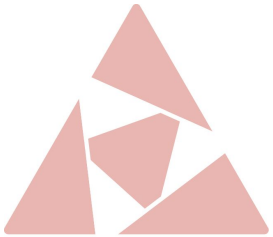
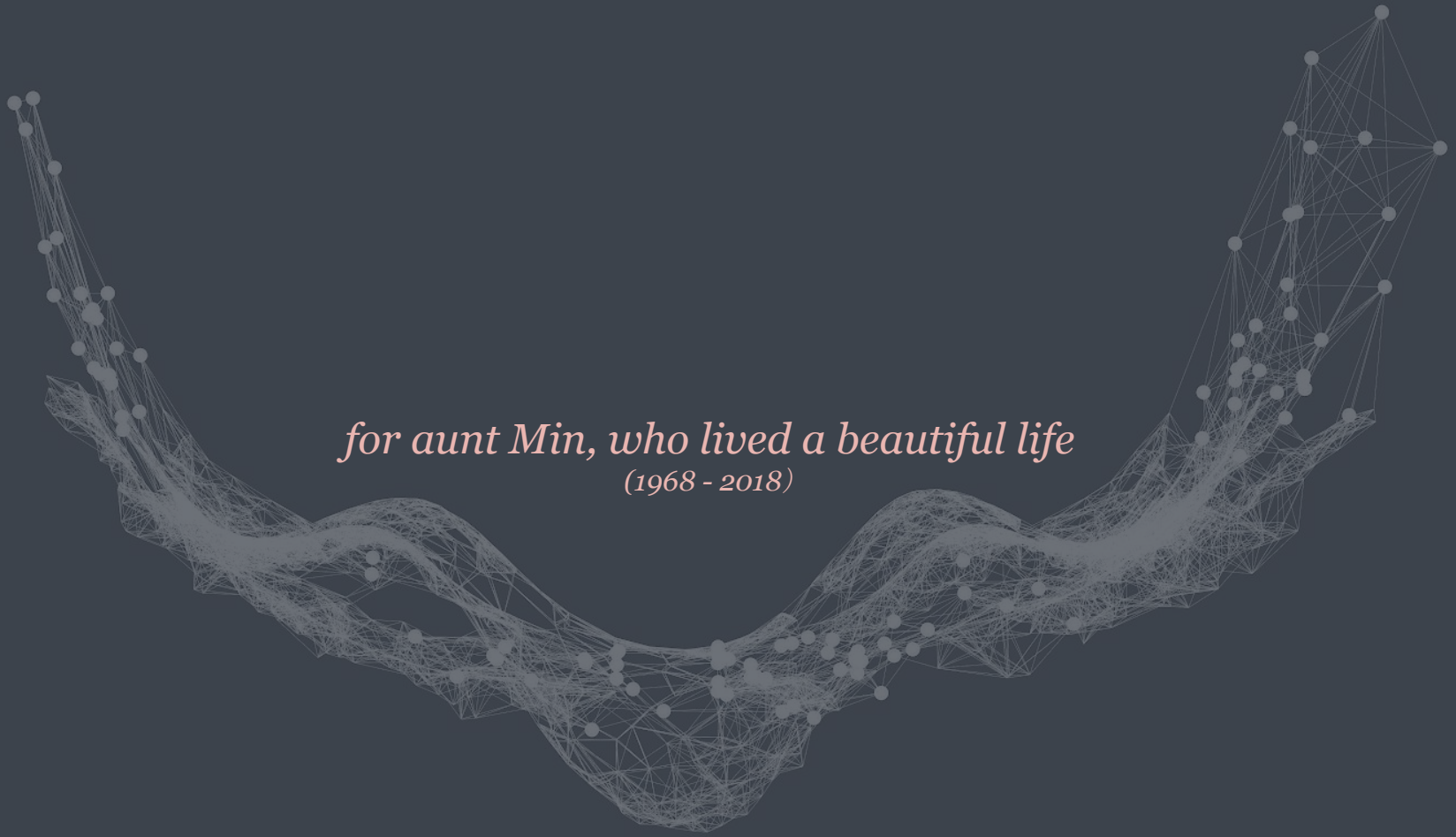


A ROSE IS A ROSE

THEN A DEATH IS NOT A DEATH

SHIYI PENG | MDES TECH 2020
GUANGYU DU | MDES TECH 2020

SCI 6338 | INTRODUCTION TO COMPUTATIONAL DESIGN - BURIAL MASK



for aunt Min, who lived a beautiful life

(1968 - 2018)

A Rose is a Rose

SCI 6338 | INTRODUCTION TO COMPUTATIONAL DESIGN - BURIAL MASK

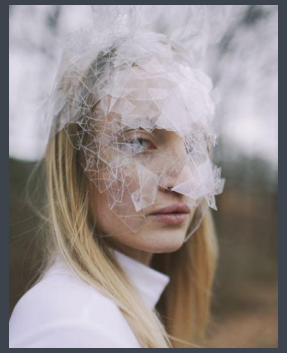
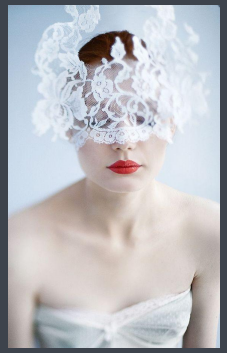
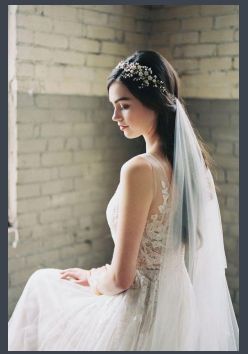
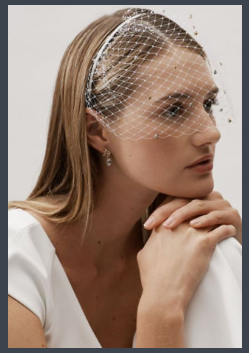
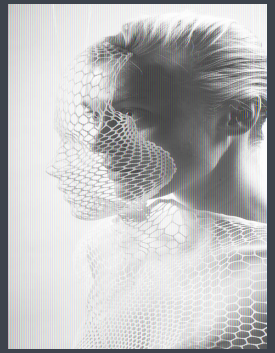
The rose is a rose,
And was always a rose.
But the theory now goes
That the apple's a rose,
And the pear is, and so's
The plum, I suppose.
The dear only knows
What will next prove a rose.
You, of course, are a rose -
But were always a rose.

Robert Lee Frost
(1875-1963)

