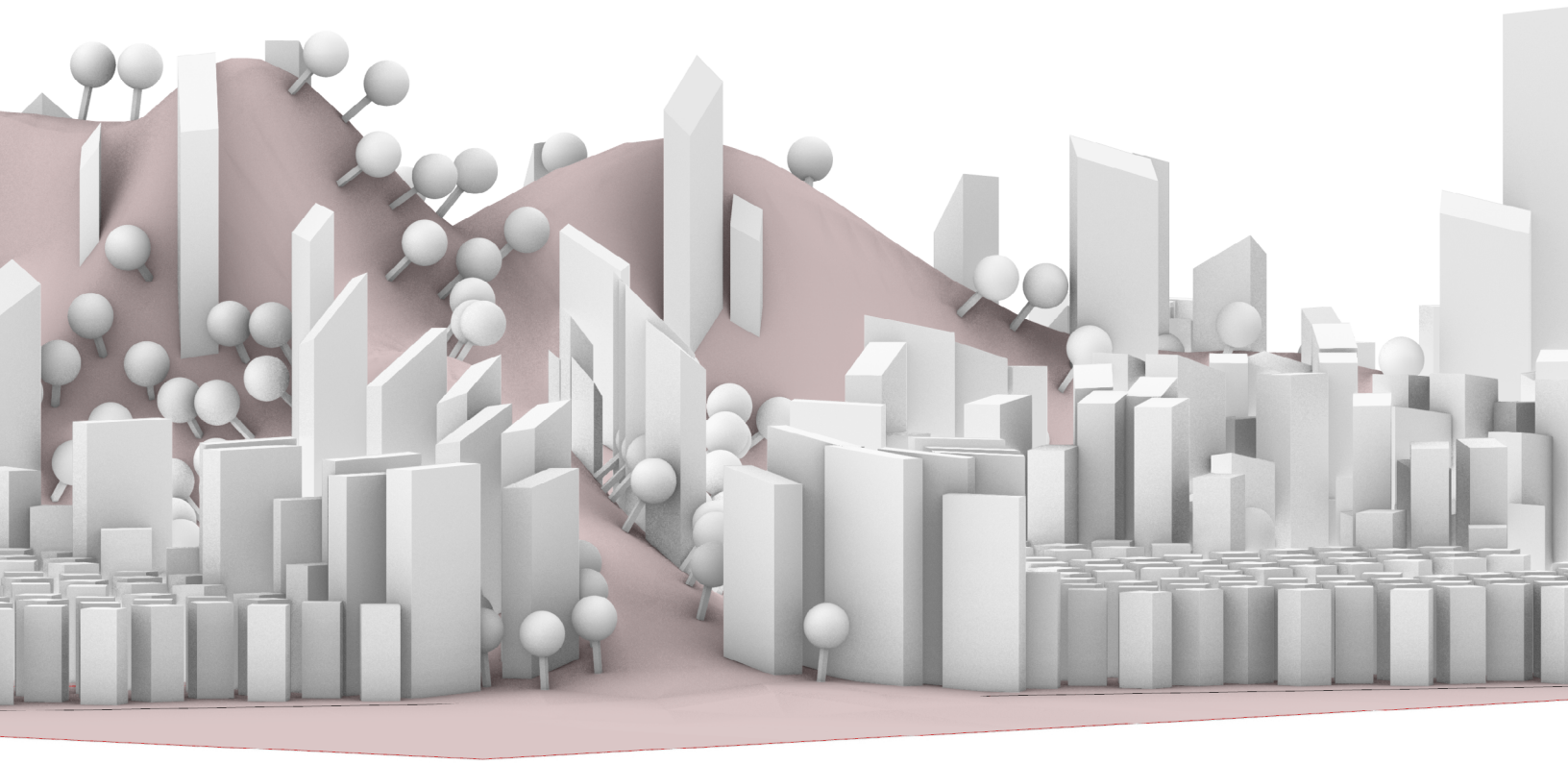
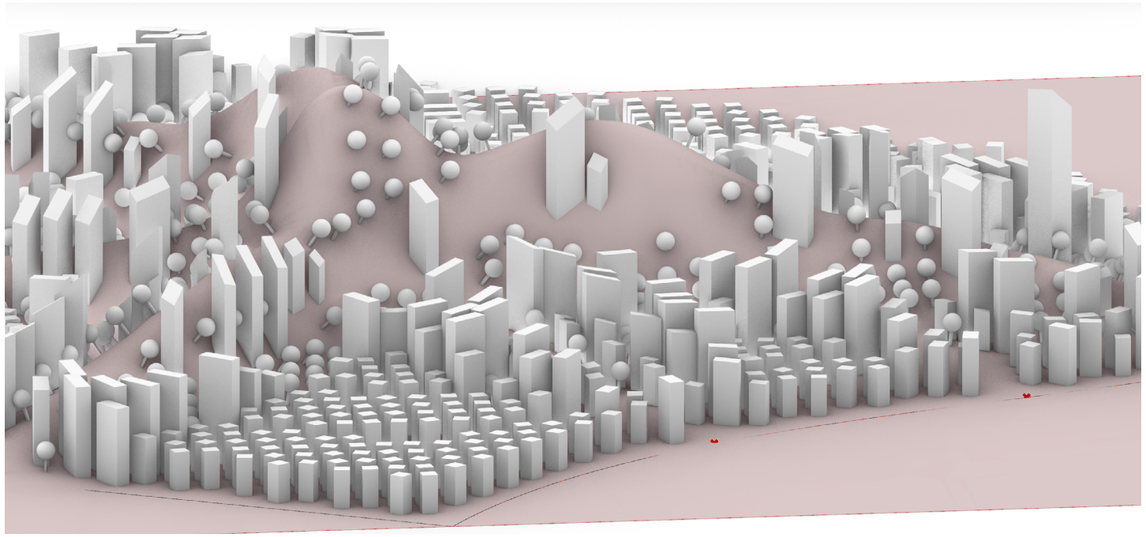
The Mountain City

Final Project

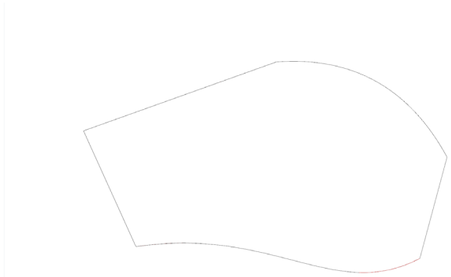
Yuzhen Zhang



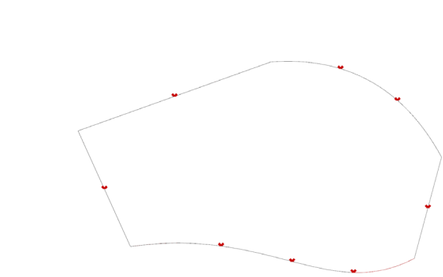


The Mountain City is situated around a mountain. Its city center is a green mountain park serving people living around it. The circular roads around the mountain and connection paths on the ridges provides easy circulation around the city. Residential buildings, office towers, and gardens are placed based on the topography of the ground. The city is generated by the C# script with the following 6 steps.

