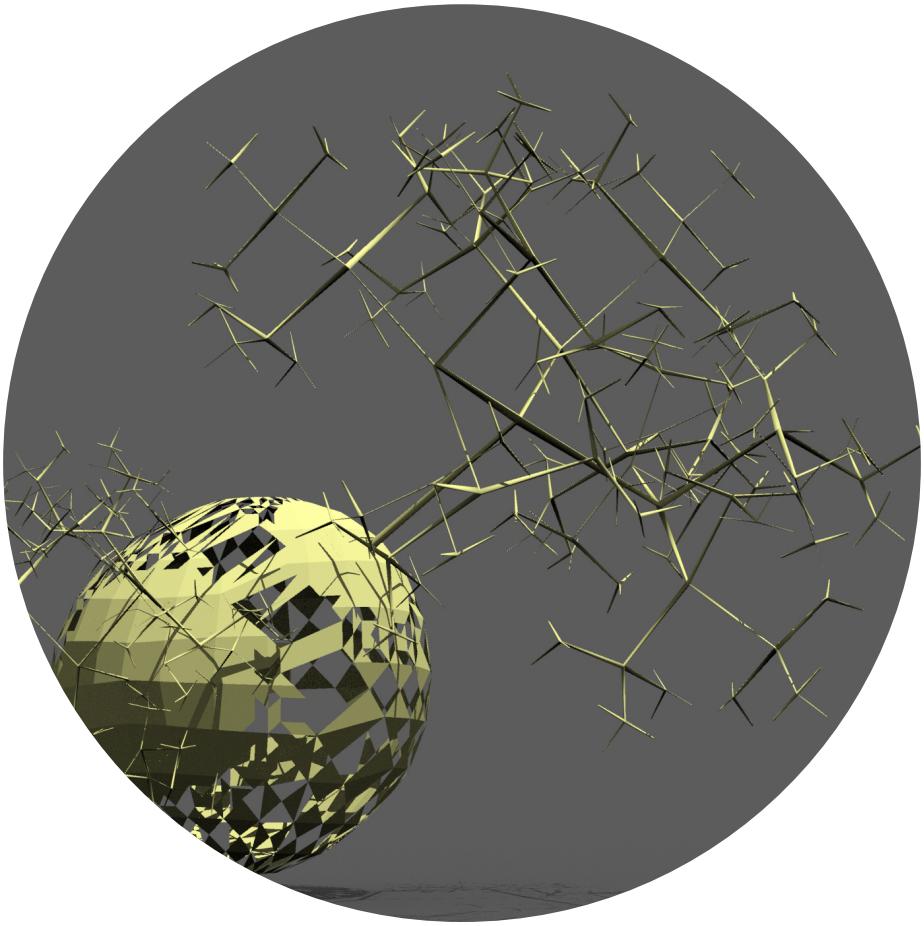


THE TREE

YUZHEN ZHANG

yz869

The script takes an input mesh and treat it as a ground plane. Four steps can be applied to subdivide, transform, sculpt the ground, and plant trees on the ground plane.

```
private void RunScript(Mesh originalMesh, List<Point3d> targetPoints, int maxTim
{
  // 01 subdivide mesh
  Mesh mesh = subdivideMesh(originalMesh, maxTimes, targetPoints, radius);
  // 02 modify mesh vertices
  mesh = modifyMesh(mesh, moveStrength, targetPoints);
  // 03 remove mesh faces
  mesh = removeFaces(mesh, cullRange, targetPoints);
  // 04 create faces to build trees
  mesh = plantTree(mesh, treePoints, treeDepth, treeAngle);
  m = mesh;
}
```