A Rose is a Rose

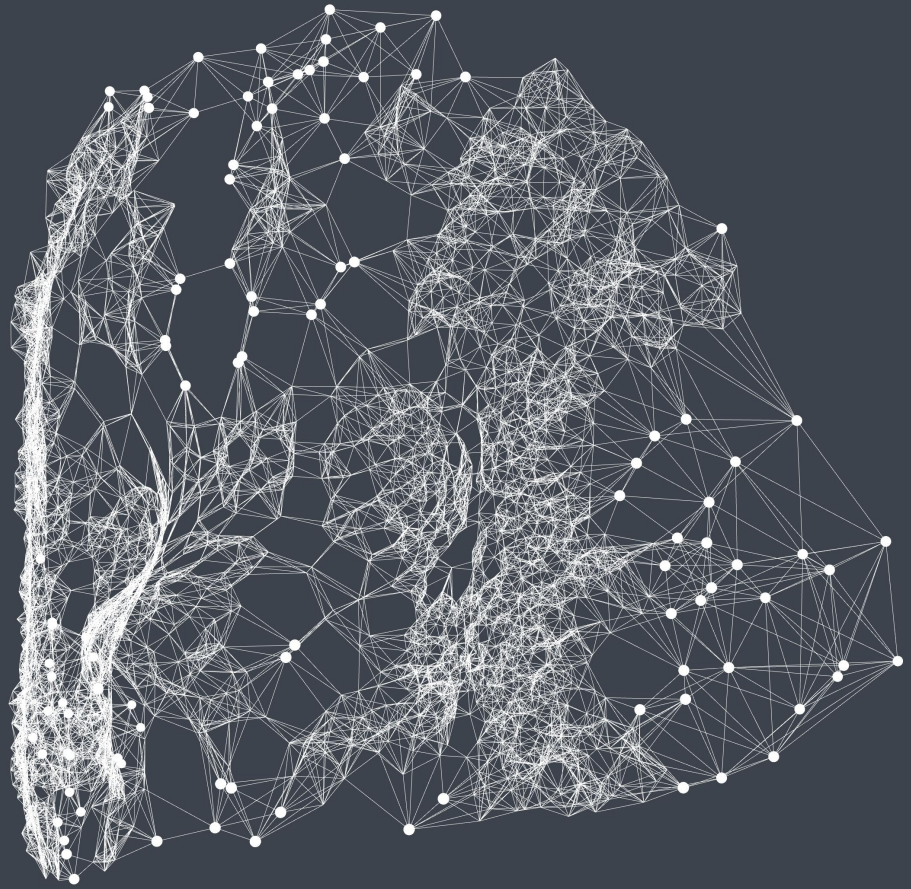
SCI 6338 | INTRODUCTION TO COMPUTATIONAL DESIGN - BURIAL MASK

We believe that a burial mask is not made of cold nor rigid stone, but it is warm and soft yarn, through which you get the first impression of the afterworld and with which you begin a new journey.