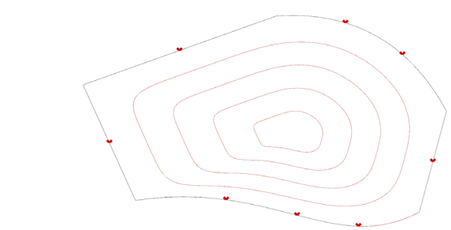
```
/// </summary>
public class Script_Instance : GH_ScriptInstance
{
Utility functions

Members

  /**/
  private void RunScript(Curve curve, ref object EntryPoints, ref object CircularPaths, ref object Entr
  {
    // 01 split the curve
    Tuple<List<Line>, List<Curve>> linecrvs = splitCurve(curve);

    // 02 get entry points
    List<Point3d> entryPts = entryPoints(linecrvs);

    // 03 create paths
    Tuple<List<Curve>, List<Curve>> paths = createPaths(entryPts, curve);

    // 04 intersection
    List<Point3d> intersectionPts = intersectCurves(paths.Item1, paths.Item2);

    // 05 topography
    Surface topo = createTopography(curve, paths.Item2);

    // 06 adaptive distribution
    List<Curve> allPaths = new List<Curve>();
    allPaths.AddRange(paths.Item1);
    allPaths.AddRange(paths.Item2);
    List<Surface> elements = adaptiveDistribution(topo, allPaths, curve);

    EntryPoints = entryPts;
    CircularPaths = paths.Item1;
    EntryPaths = paths.Item2;
    IntersectionPoints = intersectionPts;
    Topography = topo;
    Distribution = elements;
  }

  // <Custom additional code>
```



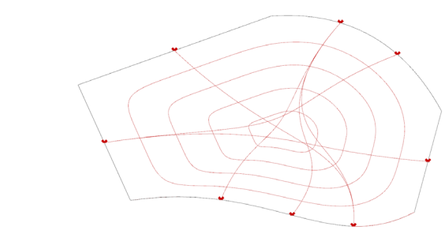
Boundary



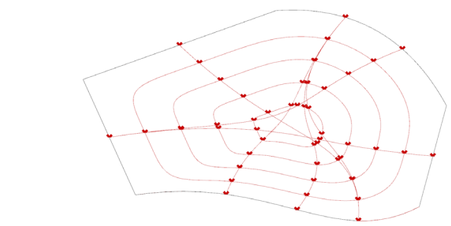
Entry Points



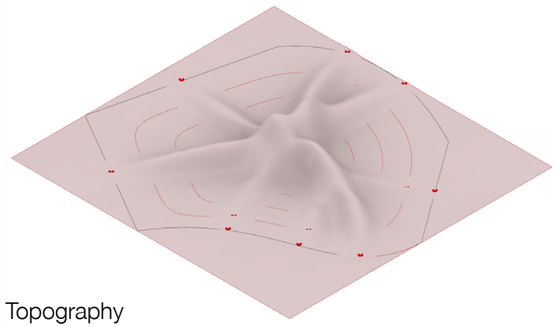
Circular Paths



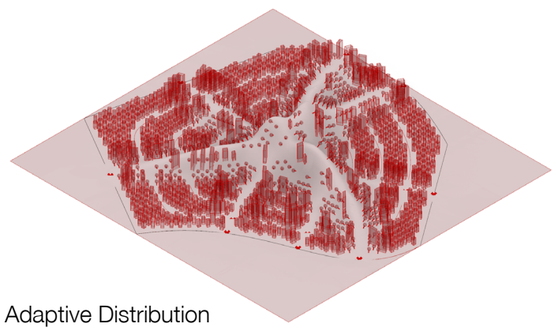
Entry Paths



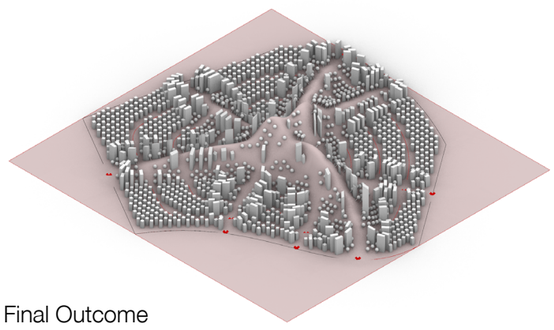
Intersection Nodes



Topography



Adaptive Distribution



Final Outcome

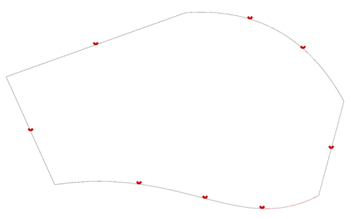
01. Boundary Splitting

The boundary curve is split into shorter segments based on the segments' properties. Line segments and curve segments are separated for next step.

```
/* splitCurve returns a tuple of the line segments and curve segments
   of the input curve
*/
private Tuple<List<Line>, List<Curve>> splitCurve(Curve curve){
  PolyCurve pc = curve as PolyCurve;

  List<Line> lines = new List<Line>();
  List<Curve> crvs = new List<Curve>();
  Curve[] segments = pc.Explode();

  foreach(Curve ci in segments){
    if (ci.IsLinear()){
      lines.Add(new Line(ci.PointAtStart, ci.PointAtEnd));
    } else if (ci is PolylineCurve){
      PolylineCurve pl = ci as PolylineCurve;
      for(int i = 0; i < pl.PointCount - 1; i++){
        lines.Add(new Line(pl.Point(i), pl.Point(i + 1)));
      }
    } else {
      crvs.Add(ci);
    }
  }

  return new Tuple<List<Line>, List<Curve>> (lines, crvs);
}
```

02. Entry Points

Entry points are placed along the curve segments. For each straight line, one entry point is placed at the midpoint of it. For each curve segment, the curve is equally subdivided into segments longer than 10.0, and the division points become entry points. This process evenly distributes entry points along the boundary and keeps them away from the corners.



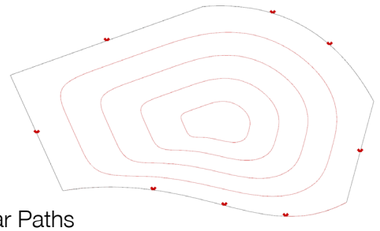
```
/* entryPoints returns a list of entry points along the lines and crvs
*/
private List<Point3d> entryPoints(Tuple<List<Line>, List<Curve>> linecrvs){
  List<Line> lines = linecrvs.Item1;
  List<Curve> crvs = linecrvs.Item2;
  List<Point3d> pts = new List<Point3d>();

  // get midpoints of line segments
  foreach(Line li in lines){
    pts.Add((li.From + li.To) / 2);
  }

  // place points along curves according to their lengths
  foreach(Curve ci in crvs){
    double len = ci.GetLength();
    int num = (int) (len / 10.0);

    double t0 = ci.Domain.Min;
    double t1 = ci.Domain.Max;

    for(int i = 0; i < num - 1; i++){
      pts.Add(ci.PointAt((t1 - t0) * (i + 1) / num + t0));
    }
  }

  return pts;
}
```

03. Create Paths

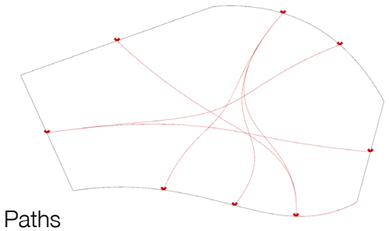
Two paths systems are generated for circulation in this city.