The first step, **subdivideMesh**, subdivides the input mesh several times. It cuts a quad face into 4 triangles by connecting vertices to the center point, or a triangular face into 4 triangles by connecting the mid points of the 3 edges. The subdivision is affected by **target points**, **range**, and **times**. The closer a face is to the target points, the more time it gets subdivided, and the maximum time of subdivision is **times**. The **range** controls the influence radius of the **target points**.

```
// subdivideMesh returns the subdivided inputMesh
// int times is the max time of division
// int radius is the impact strength of the target points
Mesh subdivideMesh(Mesh inputMesh, int times, List<Point3d> targetPts, int radius){
  // for every time, loop through faces and join the subdivided faces
  for (int t = 0; t < times; t++){
    Mesh newMesh = new Mesh();
    for(int f = 0; f < inputMesh.Faces.Count; f++){
      Mesh returnedMesh = subdivideFace(inputMesh, f, targetPts, t, radius);
      newMesh.Append(returnedMesh);
    }
    newMesh.Vertices.CombineIdentical(false, true);
    inputMesh = newMesh;
  }
  return inputMesh;
}

// subdivideFace returns the result mesh after subdividing inputMesh.Faces[f]
// int currentTime is the current time of division
Mesh subdivideFace(Mesh inputMesh, int f, List<Point3d> targetPts, int currentTime, int radius){
  // initialize points
  List<Point3d> ptList = new List<Point3d>();
  int num = 4;
  // set the points
  if (inputMesh.Faces[f].IsQuad){
    ptList.Add(inputMesh.Vertices[inputMesh.Faces[f].A]);
    ptList.Add(inputMesh.Vertices[inputMesh.Faces[f].B]);
    ptList.Add(inputMesh.Vertices[inputMesh.Faces[f].C]);
    ptList.Add(inputMesh.Vertices[inputMesh.Faces[f].D]);
  } else {
    ptList.Add(inputMesh.Vertices[inputMesh.Faces[f].A]);
    ptList.Add(inputMesh.Vertices[inputMesh.Faces[f].B]);
    ptList.Add(inputMesh.Vertices[inputMesh.Faces[f].C]);
    num = 3;
  }
  // get shortest distance between a vertex and a target
  double dist = ptList[0].DistanceTo(targetPts[0]);
  for(int j = 0; j < num; j++){
    double dist2 = ptList[j].DistanceTo(closestTargetPt(targetPts, ptList[j]));
    if (dist > dist2){
      dist = dist2;
    }
  }
  // map the dist to int, the shorter the larger
  int t = (int) (radius / dist);
  Mesh m = new Mesh();
  List<Point3d> newPts = new List<Point3d>();
  if (currentTime < t){
    // subdivide face
    if (inputMesh.Faces[f].IsQuad){
      newPts.Add((ptList[0] + ptList[1] + ptList[2] + ptList[3]) / 4);
      m.Vertices.Add(ptList[0]);//0
      m.Vertices.Add(ptList[1]);//1
      m.Vertices.Add(ptList[2]);//2
      m.Vertices.Add(ptList[3]);//3
      m.Vertices.Add(newPts[0]);//4

      m.Faces.AddFace(0, 1, 4);
      m.Faces.AddFace(1, 2, 4);
      m.Faces.AddFace(2, 3, 4);
      m.Faces.AddFace(3, 0, 4);

      m.Normals.ComputeNormals();
    } else {
      newPts.Add((ptList[0] + ptList[1]) / 2);
      newPts.Add((ptList[1] + ptList[2]) / 2);
      newPts.Add((ptList[2] + ptList[0]) / 2);

      m.Vertices.Add(ptList[0]);//0
      m.Vertices.Add(ptList[1]);//1
      m.Vertices.Add(ptList[2]);//2
      m.Vertices.Add(newPts[0]);//3
      m.Vertices.Add(newPts[1]);//4
      m.Vertices.Add(newPts[2]);//5

      m.Faces.AddFace(0, 3, 5);
      m.Faces.AddFace(1, 4, 3);
      m.Faces.AddFace(2, 5, 4);
      m.Faces.AddFace(3, 4, 5);

      m.Normals.ComputeNormals();
    }
  } else {
    // add the original face
    if (inputMesh.Faces[f].IsQuad){
      m.Vertices.Add(ptList[0]);//0
      m.Vertices.Add(ptList[1]);//1
      m.Vertices.Add(ptList[2]);//2
      m.Vertices.Add(ptList[3]);//3
      m.Faces.AddFace(0, 1, 2, 3);
      m.Normals.ComputeNormals();
    } else {
      m.Vertices.Add(ptList[0]);//0
      m.Vertices.Add(ptList[1]);//1
      m.Vertices.Add(ptList[2]);//2
      m.Faces.AddFace(0, 1, 2);
      m.Normals.ComputeNormals();
    }
  }
  return m;
}
```

The second step, **modifyMesh**, returns the mesh with vertices moved. It pulls the vertices of the mesh towards the closest **target point**, and the distance is controlled by the **strength** input.

```
// returns the modified mesh
// vertices are moved towards targetPts on distance in between and area
Mesh modifyMesh(Mesh inputMesh, int strength, List<Point3d> targetPts){
  for(int i = 0; i < inputMesh.Vertices.Count; i++){
    Point3d v = inputMesh.Vertices[i];
    // strength is inversely proportional to distance between target pt and vertex
    Point3d target = closestTargetPt(targetPts, v);
    double dist = v.DistanceTo(target);
    Vector3d direction = target - v;
    v = v + (strength / (dist * 10)) * direction;
    inputMesh.Vertices.SetVertex(i, v);
  }
  return inputMesh;
}
```

The third step, **removeFaces**, removes the faces whose distance to the **target points** is larger than **range**. Because it removes the face while iterating through the list of faces, it removes every other face and resulting in a perforated effect.

```
// returns the mesh with some faces removed
// remove the faces out of range
Mesh removeFaces(Mesh inputMesh, int range, List<Point3d> targetPts){
  for(int i = 0; i < inputMesh.Faces.Count; i++){
    // add vertices to a list
    List<Point3d> vList = new List<Point3d>();
    if (inputMesh.Faces[i].IsQuad){
      vList.Add(inputMesh.Vertices[inputMesh.Faces[i].A]);
      vList.Add(inputMesh.Vertices[inputMesh.Faces[i].B]);
      vList.Add(inputMesh.Vertices[inputMesh.Faces[i].C]);
      vList.Add(inputMesh.Vertices[inputMesh.Faces[i].D]);
    } else {
      vList.Add(inputMesh.Vertices[inputMesh.Faces[i].A]);
      vList.Add(inputMesh.Vertices[inputMesh.Faces[i].B]);
      vList.Add(inputMesh.Vertices[inputMesh.Faces[i].C]);
    }
    // get closest distance between vertices and target points
    double dist = vList[0].DistanceTo(targetPts[0]);
    for(int j = 0; j < vList.Count; j++){
      double dist2 = vList[j].DistanceTo(closestTargetPt(targetPts, vList[j]));
      if (dist > dist2){
        dist = dist2;
      }
    }
    // remove surface if out of range
    if (range < dist){
      inputMesh.Faces.RemoveAt(i);
    }
  }
  return inputMesh;
}
```

The fourth step, **plantTree**, create new faces to form tree-like structures. For each **target point**, it finds the face whose center location is closest to the point, and recursively generate the tree at the location of the face. The recursion is called **depth** times.