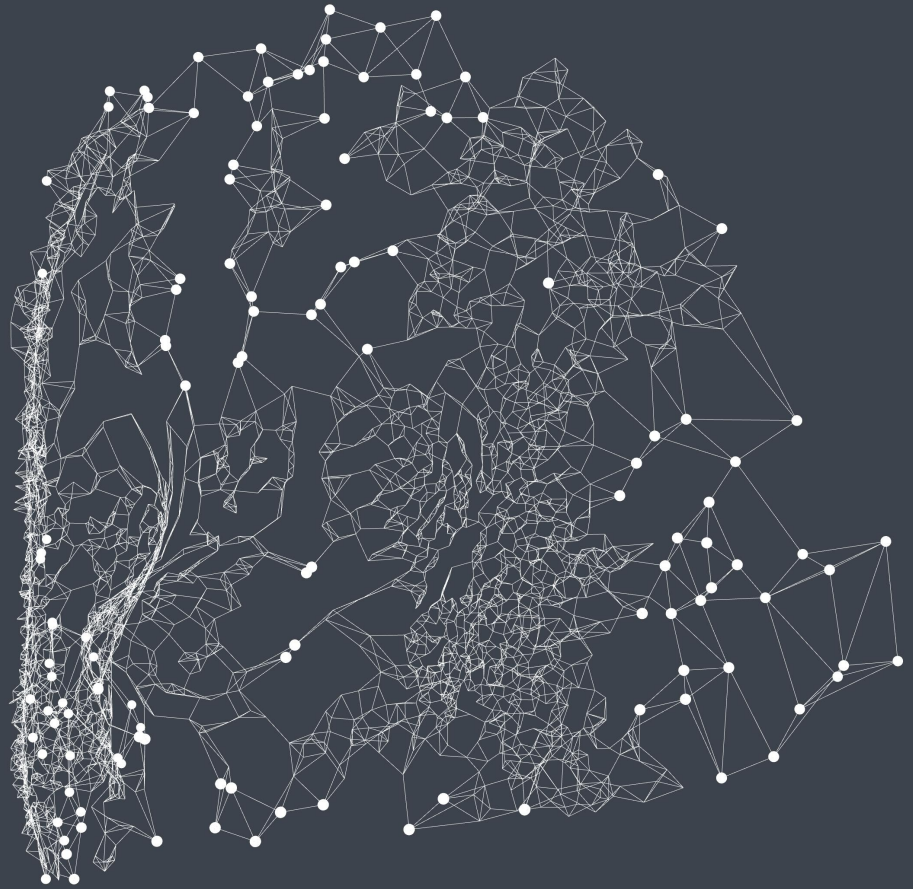
A Rose is a Rose

SCI 6338 | INTRODUCTION TO COMPUTATIONAL DESIGN - BURIAL MASK



A Rose is a Rose

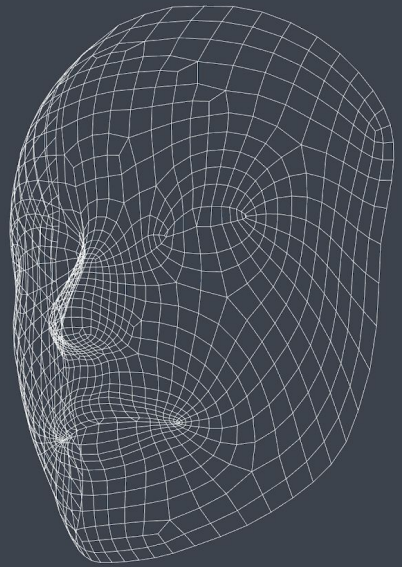
SCI 6338 | INTRODUCTION TO COMPUTATIONAL DESIGN - BURIAL MASK



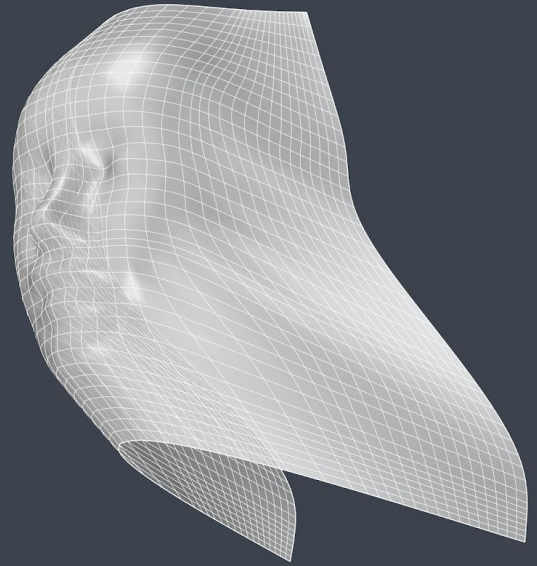
A Rose is a Rose

SCI 6338 | INTRODUCTION TO COMPUTATIONAL DESIGN - BURIAL MASK

Acquisition



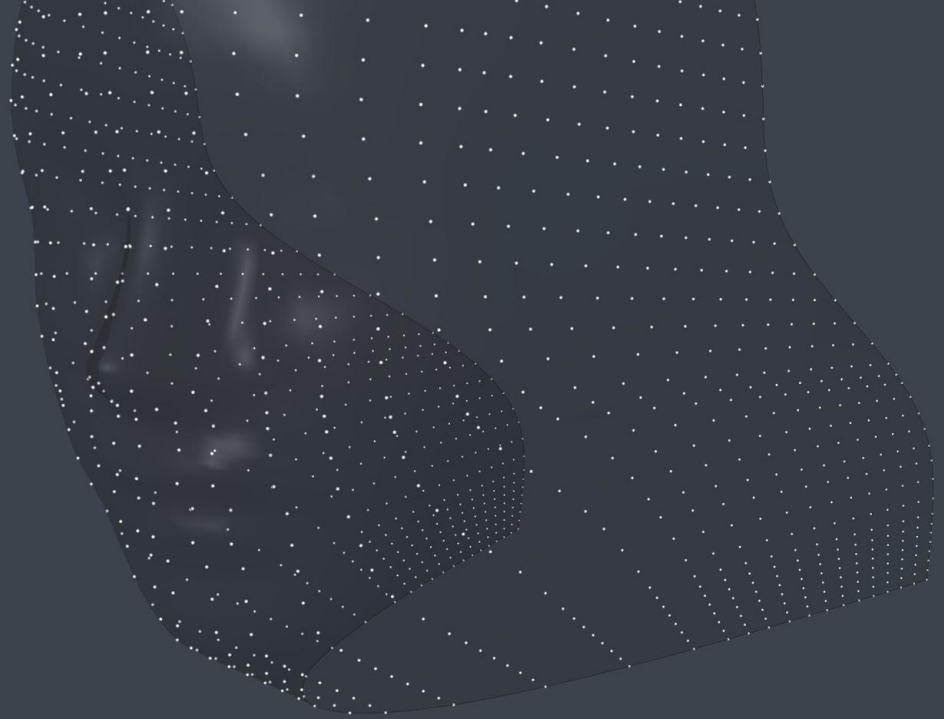
Starting Mesh from C4D

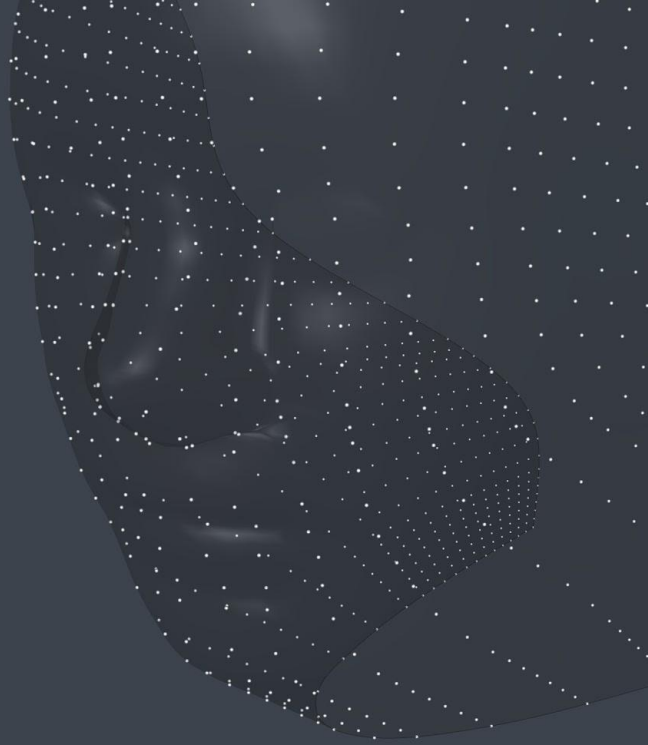


Patch and Rebuild Surface in Rhino

Deformation

Offset





Apply mean curvature flow with offset

```
offsetPoints.Add(p + n * (-cv.Mean) * off);
```

Offset = 0.0

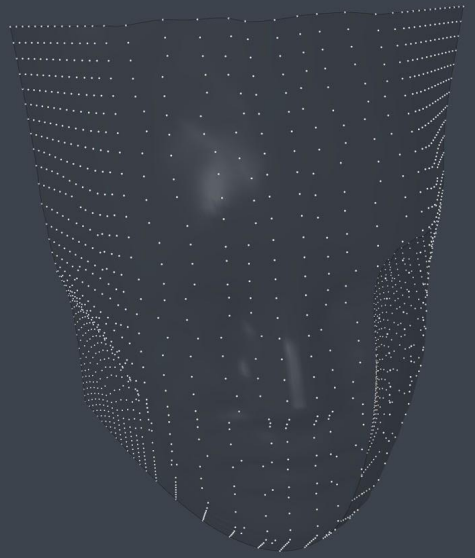
Offset = 6.0

Deformation

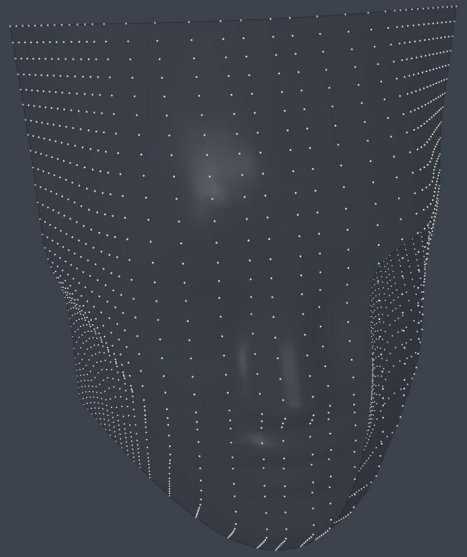
Distortion

Points move along
principal curvatures directions

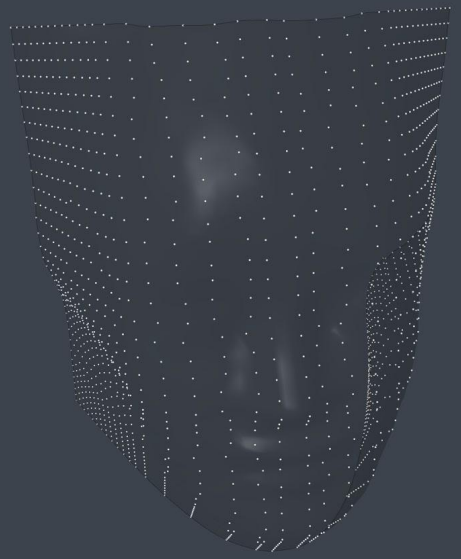
pt += (k1dir + k2dir) * torque;