The first set of paths is a set of circular paths parallel to the boundary. 20 sample points are generated along the boundary to be moved towards their center and used for generating circular paths. A total of 4 loops are created and they are equally spanned.

Then, the entry points are connected with a series of entry paths. They are curves that connect each entry point to the entry that is farthest away from it. The curves are perpendicular to the boundary at where they meet the boundary, and they also use the center point as a control point, so they all get close to the center of the city.



Circular Paths



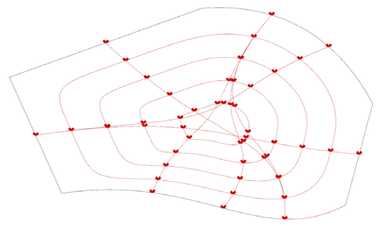
Entry Paths

```
/* createPaths returns a list of generated paths
*/
private Tuple<List<Curve>, List<Curve>> createPaths(List<Point3d> pts, Curve curve){
  Point3d center = average(pts);
  // create circular paths
  List<Curve> cPaths = createCircularPaths(samplePts(curve), center);
  // connect entry pts
  List<Curve> ePaths = connectEntryPaths(pts, curve, center);

  return new Tuple<List<Curve>, List<Curve>> (cPaths, ePaths);
}

/* average returns the average of pts
*/
private Point3d average(List<Point3d> pts){
  Point3d sum = pts[0];
  for(int i = 1; i < pts.Count; i++){
    sum += pts[i];
  }
  return sum / pts.Count;
}

/* samplePts returns 20 pts along the curve
*/
private List<Point3d> samplePts(Curve curve){
  int n = 20;
  double t0 = curve.Domain.Min;
  double t1 = curve.Domain.Max;
  double dt = (t1 - t0) / (double) (n - 1.0);
  List<Point3d> pts = new List<Point3d>();
  for(int i = 0; i < n; i++){
    pts.Add(curve.PointAt(t0 + i * dt));
  }
  return pts;
}

/* createCircularPaths returns a list of circular paths around center
*/
private List<Curve> createCircularPaths(List<Point3d> pts, Point3d center){
  List<Curve> cPaths = new List<Curve>();
  int n = 5;
  // creates 4 circular paths
  for(int i = 1; i < n; i++){
    List<Point3d> ptl = new List<Point3d>();
    foreach(Point3d pi in pts){
      ptl.Add((center - pi) * i / n + pi);
    }
    cPaths.Add(NurbsCurve.CreateControlPointCurve(ptl, 3));
  }
  return cPaths;
}
```

```
/* connectEntryPaths returns a list of paths that
   connects the entry points.
   Paths are perpendicular to the boundary curve
   and are affected by the center point too.
*/
private List<Curve> connectEntryPaths(List<Point3d> pts, Curve c, Point3d center){
  List<Curve> connection = new List<Curve>();
  // connect to the farthest point
  for(int i = 0; i < pts.Count; i++){
    double maxD = 0.0;
    Point3d maxPt = pts[i];
    for(int j = 0; j < pts.Count; j++){
      double d = pts[i].DistanceTo(pts[j]);
      if (maxD < d){
        maxD = d;
        maxPt = pts[j];
      }
    }
    // find perpendicular direction
    double t0 = 0.0;
    double t1 = 0.0;
    c.ClosestPoint(pts[i], out t0);
    c.ClosestPoint(maxPt, out t1);

    Vector3d pv0 = Vector3d.
      CrossProduct(c.TangentAt(t0), Vector3d.ZAxis);
    Vector3d pv1 = Vector3d.
      CrossProduct(c.TangentAt(t1), Vector3d.ZAxis);

    // Create Curve
    List<Point3d> cpt = new List<Point3d>();
    cpt.Add(pts[i]);
    cpt.Add(pts[i] + pv0 * maxD / 4);
    cpt.Add(center);
    cpt.Add(maxPt + pv1 * maxD / 4);
    cpt.Add(maxPt);
    connection.Add(NurbsCurve.CreateControlPointCurve(cpt, 3));
  }
  return connection;
}
```

04. Intersection Nodes

Intersection nodes are found at the intersection points between the circular paths and the entry paths.



```
/* intersectCurves returns the intersection points between
   the circular curves and the entry connection curves
*/
private List<Point3d> intersectCurves(List<Curve> cPaths, List<Curve> ePaths){
  List<Point3d> xpts = new List<Point3d>();

  foreach(Curve ci in cPaths){
    foreach(Curve ei in ePaths){
      xpts.AddRange(intersectTwoCurves(ci, ei));
    }
  }

  return xpts;
}

/* intersectTwoCurves returns the intersection points between two curves
*/
private List<Point3d> intersectTwoCurves(Curve c1, Curve c2){
  List<Point3d> xpts = new List<Point3d>();

  Rhino.Geometry.Intersect.CurveIntersections xe =
    Rhino.Geometry.Intersect.Intersection.CurveCurve(c1, c2, 1.0, 1.0);
  foreach(Rhino.Geometry.Intersect.IntersectionEvent x in xe){
    if (x.IsPoint){
      xpts.Add(x.PointA);
    }
  }

  return xpts;
}
```

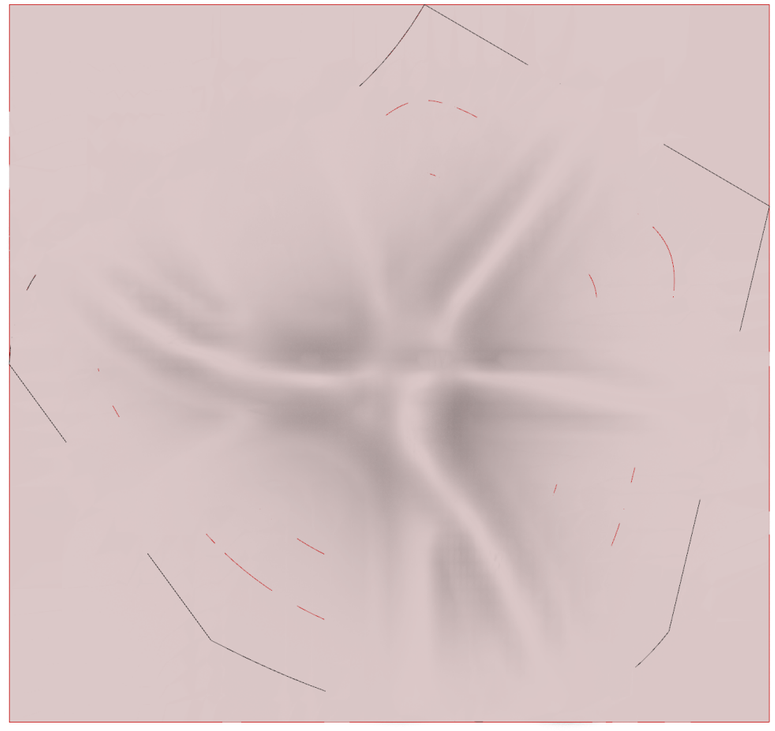
05. Topography

The topography is created by a matrix of control points. The Z-value of the points are determined by the entry paths and the boundary curve. It equals the sum of the impact of each entry paths times the distance from the point to the boundary. The impact of one path is equal to the equation below:

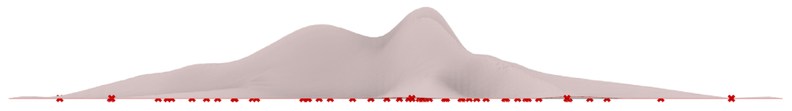
```
310         double deltaZ = db / 14 * Math.Exp(-0.2 * dist * dist);
```

Where db is the distance from the point to the boundary, and dist is the distance to the path.

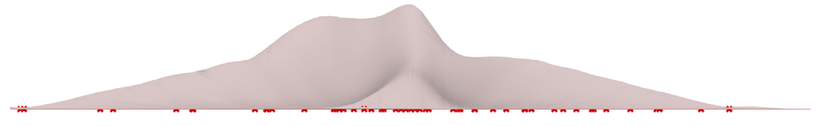
The result topography places ridges along the entry paths, and the ground become higher the closer to the center, as the distance to the boundary become larger and all entry paths pass through the center area. Therefore, a mountain is created at the city center.



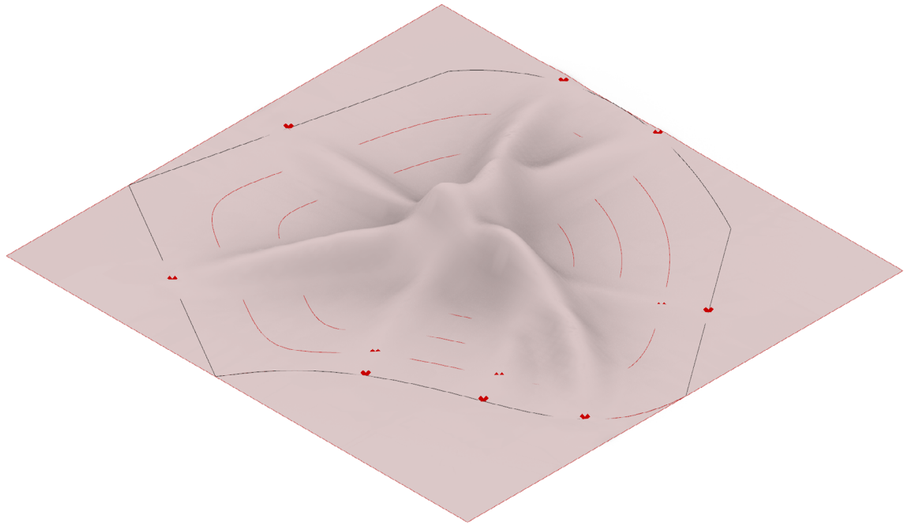
Plan



Elevation 1



Elevation 2

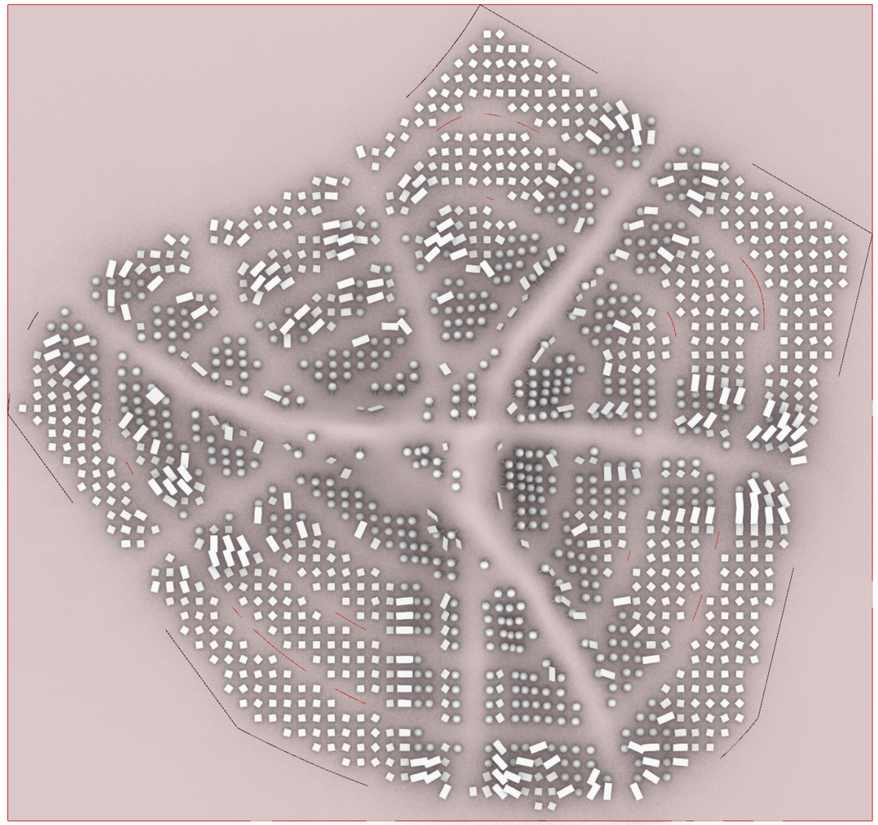


```
264
265   /* createTopography first creates topo points, then adjust their z values
266      and returns the surface through them
267   */
268   private Surface createTopography(Curve curve, List<Curve> activeCurves){
269     BoundingBox bb = curve.GetBoundingBox(true);
270     Point3d min = bb.Min;
271     Point3d max = bb.Max;
272     int nx = (int) ((max.X - min.X) / 2.0);
273     int ny = (int) ((max.Y - min.Y) / 2.0);
274
275     List<Point3d> topoPts = createTopoPoints(min, max, nx, ny);
276     topoPts = adjustTopoPoints(topoPts, activeCurves, curve);
277
278     return NurbsSurface.CreateThroughPoints(topoPts, nx, ny, 3, 3, false, false);
279   }
280
281   /* createTopoPoints creates matrix of points for topography
282   */
283   private List<Point3d> createTopoPoints(Point3d min, Point3d max, int nx, int ny){
284     List<Point3d> pts = new List<Point3d>();
285     double dx = (max.X - min.X) / (double) (nx - 1.0);
286     double dy = (max.Y - min.Y) / (double) (ny - 1.0);
287
288     for(int i = 0; i < nx; i++){
289       for(int j = 0; j < ny; j++){
290         pts.Add(new Point3d((min.X + dx * i), (min.Y + dy * j), 0.0));
291       }
292     }
293     return pts;
294   }
295
296   /* adjustTopoPoints adjust the z value of points in pts
297      according to the impact of the curves and the boundary
298   */
299   private List<Point3d> adjustTopoPoints(List<Point3d> pts, List<Curve> curves, Curve boundC){
300     List<Point3d> newPts = new List<Point3d>();
301     for(int i = 0; i < pts.Count; i++){
302       Point3d newpt = pts[i];
303       double tb = 0.0;
304       boundC.ClosestPoint(pts[i], out tb);
305       double db = pts[i].DistanceTo(boundC.PointAt(tb));
306       foreach(Curve c in curves){
307         double t = 0.0;
308         c.ClosestPoint(pts[i], out t);
309         double dist = pts[i].DistanceTo(c.PointAt(t));
310         double deltaZ = db / 14 * Math.Exp(-0.2 * dist * dist);
311         newpt.Z += deltaZ;
312       }
313       newPts.Add(newpt);
314     }
315     return newPts;
316   }
317
```