At each level, the function first subdivides the original triangle into 4 triangles by connecting center points of the edges. Then it extrudes the central triangle to the normal direction of the face to form a tree branch. The length of extrusion is controlled by the distance between the center of the original triangle and the corresponding **target point**. The farther this distance is, the longer the branch is, and the taller the whole tree is. The length is also proportional to the remaining **depth**, which means it gets shorter and shorter as the function is called recursively.

Similarly, it subdivides the center triangle again and extrudes its center to form a joint. The length of this extrusion is proportional to the distance, the remaining **depth**, and the value of **angle**. The larger the **angle** is, the longer this extrusion is, and the wider the subtrees spread. Three of the faces of this joint is used as the base triangle for subtrees.

```
// returns the mesh with trees constructed
Mesh plantTree(Mesh inputMesh, List<Point3d> targetPts, int depth, double angle){
  List<Point3d> cen = centers(inputMesh);
  Mesh treeMesh = new Mesh();
  List<int> indexs = new List<int>();
  for(int i = 0; i < targetPts.Count; i++){
    // find the cloest triangle in inputMesh for each target point
    int closestIndex = closestTargetPtIndex(cen, targetPts[i]);
    double dist = targetPts[i].DistanceTo(cen[closestIndex]);
    Point3d pt1 = inputMesh.Vertices[inputMesh.Faces[closestIndex].A];
    Point3d pt2 = inputMesh.Vertices[inputMesh.Faces[closestIndex].B];
    Point3d pt3 = inputMesh.Vertices[inputMesh.Faces[closestIndex].C];
    // add face and tree
    indexs.Add(closestIndex);
    treeMesh.Append(plantTreeFace(pt1, pt2, pt3, dist, targetPts[i], depth, angle));
  }
  // remove the faces
  for(int j = 0; j < indexs.Count; j++){
    inputMesh.Faces.RemoveAt(indexs[j]);
  }
  // add tree
  inputMesh.Append(treeMesh);
  inputMesh.Vertices.CombineIdentical(false, true);
  return inputMesh;
}

// returns the mesh of subtree for triangle
Mesh plantTreeFace(Point3d pt1, Point3d pt2, Point3d pt3, double dist, Point3d targetPt, int depth, double angle){
  Mesh m = new Mesh();
  m.Vertices.Add(pt1);//0
  m.Vertices.Add(pt2);//1
  m.Vertices.Add(pt3);//2

  // get normal
  Vector3d n = normal(pt1, pt2, pt3);

  // get 3 new points
  Point3d pt4 = (pt1 + pt2) / 2;
  Point3d pt5 = (pt2 + pt3) / 2;
  Point3d pt6 = (pt3 + pt1) / 2;
  // move 3 new points
  Vector3d move1 = n * 1 * dist * depth / 5;
  pt4 += move1;
  pt5 += move1;
  pt6 += move1;

  m.Vertices.Add(pt4);//3
  m.Vertices.Add(pt5);//4
  m.Vertices.Add(pt6);//5

  m.Faces.AddFace(0, 3, 5);
  m.Faces.AddFace(0, 1, 3);
  m.Faces.AddFace(1, 4, 3);
  m.Faces.AddFace(1, 2, 4);
  m.Faces.AddFace(2, 5, 4);
  m.Faces.AddFace(2, 0, 5);

  // get 2nd layer of pts
  Point3d pt7 = (pt5 + pt6) / 2;
  Point3d pt8 = (pt6 + pt4) / 2;
  Point3d pt9 = (pt4 + pt5) / 2;
  // move 3 new points
  Vector3d move2 = n * dist * angle * depth / 120;
  pt7 += move2;
  pt8 += move2;
  pt9 += move2;
```

```
  m.Vertices.Add(pt7);//6
  m.Vertices.Add(pt8);//7
  m.Vertices.Add(pt9);//8
  m.Faces.AddFace(4, 5, 6);
  m.Faces.AddFace(5, 3, 7);
  m.Faces.AddFace(3, 4, 8);
  m.Faces.AddFace(6, 7, 8);

  // add subtrees
  if (depth > 1){
    Mesh subtree1 = plantTreeFace(pt5, pt7, pt9, dist, targetPt, depth - 1, angle);
    Mesh subtree2 = plantTreeFace(pt6, pt8, pt7, dist, targetPt, depth - 1, angle);
    Mesh subtree3 = plantTreeFace(pt4, pt9, pt8, dist, targetPt, depth - 1, angle);
    m.Append(subtree1);
    m.Append(subtree2);
    m.Append(subtree3);
    m.Vertices.CombineIdentical(false, true);
  }
  m.Normals.ComputeNormals();
  return m;
}
```

