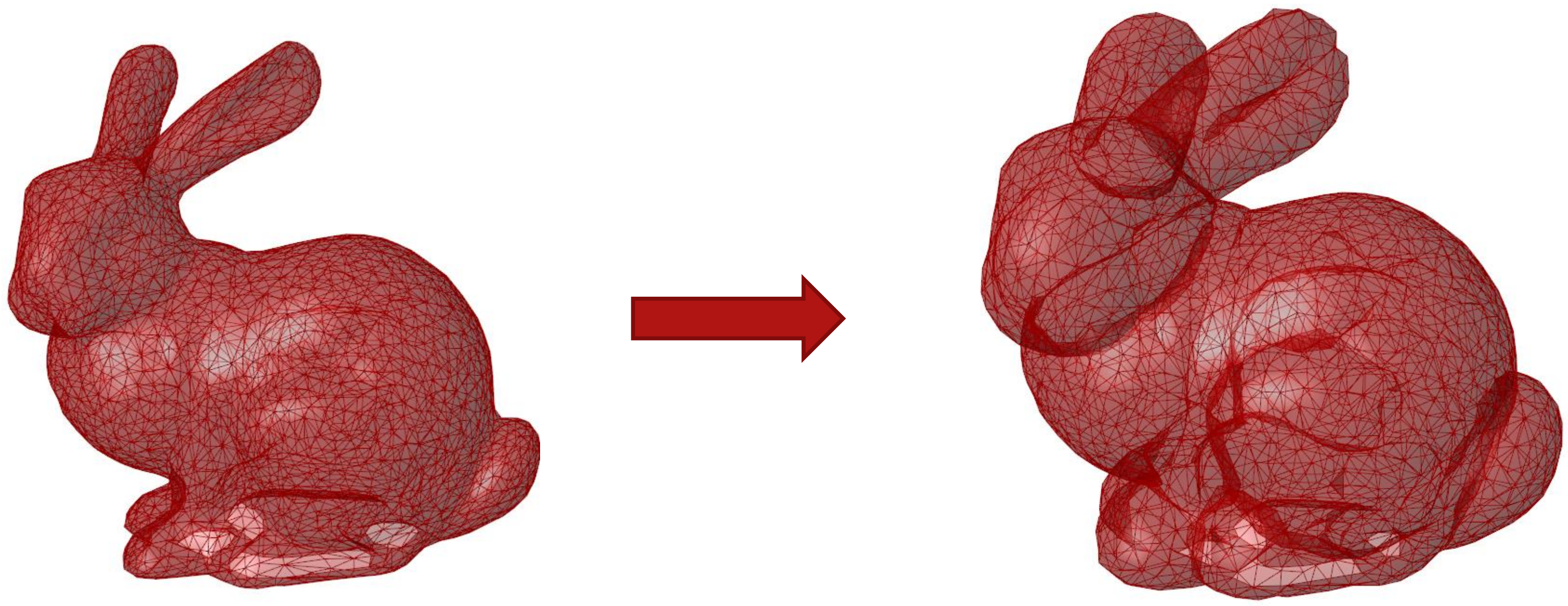
```
myMesh.Normals.ComputeNormals()
```

- We can access the normal (which is a Vector3f object) of each vertex using the square-bracket operator

```
firstVertexNormal = myMesh.Normals[1]
```

Example: Inflate a mesh

Inflate the Stanford bunny mesh by offsetting each vertex along its normal vector by a pre-specified amount