06. Adaptive Distribution

The final step generates buildings and trees in the city based on the SurfaceCurvature of the city topography. The script first creates a matrix of points in the city area and get the SurfaceCurvature at that point. Then it removes the points that are to close to the paths created in the previous steps as these places are not ideal for construction.

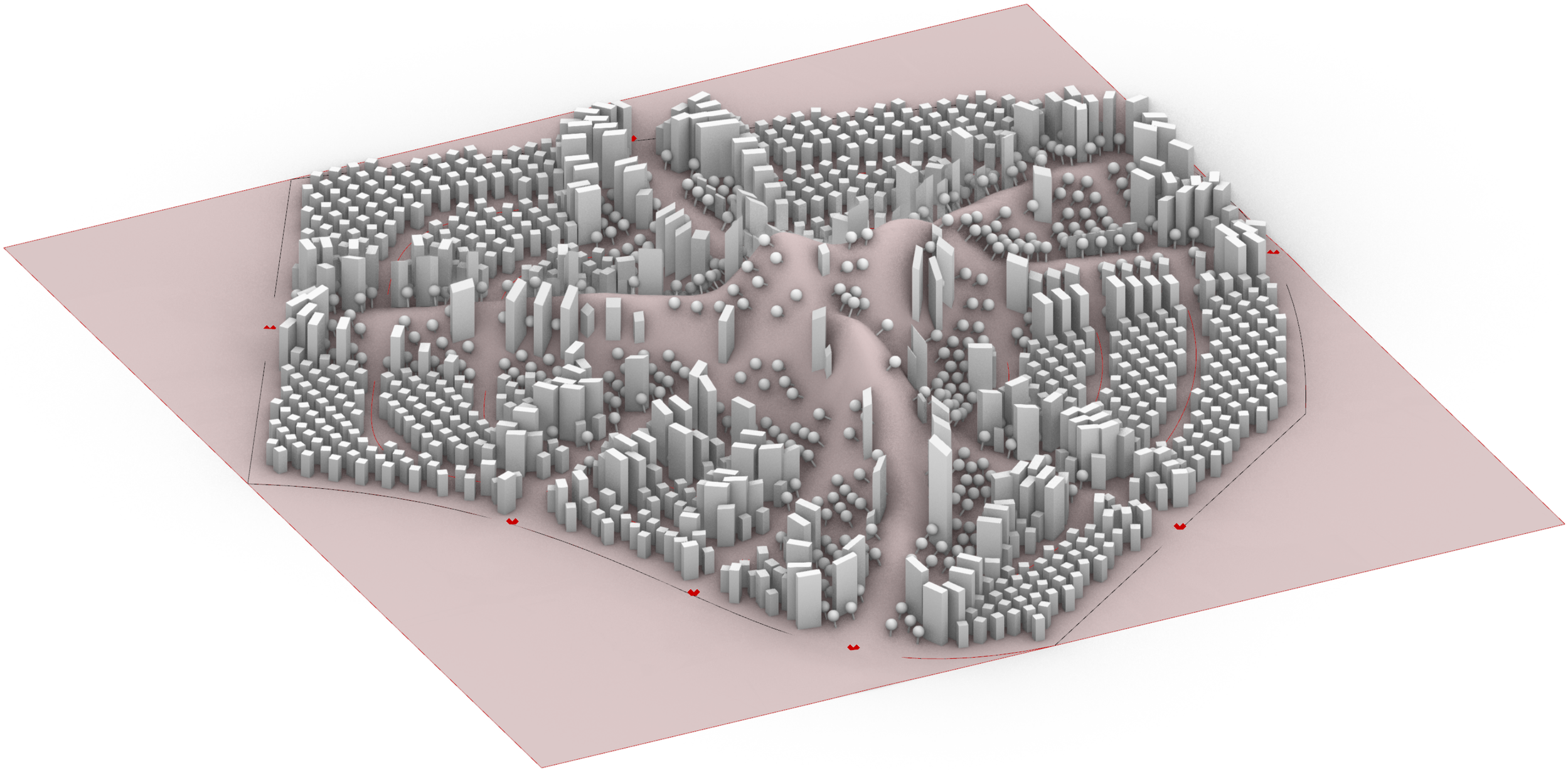
For each point, if the absolute value of its Kappa is between -0.1 to 0.1, the slope is not changing drastically so an architecture can be constructed. Otherwise, a tree is planted.