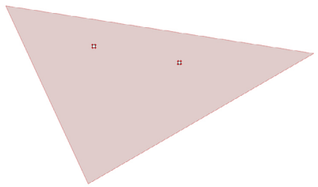
Below are some helper functions that are used in the four steps.

```
// helper function
// returns the normal of plane defined by 3 points
Vector3d normal(Point3d pt1, Point3d pt2, Point3d pt3){
  Vector3d u = pt2 - pt1;
  Vector3d v = pt3 - pt1;
  u.Unitize();
  v.Unitize();
  Vector3d n = Vector3d.CrossProduct(u, v);
  n.Unitize();
  return n;
}

// helper function
// returns the cloest pt in targetPts
Point3d closestTargetPt(List<Point3d> targetPts, Point3d pt){
  double dist = pt.DistanceTo(targetPts[0]);
  Point3d cloestPt = targetPts[0];
  for(int i = 1; i < targetPts.Count; i++){
    if (dist > pt.DistanceTo(targetPts[i])){
      dist = pt.DistanceTo(targetPts[i]);
      cloestPt = targetPts[i];
    }
  }
  return cloestPt;
}

// helper function
// returns the area of the triangle of 3 points
double area(Point3d pt1, Point3d pt2, Point3d pt3)
{
  double a = pt1.DistanceTo(pt2);
  double b = pt2.DistanceTo(pt3);
  double c = pt3.DistanceTo(pt1);
  double p = 0.5 * (a + b + c);
  return Math.Sqrt(p * (p - a) * (p - b) * (p - c));
}
```

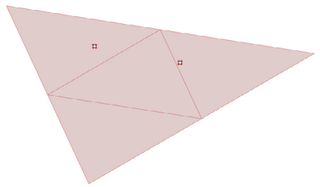
```
// helper function
// returns the list of centers of faces of mesh
// indexed in the same sequence as faces of mesh
List<Point3d> centers(Mesh inputMesh){
  List<Point3d> cens = new List<Point3d>();
  for(int i = 0; i < inputMesh.Faces.Count; i++){
    if (inputMesh.Faces[i].IsQuad){
      Point3d center = (inputMesh.Vertices[inputMesh.Faces[i].A]
        + inputMesh.Vertices[inputMesh.Faces[i].B]
        + inputMesh.Vertices[inputMesh.Faces[i].C]
        + inputMesh.Vertices[inputMesh.Faces[i].D]);
      center /= 4;
      cens.Add(center);
    } else {
      Point3d center = (inputMesh.Vertices[inputMesh.Faces[i].A]
        + inputMesh.Vertices[inputMesh.Faces[i].B]
        + inputMesh.Vertices[inputMesh.Faces[i].C]);
      center /= 3;
      cens.Add(center);
    }
  }
  return cens;
}
```

```
// helper function
// returns the index of the cloest pt in targetPts
int closestTargetPtIndex(List<Point3d> targetPts, Point3d pt){
  double dist = pt.DistanceTo(targetPts[0]);
  int index = 0;
  for(int i = 1; i < targetPts.Count; i++){
    if (dist > pt.DistanceTo(targetPts[i])){
      dist = pt.DistanceTo(targetPts[i]);
      index = i;
    }
  }
  return index;
}
```