Torque = -1.5



Torque = 0.0

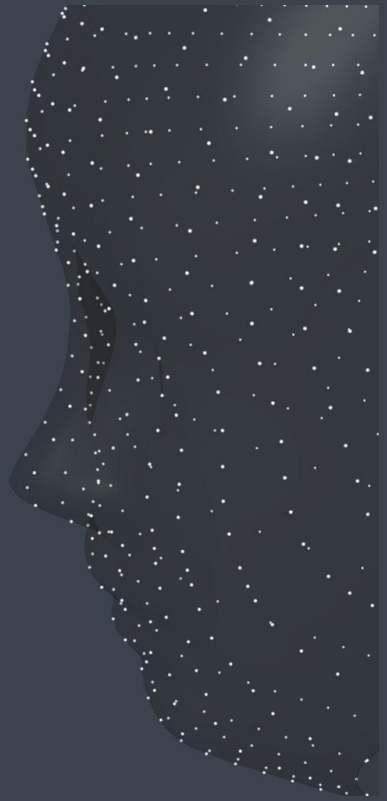


Torque = 1.0

Deformation

Force on focus point

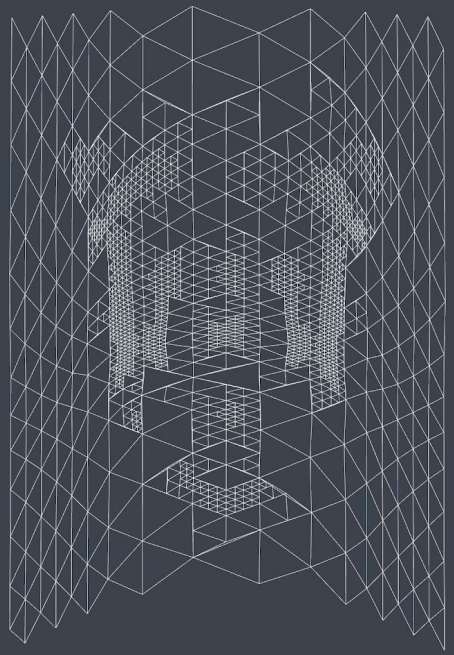
Focus at 0.15
Force = 30.0
Offset = 2.0



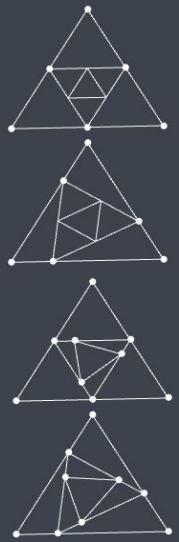
Focus at 0.5
Force = 30.0
Offset = 2.0

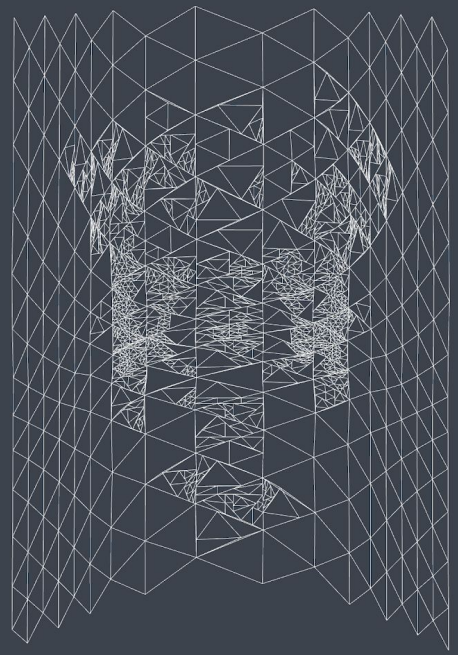
```
double uu = srf.Domain(0).NormalizedParameterAt(u);
double vv = srf.Domain(1).NormalizedParameterAt(v);

 // find the focus point along midline according to user input
double nu = uu - 0.5;
double nv = vv - focus;
double dd = nu * nu + nv * nv;

off = (0.1 + Math.Exp(-force * dd)) * offset;
```