Plan

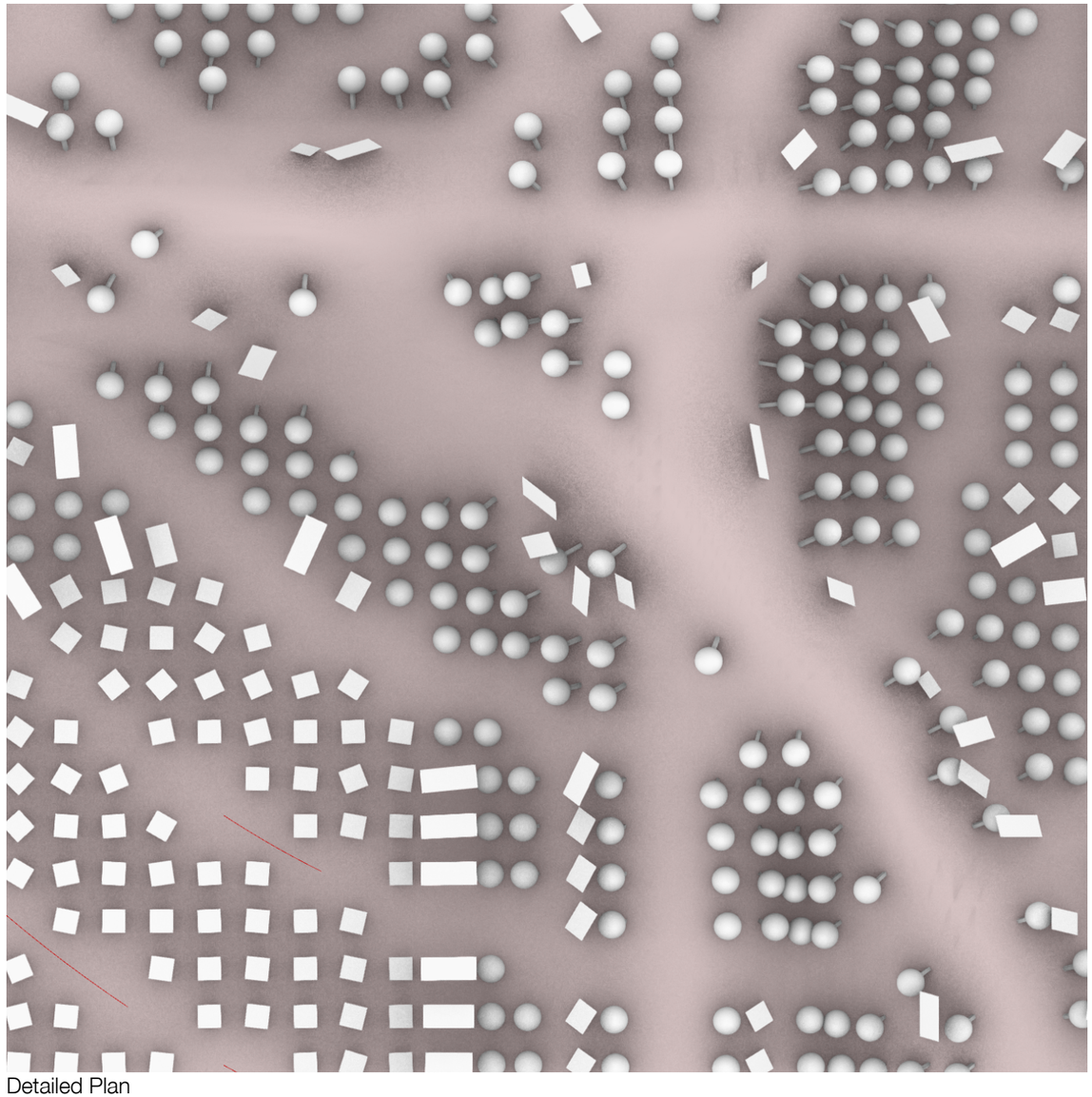
Elevation 1

Elevation 2

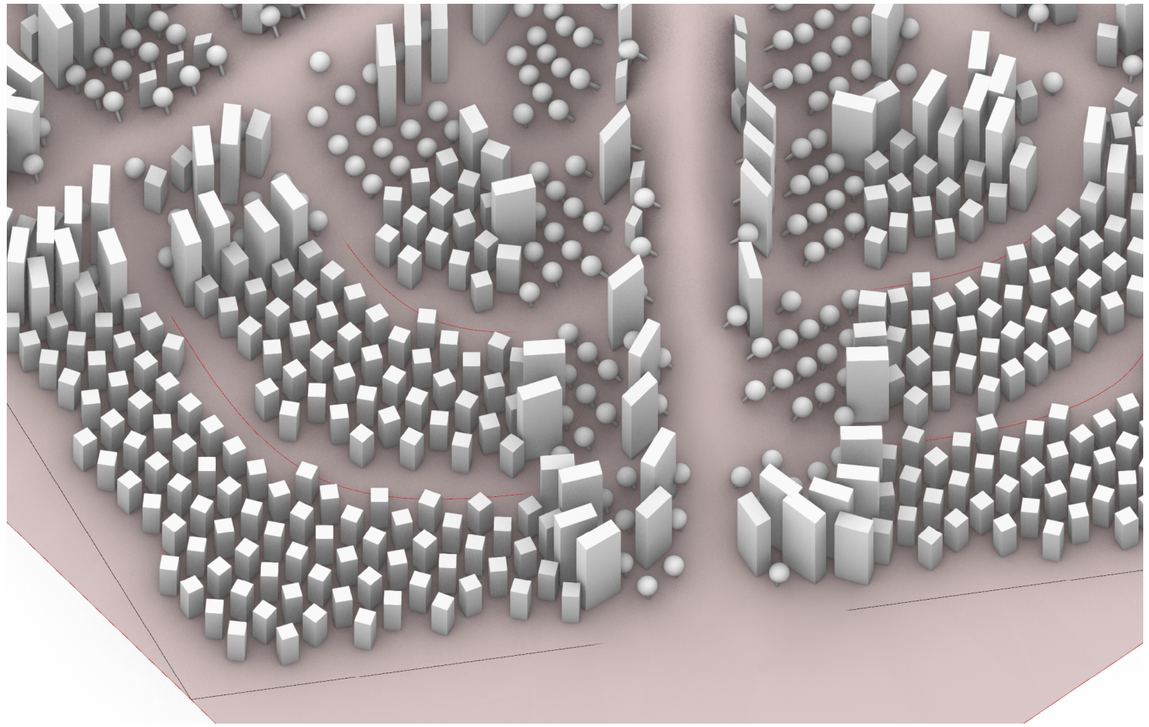


In the case of architecture, its orientation is based on the direction of the surface. The width and length of the footprint is proportional to the Kappa values, and is clamped between 0.25 and 0.6 to avoid unreasonable sizes. The building is extruded vertically parallel to the Z-axis with the height proportional to the area of its footprint. The roof of the building is also parallel to the ground plane. As a result, shorter and more regular units are placed at flat locations and can be used for residential. Taller towers are placed closer to the ridges of the connection paths, as they serve commercial and office uses and should be accessed from these highways easily.