Example: Inflate a mesh

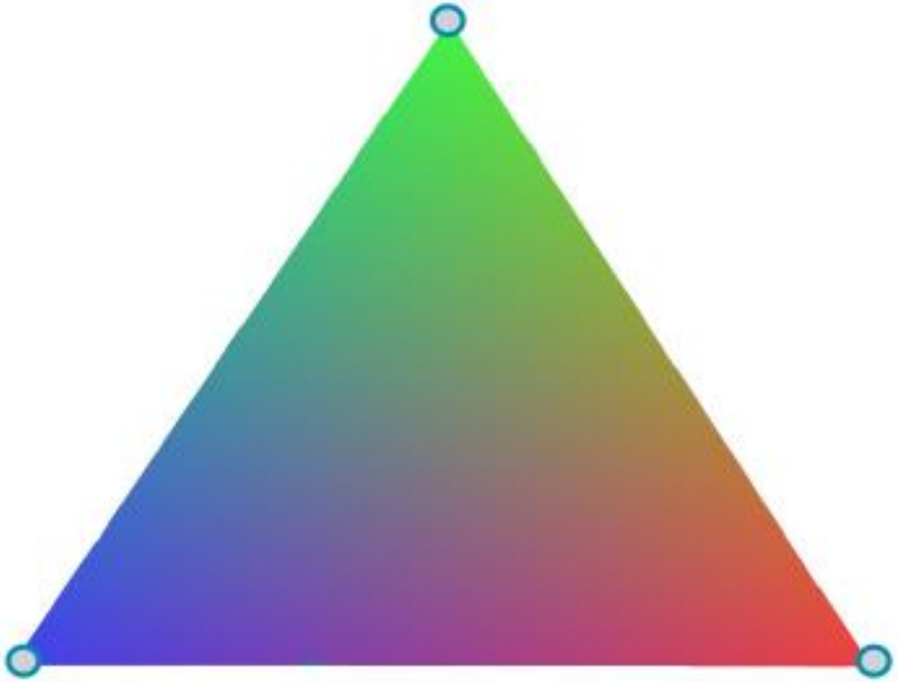
Please open file
InflateMesh.gh

```
import Rhino.Geometry as rg

for i in range(iMesh.Vertices.Count):
    # Notice that we have to "convert" from Point3f to Point3d
    position = rg.Point3d(iMesh.Vertices[i])

    # Notice that we have to "convert" from Vector3f to Vector3d
    normal = rg.Vector3d(iMesh.Normals[i])

    # The reason for the "conversion" is because Point3f and  Vector3d
does not support math operators (e.g. + or *)
    newPosition = position + normal * iInflateDistance

    iMesh.Vertices.SetVertex(i, newPosition)

oGeometry = iMesh
```

```
INPUTS:
iMesh: Mesh
iInflateDistance: float

OUTPUTS:
oGeometry
```

Vertex Colors

- We can assign a color to each vertex
- A triangular faces will be colored by blending the colors of the three vertices, resulting in overall smooth color gradient across the face.



Example: Assign random colors to mesh vertices

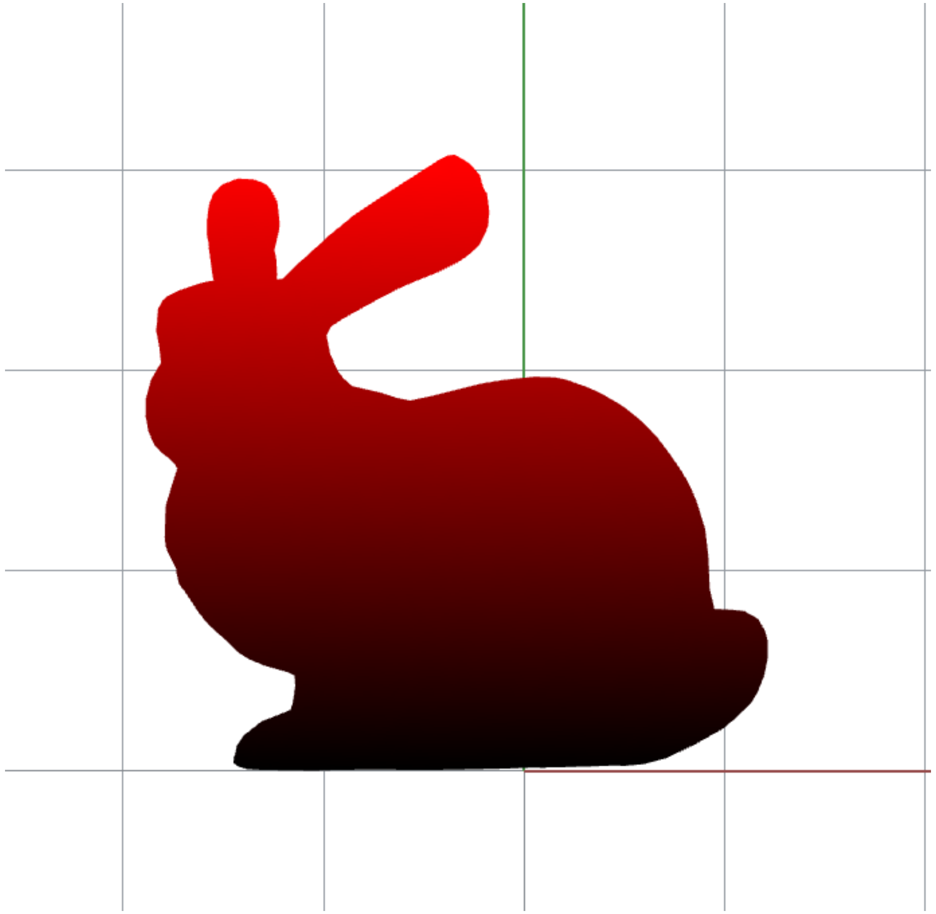
Please open file
VertexColors.gh

```python
import Rhino.Geometry as rg
from System.Drawing import Color
import random

for i in range(iMesh.Vertices.Count):
    red = random.uniform(0, 255)
    green = random.uniform(0, 255)
    blue = random.uniform(0, 255)
    randomColor = Color.FromArgb(255, red, green, blue)
    iMesh.VertexColors.SetColor(i, randomColor)

oMesh = iMesh
```