Discretization



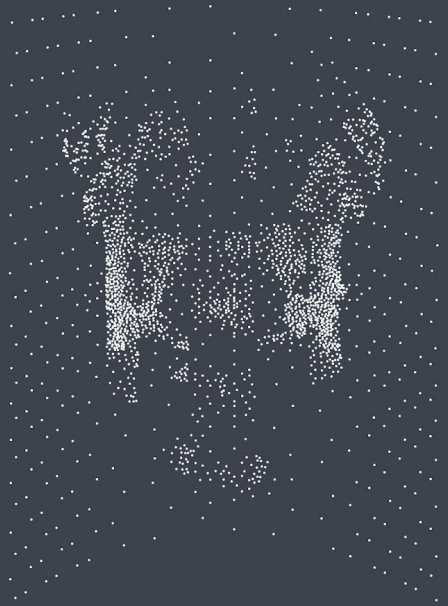
Triangle Adaptive Subdivision



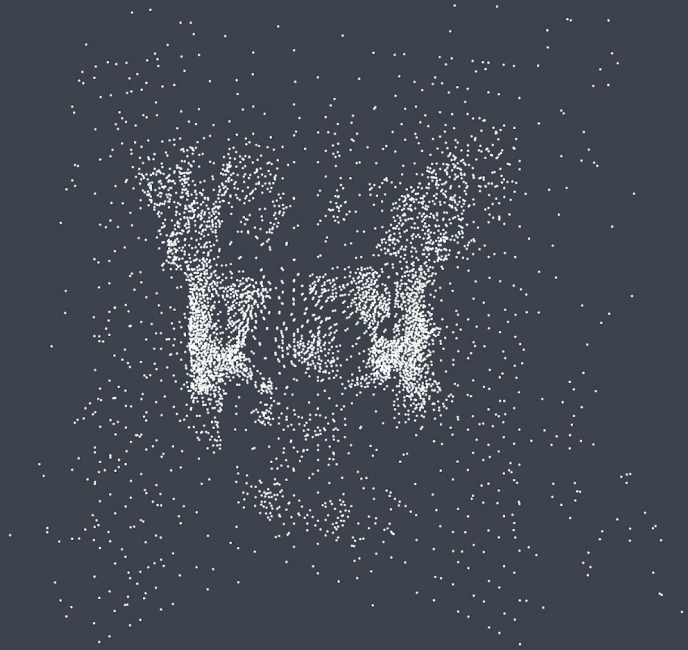


Apply Different Patterns

Discretization

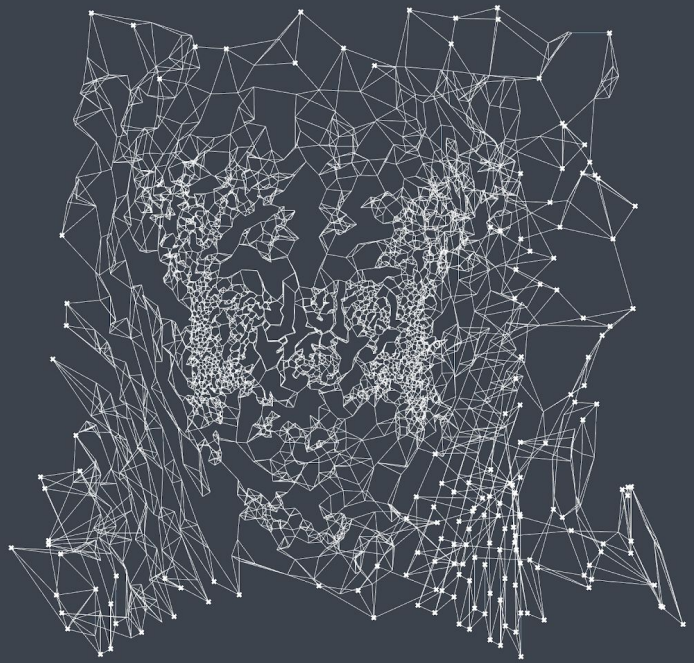


Extract Triangle Center Points

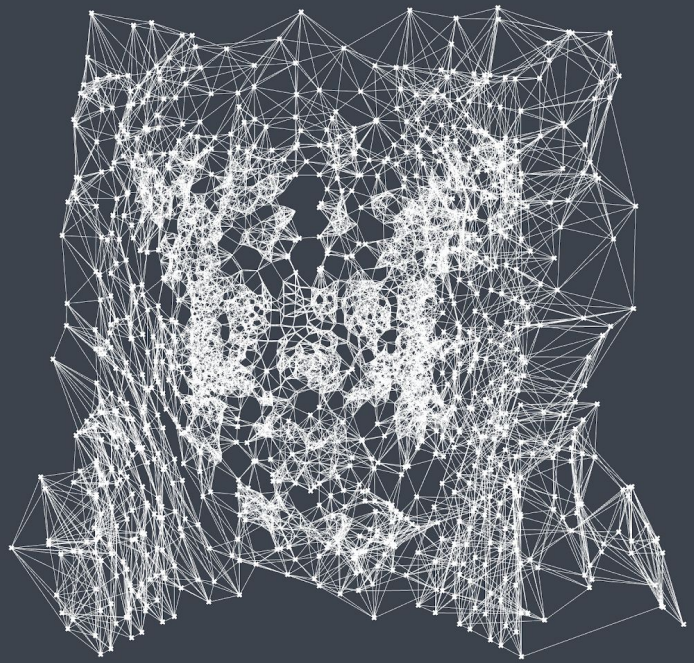


Center Points Offset

Post-Processing

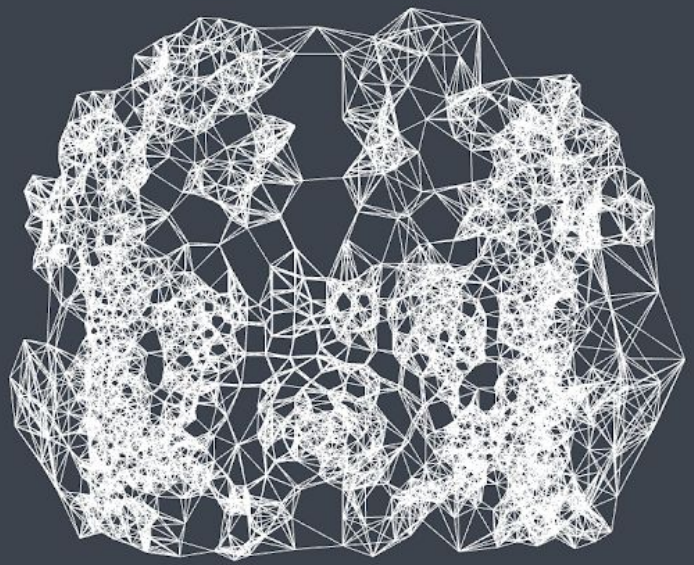


Find and Connect the nearest N neighbors

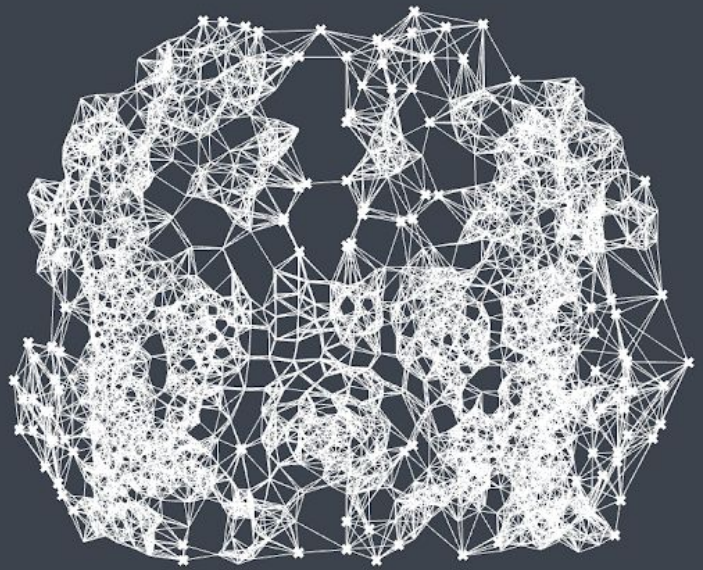


Adjust N to get different results

Post-Processing



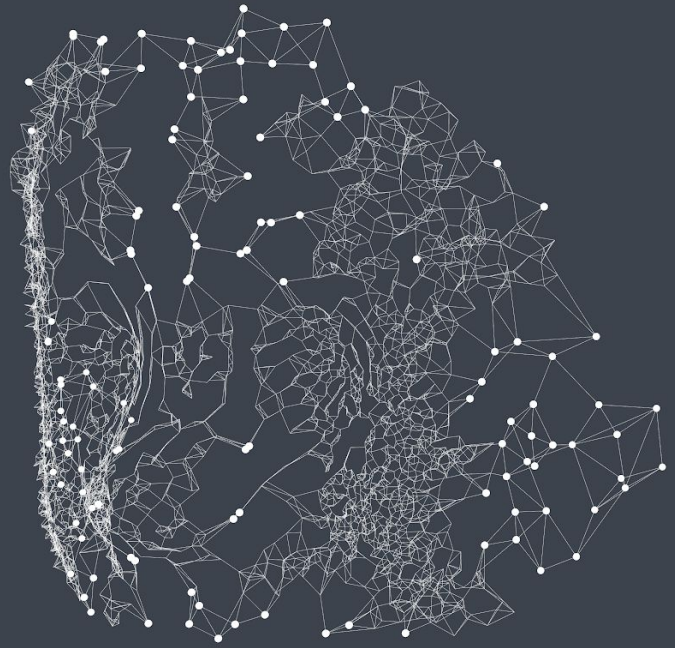
Trim Surface Boundary



Select Rose Nodes in the low-density area

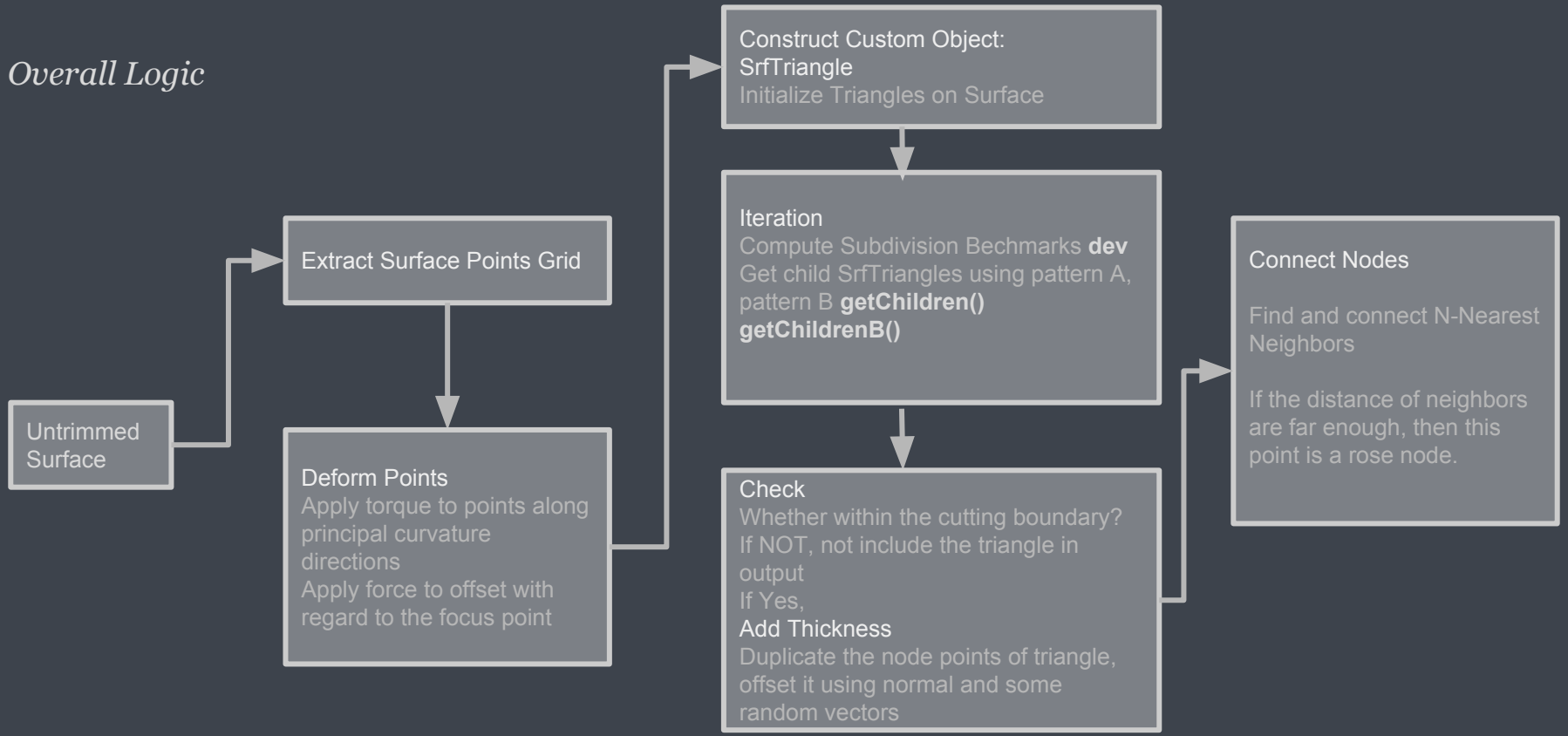
Highlights

Thickness



Organic pattern made by straight lines

Overall Logic



Acquisition *Deformation* *Discretization* *Post-Processing*

Code - Deformation

```
double u0 = srf.Domain(0).Min;
double u1 = srf.Domain(0).Max;

double v0 = srf.Domain(1).Min;
double v1 = srf.Domain(1).Max;

double du = (u1 - u0) / (uCount - 1.0);
double dv = (v1 - v0) / (vCount - 1.0);

List<Point3d> points = new List<Point3d>();
List<Vector3d> normals = new List<Vector3d>();
List<Point3d> offsetPoints = new List<Point3d>();
List<Point2d> points2d = new List<Point2d>();

for (int i = 0; j < vCount; ++j)
{
 for (int j = 0; i < uCount; ++i)
 {

   double u = u0 + i * du;
   double v = v0 + j * dv;
```

```
// transfer to interval (0,1)
    double uu = srf.Domain(0).NormalizedParameterAt(u);
    double vv = srf.Domain(1).NormalizedParameterAt(v);

    // find the focus point along midline according to user input
    double nu = uu - 0.5;
    double nv = vv - focus;

    Point3d p = srf.PointAt(u, v);
    Point2d pp = new Point2d(uu, vv);

    Vector3d n = srf.NormalAt(u, v);
    SurfaceCurvature cv = srf.CurvatureAt(u, v);

    // find the principal curvature directions at current uv
    Vector3d k1dir = cv.Direction(0);
    Vector3d k2dir = cv.Direction(1);
```

```
// apply torque to points along principal curvature directions
      /* make it symmetric
      if uCount is odd, make the midline points static*/
      if (uCount % 2 == 1)
      {
       if (i != uCount / 2)
       {
         p += (k2dir + k1dir) * torque;
       }
      }
       // if uCount is even, make the two lines of points in the middle static
      else
      {
       if (i != uCount / 2 && i != (uCount / 2) - 1)
       {
         p += (k2dir + k1dir) * torque;
       }
      }
```

```
  // apply force to offset with regard to the focus point
        double dd = nu * nu + nv * nv;
        double off = (0.1 + Math.Exp(-force * dd)) * offset;

        points2d.Add(pp);
        points.Add(p);
        offsetPoints.Add(p + n * (-cv.Mean) * off); //apply mean curvature flow with
offset
       }
      }


      Surface offsetSurface = NurbsSurface.CreateThroughPoints(offsetPoints, vCount,
uCount, 3, 3, false, false);

      OFFSETpts = offsetPoints;
      OFFSETsrf = offsetSurface;
      UCOUNT = uCount;
      VCOUNT = vCount;
      PTS2D = points2d;
```

Code - Discretization (Main)

```
// Final Triangles
  List<SrfTriangle> almostFlatTriangles = new List<SrfTriangle>();

  // Input & Output Triangles
  List<SrfTriangle> inputTriangles = new List<SrfTriangle>();
  List<SrfTriangle> outputTriangles = new List<SrfTriangle>();
  List<Point2d> srfpts2d = new List<Point2d>();

  foreach (Point3d pt in offpts)
  {