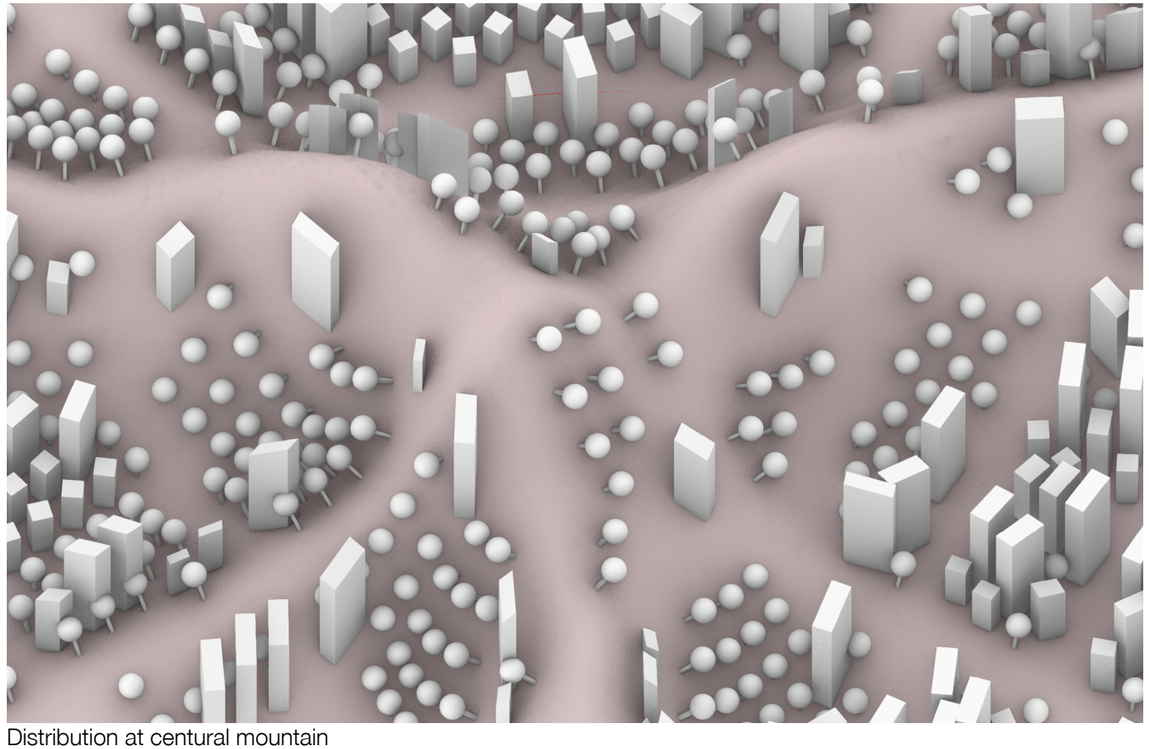
As for the trees, they are grown perpendicular to the ground. They can be found at locations less optimal for buildings. Because of the way the topography is created, they are concentrated to the commercial areas, which increases the land value of the commercial district along the entry paths.