INPUTS:
iMesh: Mesh

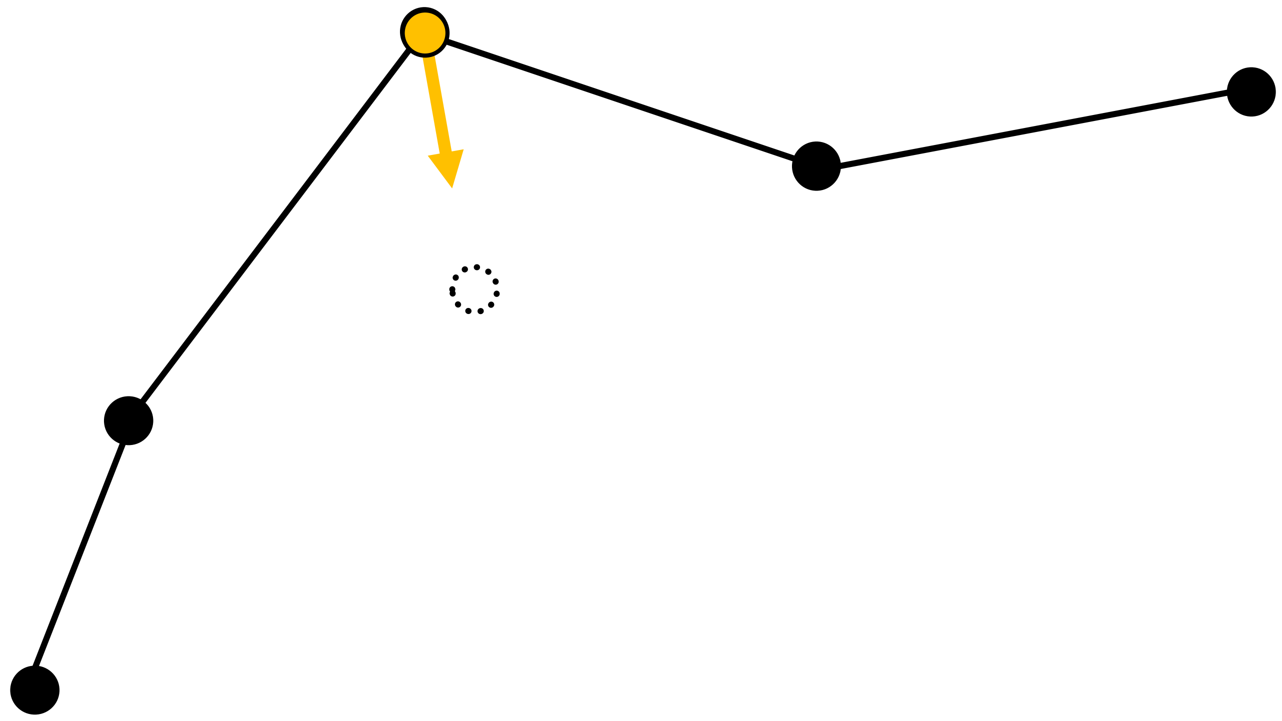
OUTPUTS:
oMesh

5-minute exercise:
Assigning vertex colors based on z-coordinates



Mesh Topology

Example: Mesh smoothing



Key Idea: Move each vertex toward the AVERAGE of all the neighbor vertices

Example: Mesh smoothing


```
import Rhino.Geometry as rg

for k in range(iIterations):
    smoothMesh = rg.Mesh()

    for face in iMesh.Faces:
        smoothMesh.Faces.AddFace(face)

    for i in range(iMesh.Vertices.Count):
        neighborIndices = iMesh.TopologyVertices.ConnectedTopologyVertices(i)
        averageOfNeighbors = rg.Point3d(0.0, 0.0, 0.0)
        for j in neighborIndices:
            averageOfNeighbors += rg.Point3d(iMesh.Vertices[j])

        averageOfNeighbors /= len(neighborIndices);
        smoothVertex = 0.5 * rg.Point3d(iMesh.Vertices[i]) + 0.5 * averageOfNeighbors
        smoothMesh.Vertices.Add(smoothVertex)

    iMesh = smoothMesh;

oSmoothMesh = iMesh;
```