   srfpts2d.Add(new Point2d(pt.X, pt.Y));
  }

  // Initialize Triangles on Surface
  List<SrfTriangle> srfTris = getSrfTriangles(uCount, vCount, srfpts2d, type);

  // Iteration
  inputTriangles.AddRange(srfTris);

  for (int i = 0; i < maxIter; ++i)

  {
   //Update Benchmarks
   double[] GauDev = ComputeGaussianDeviation(srf, inputTriangles);
   double dev = GauDev[1] * multiply;

outputTriangles.Clear();
     int j = 0;
     foreach(SrfTriangle tri in inputTriangles){

      // Subdivide, Using different patterns
      if (tri.Deviation(srf) > dev ){
       Random random = new Random(seed + j * 101);
       int choice = random.Next(0, 100);
       if (choice > 30){
        SrfTriangle[] triChildren = tri.getChildrenB();
        outputTriangles.AddRange(triChildren);
       }
       else{
        SrfTriangle[] triChildren = tri.getChildren();
        outputTriangles.AddRange(triChildren);
       }
      }
      else {
       outputTriangles.Add(tri);
      }
      j++;
     }
```

```
List<SrfTriangle> swap = inputTriangles;
    inputTriangles = outputTriangles;
    outputTriangles = swap;
  }

  almostFlatTriangles = outputTriangles;
  List <Line> cutLines = new List<Line>();
  List <Point3d> triCenters = new List<Point3d>();
  List <Point2d> triCenters2d = new List<Point2d>();
  List <Vector3d> triCentersNormal = new List<Vector3d>();
  List <Surface> triSrfs = new List<Surface>();
  List <Point3d> offsetPoints = new List<Point3d>();

  int ran = 0;
  foreach (SrfTriangle tri in almostFlatTriangles)
  {

    Point2d uv = tri.SrfTriCenter2d(srf);
    double nu = srf.Domain(0).NormalizedParameterAt(uv.X);
    double nv = srf.Domain(1).NormalizedParameterAt(uv.Y);
    double du = nu - cutMidU;
    double dv = nv - cutMidV;
```

```
/// Cutting the Mask
    if( (du * du + dv * dv ) < radius  ){

      //Output Lines and Points
      cutLines.AddRange(tri.SrfTriLines(srf));
      Point3d center = tri.SrfTriCenter(srf);
      triCenters.Add(center);

      //triCenters2d.Add(uv);

      Vector3d n = srf.NormalAt(uv.X, uv.Y);
      n.Unitize();
      triCentersNormal.Add(n);

      // Offset center pts, get random result in thickness
      Vector3d offset = n * ( thick + (du * du + dv * dv) * 200);

      Random random = new Random(seed + ran * 103);
      Random random0 = new Random(seed + ran * 107);
      Random random1 = new Random(seed + ran * 109);
      Random random2 = new Random(seed + ran * 101);

      Vector3d ranv = new Vector3d(random0.Next(0, 100), random1.Next(0, 100),
random2.Next(0, 100));
      ranv.Unitize();
```

```
        Vector3d offset2 = (offset + offset.Length * ranv) * 0.5;

        Point3d offcenter = center + offset;
        Point3d offcenter2 = center + offset2;


        triCenters.Add(offcenter2);
        offset2.Unitize();
        triCentersNormal.Add(offset2);

        //This is used when you want the offset becomes less random
        //triCenters.Add(offcenter);


        // Output Srfs
        //This is used when you want some srf fragments appear in the result, feels
like roses
        /*
        int choice = random.Next(0, 100);
        if(choice > choiceRandom){
        triSrfs.AddRange(tri.SrfTriSrfB(srf));
        }
        */
      }

      ran += 1;
    }
```

```
CUTLINES = cutLines;
TRICENTERS = triCenters;
//TRISRFS = triSrfs;
TRINORMALS = triCentersNormal;
//TRICENTERS2D = triCenters2d;
```