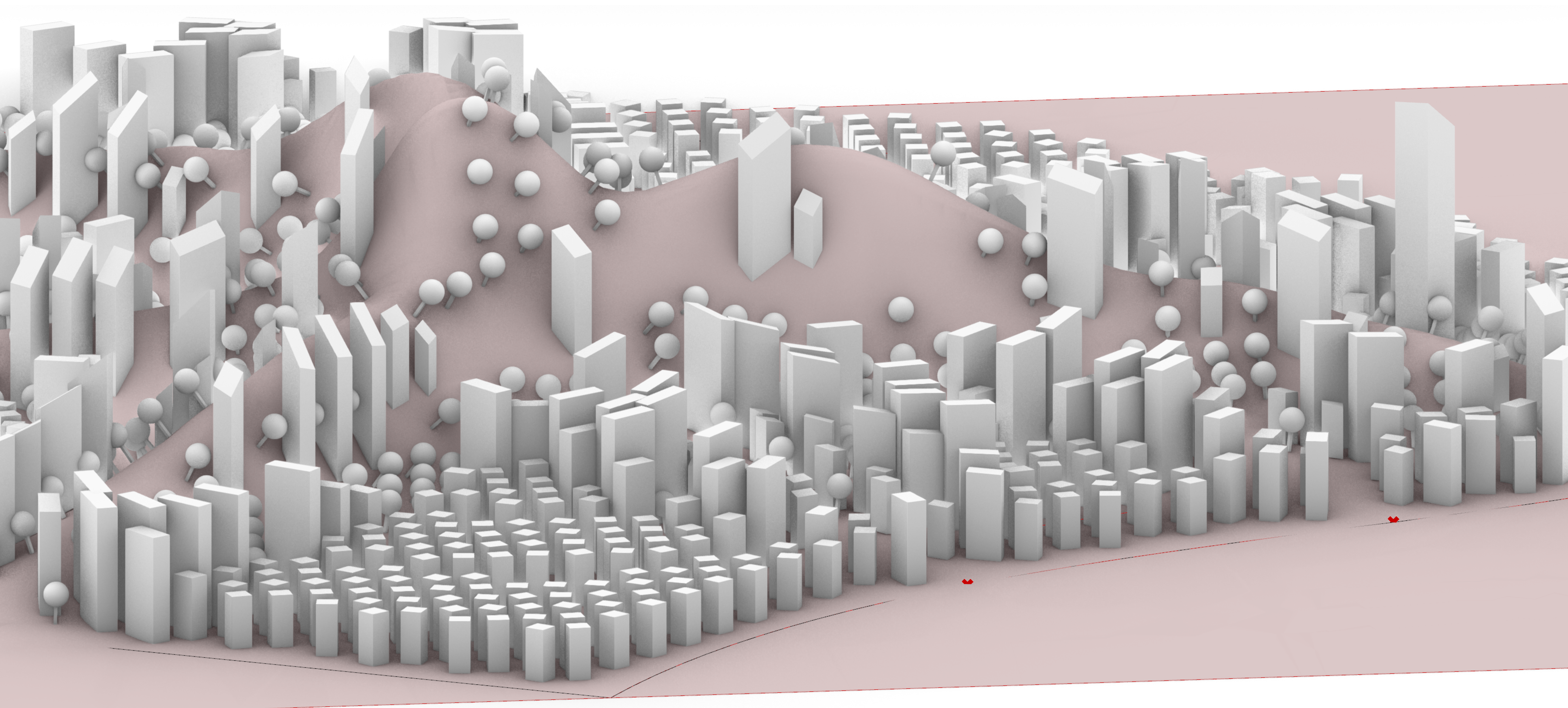
Detailed Plan



Distribution along entry paths



Distribution at centural mountain



```
/* adaptiveDistribution generates trees and buildings on topo
*/
private List<Surface> adaptiveDistribution(Surface topo,
List<Curve> curves, Curve boundC){
  // get curvatures
  List<Tuple<Point3d, SurfaceCurvature>> ptCurv = getCurvatures(topo, boundC);
  // spawn elements
  return spawnElements(ptCurv, curves);
}

/* getCurvatures returns a matrix of points and their
   corresponding SurfaceCurvature
*/
private List<Tuple<Point3d, SurfaceCurvature>> getCurvatures(
Surface topo, Curve boundC){
  List<Tuple<Point3d, SurfaceCurvature>> ptCurv =
    new List<Tuple<Point3d, SurfaceCurvature>>();
  double u0 = topo.Domain(0).Min;
  double u1 = topo.Domain(0).Max;
  double v0 = topo.Domain(1).Min;
  double v1 = topo.Domain(1).Max;
  int uCount = (int) ((u1 - u0) / 1.0);
  int vCount = (int) ((v1 - v0) / 1.0);
  double du = (u1 - u0) / (uCount - 1.0);
  double dv = (v1 - v0) / (vCount - 1.0);

  for(int i = 0; i < uCount; i++){
    for(int j = 0; j < vCount; j++){
      double u = u0 + i * du;
      double v = v0 + j * dv;
      Point3d p = topo.PointAt(u, v);
      if(boundC.Contains(p, Plane.WorldXY, 1.0) == PointContainment.Inside){
        SurfaceCurvature curv = topo.CurvatureAt(u, v);
        ptCurv.Add(new Tuple<Point3d, SurfaceCurvature>(p, curv));
      }
    }
  }

  return ptCurv;
}
```

```
/* spawnElements spawn one element at a point.
   It skips the point if it's too close to the curves.
*/
private List<Surface> spawnElements(
List<Tuple<Point3d, SurfaceCurvature>> ptCurv, List<Curve> curves){
  List<Surface> elements = new List<Surface>();

  foreach(Tuple<Point3d, SurfaceCurvature> pc in ptCurv){
    Point3d pt = pc.Item1;
    if(!closeToRoad(pt, curves)){
      SurfaceCurvature curv = pc.Item2;
      // spawn one element
      double k1 = curv.Kappa(0);
      double k2 = curv.Kappa(1);
      if (k1 < 0.1 && k2 < 0.1 && k1 > -0.1 && k2 > -0.1){
        elements.AddRange(spawnBuilding(pt, curv));
      } else{
        elements.AddRange(spawnPlants(pt, curv));
      }
    }
  }

  return elements;
}

/* closeToRoad is true if pt is close to curves.
   False otherwise.
 */
private bool closeToRoad(Point3d pt, List<Curve> curves){
  bool isClose = false;
  Point3d projectPt = pt;
  projectPt.Z = 0.0;
  foreach(Curve c in curves){
    double t = 0.0;
    c.ClosestPoint(projectPt, out t);
    double d = projectPt.DistanceTo(c.PointAt(t));
    if(d < 0.6){
      isClose = true;
    }
  }
  return isClose;
}
```

```
/* spawnBuilding spawns a box at pt representing a building
*/
private List<Surface> spawnBuilding(Point3d pt, SurfaceCurvature curv){
  double k1 = curv.Kappa(0);
  double k2 = curv.Kappa(1);
  Vector3d kDir1 = curv.Direction(0);
  Vector3d kDir2 = curv.Direction(1);
  Vector3d normal = Vector3d.CrossProduct(kDir1, kDir2);

  List<Surface> element = new List<Surface>();

  double a = Math.Min(Math.Max(k1 * 10, 0.25), 0.6);
  double b = Math.Min(Math.Max(k2 * 10, 0.25), 0.6);
  Vector3d zvec = Vector3d.ZAxis * a * b * 20;

  Point3d p11 = (pt + a * kDir1 + b * kDir2);
  Point3d p12 = (pt + a * kDir1 - b * kDir2);
  Point3d p13 = (pt - a * kDir1 + b * kDir2);
  Point3d p14 = (pt - a * kDir1 - b * kDir2);
  Point3d p21 = p11 + zvec;
  Point3d p22 = p12 + zvec;
  Point3d p23 = p13 + zvec;
  Point3d p24 = p14 + zvec;

  List<Point3d> basePts = new List<Point3d>();
  basePts.Add(p11);
  basePts.Add(p12);
  basePts.Add(p13);
  basePts.Add(p14);
  List<Point3d> topPts = new List<Point3d>();
  topPts.Add(p21);
  topPts.Add(p22);
  topPts.Add(p23);
  topPts.Add(p24);
  List<Point3d> aPts = new List<Point3d>();
  aPts.Add(p12);
  aPts.Add(p14);
  aPts.Add(p22);
  aPts.Add(p24);
  List<Point3d> bPts = new List<Point3d>();
  bPts.Add(p11);
  bPts.Add(p12);
  bPts.Add(p21);
  bPts.Add(p22);
  List<Point3d> cPts = new List<Point3d>();
  cPts.Add(p13);
  cPts.Add(p14);
  cPts.Add(p23);
  cPts.Add(p24);
```

```
    List<Point3d> dPts = new List<Point3d>();
    dPts.Add(p13);
    dPts.Add(p11);
    dPts.Add(p23);
    dPts.Add(p21);

    element.Add(NurbsSurface.CreateFromPoints(basePts, 2, 2, 1, 1));
    element.Add(NurbsSurface.CreateFromPoints(topPts, 2, 2, 1, 1));
    element.Add(NurbsSurface.CreateFromPoints(aPts, 2, 2, 1, 1));
    element.Add(NurbsSurface.CreateFromPoints(bPts, 2, 2, 1, 1));
    element.Add(NurbsSurface.CreateFromPoints(cPts, 2, 2, 1, 1));
    element.Add(NurbsSurface.CreateFromPoints(dPts, 2, 2, 1, 1));
    return element;
  }

  /* spawnPlants spawns a sphere and a box at pt
     representing a tree
  */
  private List<Surface> spawnPlants(Point3d pt, SurfaceCurvature curv){
    Vector3d kDir1 = 0.04 * curv.Direction(0);
    Vector3d kDir2 = 0.04 * curv.Direction(1);
    Vector3d normal = curv.Normal;
    Vector3d zvec = 0.5 * normal;

    List<Surface> element = new List<Surface>();

    Point3d p11 = (pt + kDir1 + kDir2);
    Point3d p12 = (pt + kDir1 - kDir2);
    Point3d p13 = (pt - kDir1 + kDir2);
    Point3d p14 = (pt - kDir1 - kDir2);
    Point3d p21 = p11 + zvec;
    Point3d p22 = p12 + zvec;
    Point3d p23 = p13 + zvec;
    Point3d p24 = p14 + zvec;

    Point3d ps = (p21 + p22 + p23 + p24) / 4 + zvec * 0.5;
    Surface s = (new Sphere(ps, 0.3)).ToNurbsSurface();
    element.Add(s);

    List<Point3d> basePts = new List<Point3d>();
    basePts.Add(p11);
    basePts.Add(p12);
    basePts.Add(p13);
    basePts.Add(p14);
    List<Point3d> topPts = new List<Point3d>();
    topPts.Add(p21);
    topPts.Add(p22);
    topPts.Add(p23);
    topPts.Add(p24);
    List<Point3d> aPts = new List<Point3d>();
    aPts.Add(p12);
    aPts.Add(p14);
    aPts.Add(p22);
    aPts.Add(p24);
    List<Point3d> bPts = new List<Point3d>();
    bPts.Add(p11);
    bPts.Add(p12);
    bPts.Add(p21);
    bPts.Add(p22);
    List<Point3d> cPts = new List<Point3d>();
    cPts.Add(p13);
    cPts.Add(p14);
    cPts.Add(p23);
    cPts.Add(p24);
    List<Point3d> dPts = new List<Point3d>();
    dPts.Add(p13);
    dPts.Add(p11);
    dPts.Add(p23);
    dPts.Add(p21);

    element.Add(NurbsSurface.CreateFromPoints(basePts, 2, 2, 1, 1));
    element.Add(NurbsSurface.CreateFromPoints(topPts, 2, 2, 1, 1));
    element.Add(NurbsSurface.CreateFromPoints(aPts, 2, 2, 1, 1));
    element.Add(NurbsSurface.CreateFromPoints(bPts, 2, 2, 1, 1));
    element.Add(NurbsSurface.CreateFromPoints(cPts, 2, 2, 1, 1));
    element.Add(NurbsSurface.CreateFromPoints(dPts, 2, 2, 1, 1));
    return element;
  }
  /**/
}
```