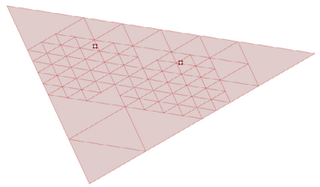
TRIANGLE
01 Subdivide + Move



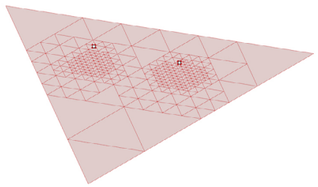
Times: 0
Radius: 264



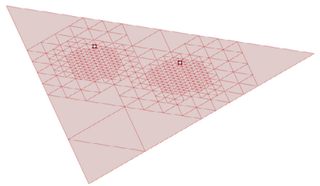
Times: 1
Radius: 264



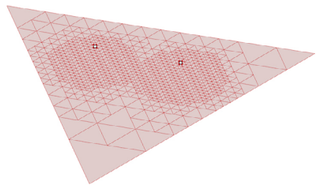
Times: 4
Radius: 264



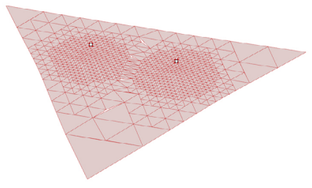
Times: 6
Radius: 264



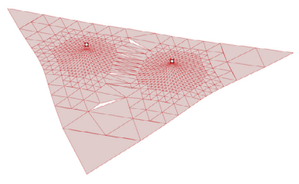
Times: 6
Radius: 300



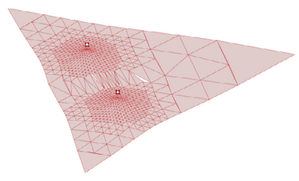
Times: 6
Radius: 450



Times: 6
Radius: 450
Move Strength: 80

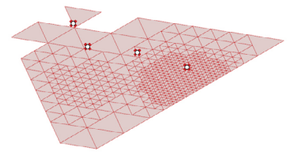


Times: 6
Radius: 450
Move Strength: 220

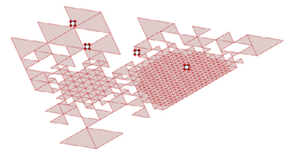


Times: 6
Radius: 450
Move Strength: 220
Target Points Moved

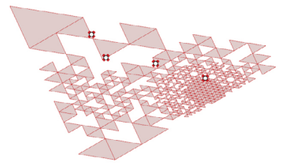
TRIANGLE
02 Subdivide + Cull



Times: 6
Radius: 450
Cull Range: 150

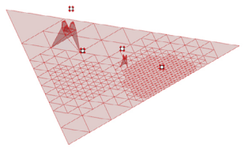


Times: 6
Radius: 450
Cull Range: 90

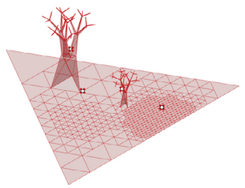


Times: 6
Radius: 450
Cull Range: 40

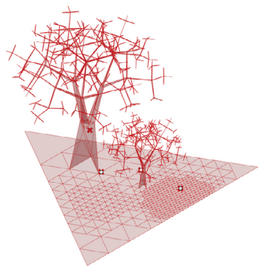
TRIANGLE
03 Subdivide + Tree



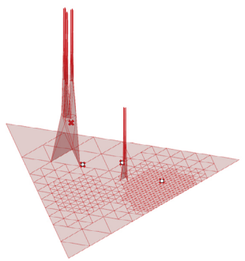
Times: 6
Radius: 450
Tree Depth: 2
Tree Angle: 0.877



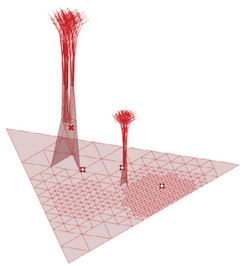
Times: 6
Radius: 450
Tree Depth: 4
Tree Angle: 0.877



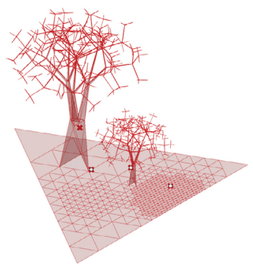
Times: 6
Radius: 450
Tree Depth: 6
Tree Angle: 0.877



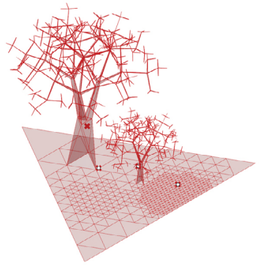
Times: 6
Radius: 450
Tree Depth: 6
Tree Angle: 0.000



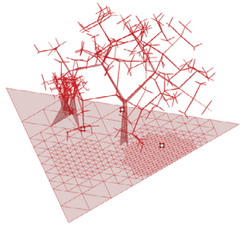
Times: 6
Radius: 450
Tree Depth: 6
Tree Angle: 0.024



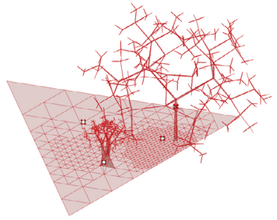
Times: 6
Radius: 450
Tree Depth: 6
Tree Angle: 0.526