Code - Discretization (Class)

```
    public double[] ComputeGaussianDeviation(Surface srf, List<SrfTriangle>
triangles)
 {
  // We choose not to use Gaussian because the result is not good

  double DevSum = 0.0;
  //double GauSum = 0.0;

  foreach(SrfTriangle tri in triangles)
  {
   //GauSum += tri.Gaussian(srf);
   DevSum += tri.Deviation(srf);
  }

  double[] result = new double[3];
  //result[0] = GauSum * (1.0 / triangles.Count);
  result[1] = DevSum * (1.0 / triangles.Count);

  return result;
 }
```

```
public List<Point2d> getSrfPoints2d(int uCount, int vCount)
 {

  double u0 = 0.0;
  double u1 = 1.0;

  double v0 = 0.0;
  double v1 = 1.0;

  double du = (u1 - u0) / (uCount - 1.0);
  double dv = (v1 - v0) / (vCount - 1.0);

  List<Point2d> points = new List<Point2d>();

  for(int j = 0; j < vCount; ++j) {
   for(int i = 0; i < uCount; ++i) {

    double u = u0 + i * du;
    double v = v0 + j * dv;

    points.Add(new Point2d(u, v));
   }
  }
  return points;
 }
```

```
public List<SrfTriangle> getSrfTriangles(int uCount, int vCount, List<Point2d>
pts, bool type)
 {
  List<SrfTriangle> triangles = new List<SrfTriangle>();
  //Pattern 1
  if (type == true){
   for(int j = 0; j < vCount; ++j) {
    for(int i = 0; i < uCount; ++i) {
     if ((i + j) % 2 == 0 && j + 1 < vCount ){
      // Current point
      int A = uCount * j + i;
      int B = A + uCount + 1;
      int C = A + 2;
      int D = A + uCount - 1;
      //Left Corner : 1 Triangle
      if (i - 1 < 0 && i + 2 < uCount){
       triangles.Add(new SrfTriangle(pts[A], pts[B], pts[C]));
      }
       //Majority: 2 Triangles
      else if(i - 1 >= 0 && i + 2 < uCount){
       triangles.Add(new SrfTriangle(pts[A], pts[B], pts[C]));
       triangles.Add(new SrfTriangle(pts[A], pts[D], pts[B]));
      }
  //Right Corner: 1 Triangle
      else if(i - 1 >= 0 && i + 1 < uCount){
       triangles.Add(new SrfTriangle(pts[A], pts[D], pts[B]));
      }
     }
    }
   }
  }
  // Pattern 2
 else{
  for(int j = 0; j < vCount; ++j) {
   for(int i = 0; i < uCount; ++i) {
    if ((i + j) % 2 == 1 && j > 0 ){
     // Current point
     int A = uCount * j + i;
     int B = A - uCount + 1;
     int C = A + 2;
     int D = A - uCount - 1;
     //Left Corner : 1 Triangle
     if (i - 1 < 0 && i + 2 < uCount){
      triangles.Add(new SrfTriangle(pts[A], pts[B], pts[C]));
     }
      //Majority: 2 Triangles
     else if(i - 1 >= 0 && i + 2 < uCount){
      triangles.Add(new SrfTriangle(pts[A], pts[B], pts[C]));
      triangles.Add(new SrfTriangle(pts[A], pts[D], pts[B]));
     }
      //Right Corner: 1 Triangle
     else if(i - 1 >= 0 && i + 1 < uCount){
      triangles.Add(new SrfTriangle(pts[A], pts[D], pts[B]));
     }
    }
   }
  }
  return triangles;
 }
```

```
//Declare triangle on surface class
public class SrfTriangle {

  // A.B.C Node
  public Point2d A {get;set;}
  public Point2d B {get;set;}
  public Point2d C {get;set;}


  // Constructor that takes no arguments:
  public SrfTriangle()
  {

    A = new Point2d(0, 0);
    B = new Point2d(0, 0);
    C = new Point2d(0, 0);
  }
  // Constructor that takes 6 arguments:
  public SrfTriangle(double _Au, double _Av, double _Bu, double _Bv, double
_Cu, double _Cv)
  {

    A = new Point2d(_Au, _Av);
    B = new Point2d(_Bu, _Bv);
    C = new Point2d(_Cu, _Cv);

  }
```

```
// Constructor that takes 3 arguments:
  public SrfTriangle(Point2d _A, Point2d _B, Point2d _C)
  {
    A = _A;
    B = _B;
    C = _C;
  }

  //Extract the points in surface parameter space using Normalized Parameters:
  public Point2d[] SrfTriPoints2d(Surface srf)
  {
    Point2d[] Nodes2d = new Point2d[3];
    double uMin = srf.Domain(0).Min;
    double uMax = srf.Domain(0).Max;
    double vMin = srf.Domain(1).Min;
    double vMax = srf.Domain(1).Max;

    double nAu = (uMax - uMin) * A.X + uMin;
    double nAv = (vMax - vMin) * A.Y + vMin;
    double nBu = (uMax - uMin) * B.X + uMin;
    double nBv = (vMax - vMin) * B.Y + vMin;
    double nCu = (uMax - uMin) * C.X + uMin;
    double nCv = (vMax - vMin) * C.Y + vMin;

    Nodes2d[0] = new Point2d(nAu, nAv);
    Nodes2d[1] = new Point2d(nBu, nBv);
    Nodes2d[2] = new Point2d(nCu, nCv);

    return Nodes2d;
  }
```

```
//Extract the actual Point3d of SrfTriangle Nodes using Normalized Parameters:
  public Point3d[] SrfTriPoints(Surface srf)
  {
   Point3d[] Nodes = new Point3d[3];
   Point2d[] Nodes2d = this.SrfTriPoints2d(srf);

   Point2d A = Nodes2d[0];
   Point2d B = Nodes2d[1];
   Point2d C = Nodes2d[2];

   Nodes[0] = srf.PointAt(A.X, A.Y);
   Nodes[1] = srf.PointAt(B.X, B.Y);
   Nodes[2] = srf.PointAt(C.X, C.Y);

   return Nodes;
  }

  //Compute the Curve Edges of SrfTriangle:
  public Curve[] SrfTriCurves(Surface srf)
  {
   Curve[] SrfTriCurves = new Curve[3];
   Point2d[] Nodes2d = this.SrfTriPoints2d(srf);

   Point2d A = Nodes2d[0];
   Point2d B = Nodes2d[1];
   Point2d C = Nodes2d[2];

SrfTriCurves[0] = srf.ShortPath(A, B, 0.001);
    SrfTriCurves[1] = srf.ShortPath(B, C, 0.001);
    SrfTriCurves[2] = srf.ShortPath(C, A, 0.001);

    return SrfTriCurves;
   }


  //Compute the straight Line Edges of SrfTriangle:
  public Line[] SrfTriLines(Surface srf)
  {
   Line[] SrfTriLines = new Line[3];
   Point3d[] Nodes = this.SrfTriPoints(srf);

   SrfTriLines[0] = new Line(Nodes[0], Nodes[1]);
   SrfTriLines[1] = new Line(Nodes[1], Nodes[2]);
   SrfTriLines[2] = new Line(Nodes[2], Nodes[0]);

   return SrfTriLines;
  }

  //Compute the triangle Srf using nodes
  public Surface SrfTriSrf(Surface srf)
  {
   Point3d[] Nodes = this.SrfTriPoints(srf);
   Surface planeSrf = NurbsSurface.CreateFromCorners(Nodes[0], Nodes[1],
Nodes[2]);
    return planeSrf;
  }
```

```
public Surface[] SrfTriSrfB(Surface srf)

  {
   SrfTriangle[] childs = this.getChildrenB();

   Surface[] planeSrf = new Surface[3];
   planeSrf[0] = childs[1].SrfTriSrf(srf);
   planeSrf[1] = childs[2].SrfTriSrf(srf);
   planeSrf[2] = childs[3].SrfTriSrf(srf);

   return planeSrf;
  }

  //Compute Central Point2d of a triangular on a srf
  public Point2d SrfTriCenter2d(Surface srf)

  {
   Point2d[] Nodes2d = this.SrfTriPoints2d(srf);
   Point2d center2d = (Nodes2d[0] + Nodes2d[1] + Nodes2d[2]) * (1 / 3.0);
   return center2d;
  }

  //Compute Central Point3d of a triangular on a srf
  public Point3d SrfTriCenter(Surface srf)
  {
   Point2d center2d = this.SrfTriCenter2d(srf);
   Point3d center = srf.PointAt(center2d.X, center2d.Y);
   return center;
  }
```

```
public Point2d[] samplePoints2d(Surface srf)
  {
   Point2d[] Nodes = this.SrfTriPoints2d(srf);
   Point2d midAB = (Nodes[0] + Nodes[1]) * 0.5;
   Point2d midBC = (Nodes[1] + Nodes[2]) * 0.5;
   Point2d midCA = (Nodes[2] + Nodes[0]) * 0.5;

   Point2d nA = (Nodes[0] + midAB + midCA ) * (1.0 / 3.0);
   Point2d nB = (Nodes[1] + midAB + midBC ) * (1.0 / 3.0);
   Point2d nC = (Nodes[2] + midCA + midBC ) * (1.0 / 3.0);

   Point2d[] sample2d = new Point2d[7];
   sample2d[0] = this.SrfTriCenter2d(srf);
   sample2d[1] = midAB;
   sample2d[2] = midBC;
   sample2d[3] = midCA;
   sample2d[4] = nA;
   sample2d[5] = nB;
   sample2d[6] = nC;
   return sample2d;
  }
```

```
public Point3d[] samplePoints(Surface srf)
  {
    Point2d[] sample2d = this.samplePoints2d(srf);
    Point3d[] sample = new Point3d[sample2d.Length];
    for(int i = 0, max = sample2d.Length; i < max; i++) {
      sample[i] = srf.PointAt(sample2d[i].X, sample2d[i].Y);
    }
    return sample;
  }

  //Compute Relative Perimeter Length
  public double perimeter2d()
  {
    return (A.X - B.X) * (A.X - B.X) + (A.Y - B.Y) * (A.Y - B.Y) +
      (B.X - C.X) * (B.X - C.X) + (B.Y - C.Y) * (B.Y - C.Y) +
      (A.X - C.X) * (A.X - C.X) + (A.Y - C.Y) * (A.X - C.Y);
  }

 public double perimeter(Surface srf)
  {
    Point3d[] Nodes = this.SrfTriPoints(srf);
    double dist = (Nodes[0].X - Nodes[1].X) * (Nodes[0].X - Nodes[1].X) +
(Nodes[0].Y - Nodes[1].Y) * (Nodes[0].Y - Nodes[1].Y)
      + (Nodes[1].X - Nodes[2].X) * (Nodes[1].X - Nodes[2].X) + (Nodes[1].Y -
Nodes[2].Y) * (Nodes[1].Y - Nodes[2].Y)
      + (Nodes[0].X - Nodes[2].X) * (Nodes[0].X - Nodes[2].X) + (Nodes[0].Y -
Nodes[2].Y) * (Nodes[0].Y - Nodes[2].Y);
    return dist;
  }
```

```
//Compute Relative Gaussian Curvature at the Central Point
public double Gaussian(Surface srf)
{
 Point2d[] sample2d = this.samplePoints2d(srf);
 double gauSum = 0.0;
 foreach (Point2d pt in sample2d)
 {
  SurfaceCurvature cv = srf.CurvatureAt(pt.X, pt.Y);
  gauSum += Math.Abs(cv.Gaussian);
 }
 gauSum *= (1.0 / this.perimeter(srf));
 return gauSum;
}

//Compute the Deviation (Dist to its triangular plane) at the Central Point
public double Deviation(Surface srf)
{
 Point3d[] sample = this.samplePoints(srf);
 Point3d[] Nodes = this.SrfTriPoints(srf);
 Plane triPlane = new Plane(Nodes[0], Nodes[1], Nodes[2]);

 double devSum = 0.0;
 foreach (Point3d pt in sample)
 {
  devSum += Math.Abs(triPlane.DistanceTo(this.SrfTriCenter(srf)));
 }
 devSum *= (1.0 / this.perimeter(srf));
 return devSum;
}
```

```
//Produce Child SrfTriangles - Pattern A
public SrfTriangle[] getChildren()
{
  Point2d midAB = (A + B) * 0.5;
  Point2d midBC = (B + C) * 0.5;
  Point2d midCA = (C + A) * 0.5;


  SrfTriangle[] childs = new SrfTriangle[4];
  childs[0] = new SrfTriangle(midAB, midBC, midCA);
  childs[1] = new SrfTriangle(A, midAB, midCA);
  childs[2] = new SrfTriangle(midAB, B, midBC);
  childs[3] = new SrfTriangle(midCA, midBC, C);

  return childs;
}

//Produce Child SrfTriangles - Pattern B
public SrfTriangle[] getChildrenB()
{
  Point2d midAB = (A * 2 + B ) * (1.0 / 3.0);
  Point2d midBC = (B * 2 + C ) * (1.0 / 3.0);
  Point2d midCA = (C * 2 + A ) * (1.0 / 3.0);


  SrfTriangle[] childs = new SrfTriangle[4];
  childs[0] = new SrfTriangle(midAB, midBC, midCA);
  childs[1] = new SrfTriangle(A, midAB, midCA);
  childs[2] = new SrfTriangle(midAB, B, midBC);
  childs[3] = new SrfTriangle(midCA, midBC, C);

  return childs;
 }


}
```