Times: 6
Radius: 450
Tree Depth: 6
Tree Angle: 0.783
Tree Points Moved

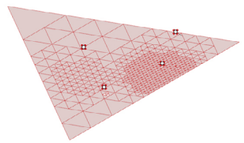


Times: 6
Radius: 450
Tree Depth: 6
Tree Angle: 0.783
Tree Points Moved

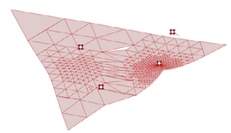


Times: 6
Radius: 450
Tree Depth: 6
Tree Angle: 0.783
Tree Points Moved

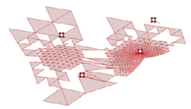
TRIANGLE
Combinations



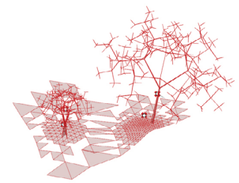
01 Subdivide
Times: 6
Radius: 470



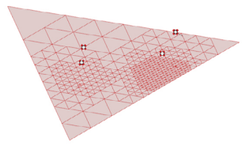
02 Move
Move Strength: 260



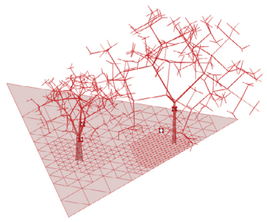
03 Cull
Cull Range: 69



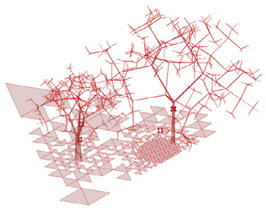
04 Tree
Tree Depth: 6
Tree Angle: 0.394



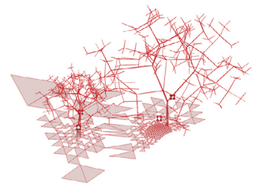
01 Subdivide
Times: 6
Radius: 470



02 Tree
Tree Depth: 6
Tree Angle: 0.394

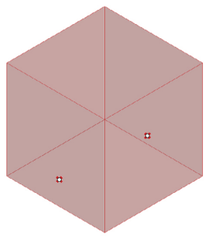


03 Cull
Cull Range: 69

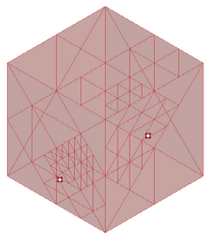


04 Move
Move Strength: 260

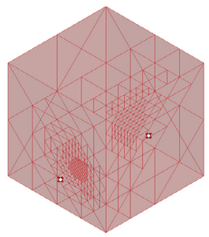
CUBE
01 Subdivide + Move



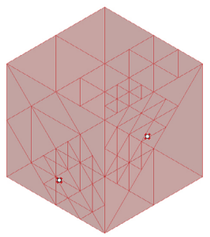
Times: 0
Radius: 450



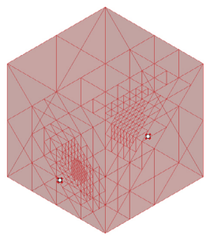
Times: 1
Radius: 450



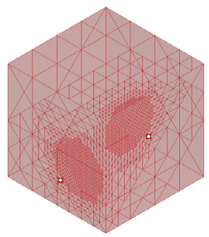
Times: 4
Radius: 450



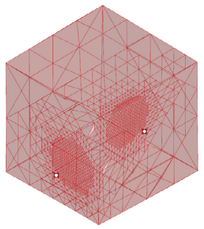
Times: 6
Radius: 400



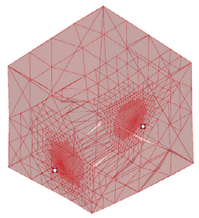
Times: 6
Radius: 600



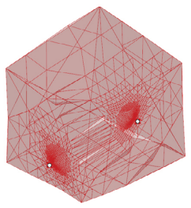
Times: 6
Radius: 900



Times: 6
Radius: 900
Move Strength: 180

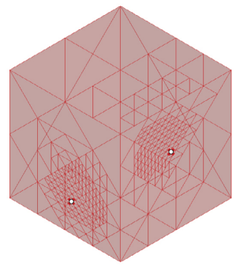


Times: 6
Radius: 900
Move Strength: 500

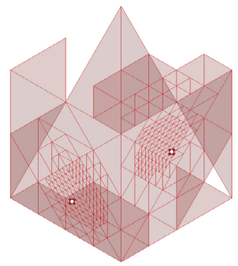


Times: 6
Radius: 900
Move Strength: 800

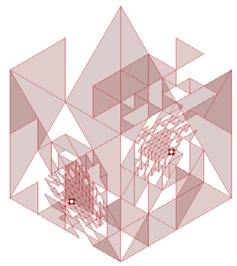
CUBE
02 Subdivide + Cull



Times: 5
Radius: 500
Cull Range: 800

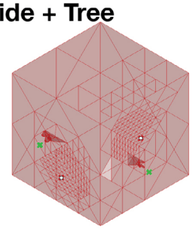


Times: 5
Radius: 500
Cull Range: 260

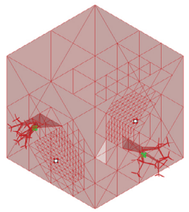


Times: 5
Radius: 500
Cull Range: 70

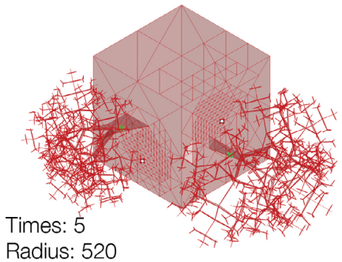
CUBE
03 Subdivide + Tree



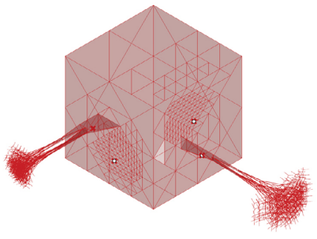
Times: 5
Radius: 520
Tree Depth: 2
Tree Angle: 1.556



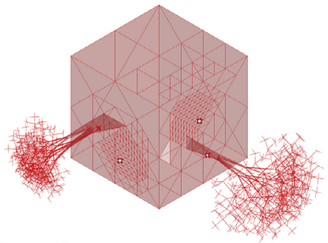
Times: 5
Radius: 520
Tree Depth: 4
Tree Angle: 1.556



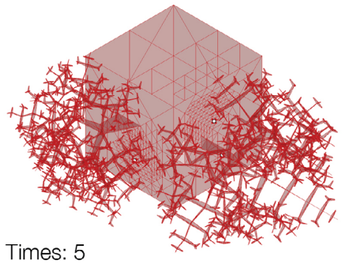
Times: 5
Radius: 520
Tree Depth: 7
Tree Angle: 1.556



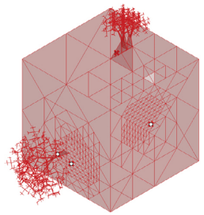
Times: 5
Radius: 520
Tree Depth: 7
Tree Angle: 0.029



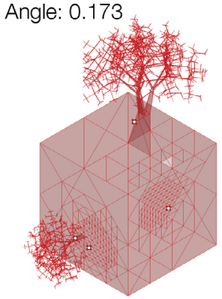
Times: 5
Radius: 520
Tree Depth: 7
Tree Angle: 0.173



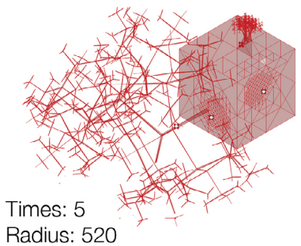
Times: 5
Radius: 520
Tree Depth: 7
Tree Angle: 4.235