Code - PostProcessing

```
List<List<int>> neighbors = new List<List<int>> ();
  List<List<double>> dists = new List<List<double>> ();

  // Traverse point list to select point center
  for (int j = 0; j < pts.Count; j++){

    // Store the index, distance of each neighbor to current center point
    List <int> neighborsLocal = new List<int>();
    List <double> distsLocal = new List<double>();

    for (int i = 0; i < pts.Count; i++){
      // not calculating itself
      if(i != j)
      {
        double dist = pts[j].DistanceTo(pts[i]);
        neighborUpdate(i, dist, n, ref neighborsLocal, ref distsLocal);
      }
    }

    // Add the Local Neighbors to the Global Neighbors List
    neighbors.Add(neighborsLocal);
    dists.Add(distsLocal);
  }

// Clean Neighbors, Ensure no more than N lines is connected to each point
  //neighborClean(n, ref neighbors);
```

```
    // output Rose Nodes, neighbor lines, Roses
   List <Point3d> roseNodes = new List<Point3d>();
  List<Line> lines = new List<Line>();
   List<Transform> roseTransforms = new List<Transform>();
   List<double> roseScale = new List<double>();

for (int j = 0; j < pts.Count; j++){
    Line[] neighborLines = new Line[n];
    double sum = 0;
    for (int i = 0; i < neighbors[j].Count; i++){
      Line link = new Line(pts[j], pts[neighbors[j][i]]);
      sum += link.Length;
      neighborLines[i] = link;
    }
    lines.AddRange(neighborLines);

    if (sum > roseRange ){
      roseNodes.Add(pts[j]);
      Plane frame= new Plane(pts[j], normals[j]);
      Transform tform = Transform.PlaneToPlane(Plane.WorldXY, frame);
      roseTransforms.Add(tform);
      roseScale.Add(sum * 0.005);
    }
  }

  CONNECTIONS = lines;
  ROSENODES = roseNodes;
  ROSETFORM = roseTransforms;
  ROSESCALE = roseScale;
```

```
// <Custom additional code>
  public void neighborClean(int n, ref List<List<int>> neighbors)
  {
    //Ensure Every Point is only connected to equal to N neighbors or less

    for (int j = 0; j < neighbors.Count; j++){
      for (int i = n - 1; i >= 0; i--){

        // Weather The neighbor current point links to Contain the current point or
Not
        bool isContained = false;
        for (int k = 0; k < neighbors[neighbors[j][i]].Count; k++)
        {
          if (neighbors[neighbors[j][i]][k] == j){
            isContained = true;
            break;
          }
        }
        // the neighbor that the current point links to does not contain current point
          if (!isContained)
          {
            // Not Full
            if(neighbors[neighbors[j][i]].Count < n)
            {
              neighbors[neighbors[j][i]].Add(j);
            }
              // Full
            else
            {
              neighbors[j].RemoveAt(i);
            }
          }
        }
      }
    }
```

```
public void neighborUpdate(int index, double dist, int n, ref List<int>
neighbors, ref List<double> dists)
 {
   // Empty Neighbors
   if (dists.Count == 0){
    dists.Add(dist);
    neighbors.Add(index);
    return;
   }
    // Part Empty Neighbors
   else if(dists.Count < n){
    //If Closer than the farest neighbor, starting the sort, find its place in the
neighbors list
    if(dist < dists[dists.Count - 1])
    {
     for (int i = 0; i < dists.Count; i++)
     {
      // Insert according the distance order
      if (dist < dists[i])
      {
       dists.Insert(i, dist);
       neighbors.Insert(i, index);
       break;
      }
     }
    }
     //Add to the Farest Point
    else{
     dists.Add(dist);
     neighbors.Add(index);
    }
   }
// Full Neighbors
    //If Closer than the farest neighbor, starting the sort, find its place in the
neighbors list
   else if(dist < dists[dists.Count - 1])
   {

    // Traverse neighbors
    for (int i = 0; i < dists.Count; i++)
    {
     // Insert according the distance order
     if (dist < dists[i])
     {
      dists.Insert(i, dist);
      neighbors.Insert(i, index);
      break;
     }
    }
    // Remove the farest point
    dists.RemoveAt(dists.Count - 1);
    neighbors.RemoveAt(neighbors.Count - 1);
   }
 }
```