Times: 5
Radius: 520
Tree Depth: 7
Tree Angle: 1.556
Tree Points Moved

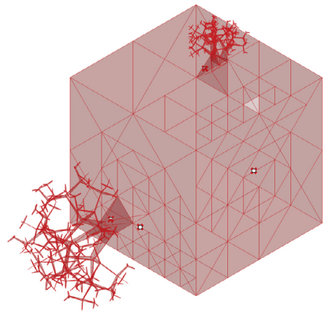


Times: 5
Radius: 520
Tree Depth: 7
Tree Angle: 1.556
Tree Points Moved

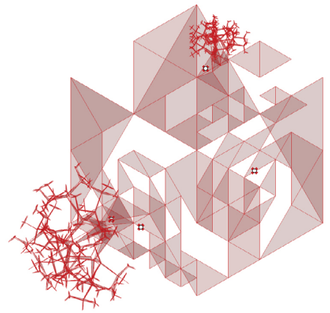


Times: 5
Radius: 520
Tree Depth: 7
Tree Angle: 1.556
Tree Points Moved

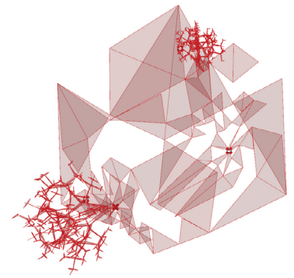
CUBE
Combinations



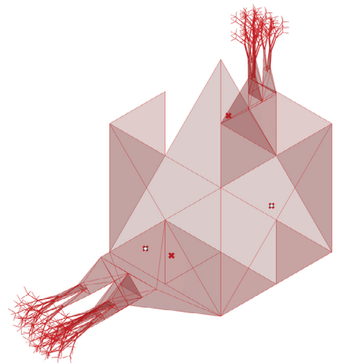
01 Subdivide + Tree
Times: 3
Radius: 550
Tree Depth: 6
Tree Angle: 2.405



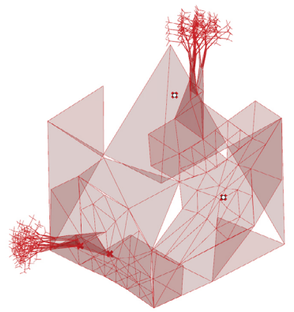
02 Cull
Cull Range: 0



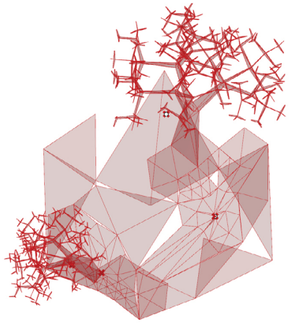
03 Move
Move Strength: 800



01 Subdivide + Cull + Tree
Times: 1
Radius: 550
Cull Range: 280
Tree Depth: 6
Tree Angle: 0.201

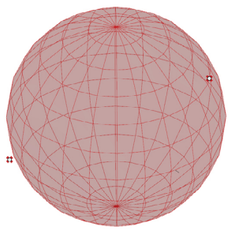


02 Move
Move Strength: 530

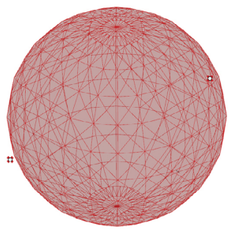


03 Adjust Tree Angle
Tree Angle: 2.405

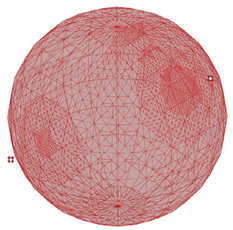
SPHERE
01 Subdivide + Move



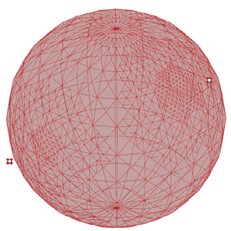
Times: 0
Radius: 500



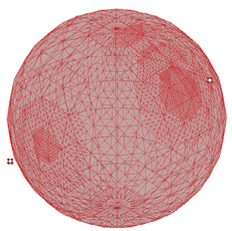
Times: 1
Radius: 500



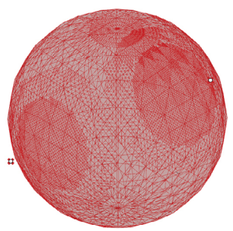
Times: 4
Radius: 500



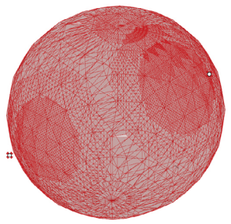
Times: 4
Radius: 380



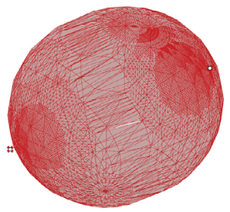
Times: 4
Radius: 510



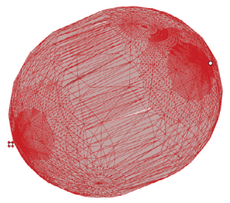
Times: 4
Radius: 710



Times: 4
Radius: 710
Move Strength: 157

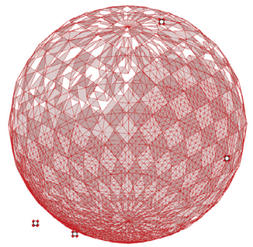


Times: 4
Radius: 710
Move Strength: 512

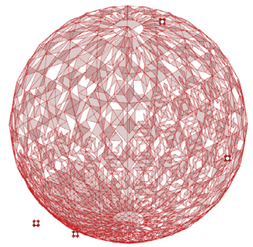


Times: 4
Radius: 710
Move Strength: 800

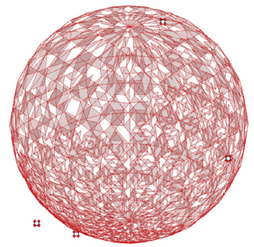
SPHERE
02 Subdivide + Cull



Times: 3
Radius: 620
Cull Range: 260

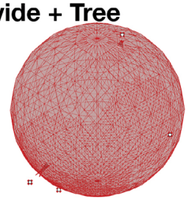


Times: 3
Radius: 620
Cull Range: 130

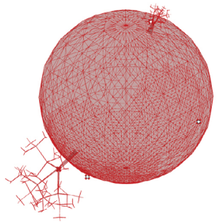


Times: 3
Radius: 620
Cull Range: 0

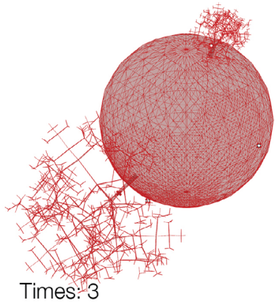
SPHERE
03 Subdivide + Tree



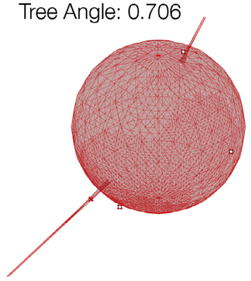
Times: 3
Radius: 620
Tree Depth: 2
Tree Angle: 0.706



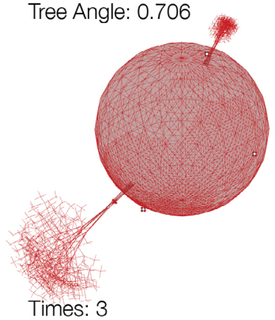
Times: 3
Radius: 620
Tree Depth: 5
Tree Angle: 0.706



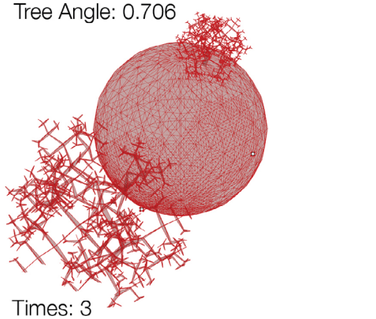
Times: 3
Radius: 620
Tree Depth: 7
Tree Angle: 0.706



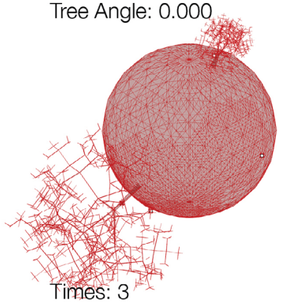
Times: 3
Radius: 620
Tree Depth: 7
Tree Angle: 0.000



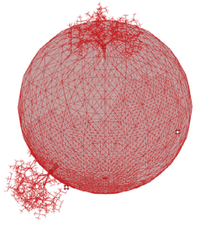
Times: 3
Radius: 620
Tree Depth: 7
Tree Angle: 0.043



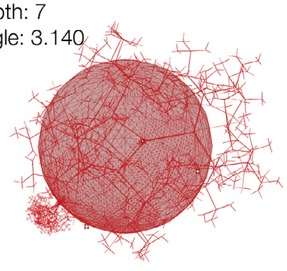
Times: 3
Radius: 620
Tree Depth: 7
Tree Angle: 3.140



Times: 3
Radius: 620
Tree Depth: 7
Tree Angle: 0.547
Tree Points Moved

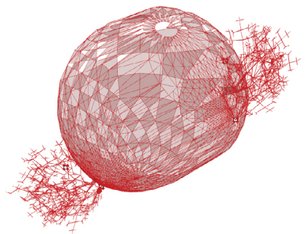


Times: 3
Radius: 620
Tree Depth: 7
Tree Angle: 0.547
Tree Points Moved

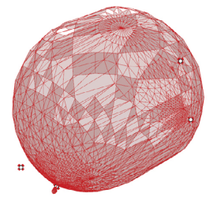


Times: 3
Radius: 620
Tree Depth: 7
Tree Angle: 0.547
Tree Points Moved

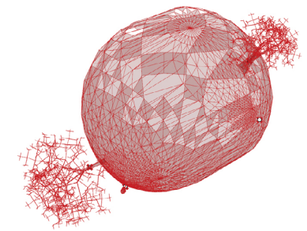
SPHERE
Combinations



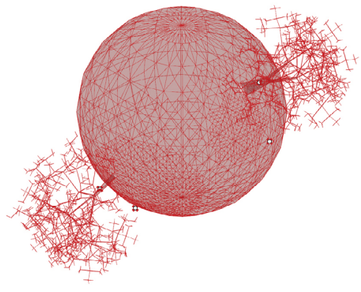
01 Subdivide + Move
Times: 3
Radius: 470
Move Strength: 800



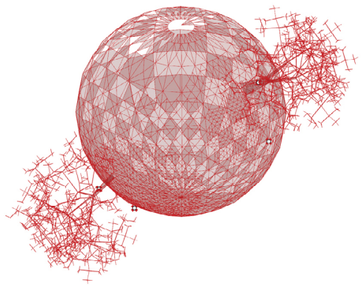
02 Cull
Cull Range: 350



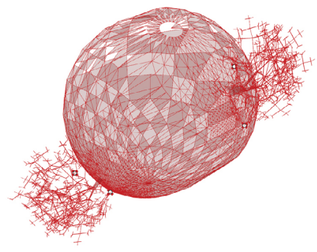
03 Tree
Tree Depth: 7
Tree Angle: 0.547



01 Subdivide + Tree
Times: 3
Radius: 470
Tree Depth:7
Tree Angle: 0.547



02 Cull
Cull Range: 350



03 Move
Move Strength: 623