

QuadSdk

A Programmer's Quick Guide

Computations & Graphics, Inc.

Highlands Ranch, CO 80130, USA

<https://www.cg-inc.com>

Chapter 0 - Introduction

QuadSdk is a 64-bit Windows finite element mesh generator that produces 100% quadrilateral elements (shell4) on planar and curved surfaces in 3D space. QuadSdk supports both .NET languages such as C# and VB.NET, and native C++ language. A mesh model is defined and solved through an easy-to-use Application Programming Interface (API). A mesh model consists of one or more mesh regions (aka sub-regions) of different geometrical shapes. The supported regions include flat surface, cylindrical surface, spherical surface, surface of revolution (rev), and general patch surface. With these types of regions, arbitrarily complex geometrical shapes can be constructed.

QuadSdk is based on the mesh generation engine used in QuadMaker – an interactive 64-bit Windows finite element mesh generator. You can use QuadMaker to open and graphically view model files generated by QuadSdk. You can use QuadMaker to automatically generate QuadSdk source code that corresponds to the current QuadMaker model. Therefore, it is highly recommended that you download and install a copy of QuadMaker.

Nodes

A **node** is a 3D point in space. It is identified by a number called node id.

Curves

A **curve** represents a boundary part of a surface. It is identified by a number called curve id. There are three types of curves:

1. QL-type curve which is a line segment defined by two nodes.
2. C-type curve which is an arc defined in 3D space by three consequent nodes not belonging to one and the same line.
3. Q-type curve which is a general splice consists of four or more nodes. The curve must neither self-intersect nor contain cusps;

An important property in each curve is “step size” (STEP), a user-defined distance between neighboring nodes that are to be generated along the curve during the preprocessing stage.

Edges

An edge is a non-intersecting sequence of QL-type, C-type, and/or Q-type curves. Edges are used in sphere, rev, and general patch regions. They are useful when parts of a surface boundary contain one or several cusps.

Regions (aka sub-regions)

A **region** (aka sub-region) is a closed, non-intersecting sequence of curves that defines a 3D surface.

Plane Region

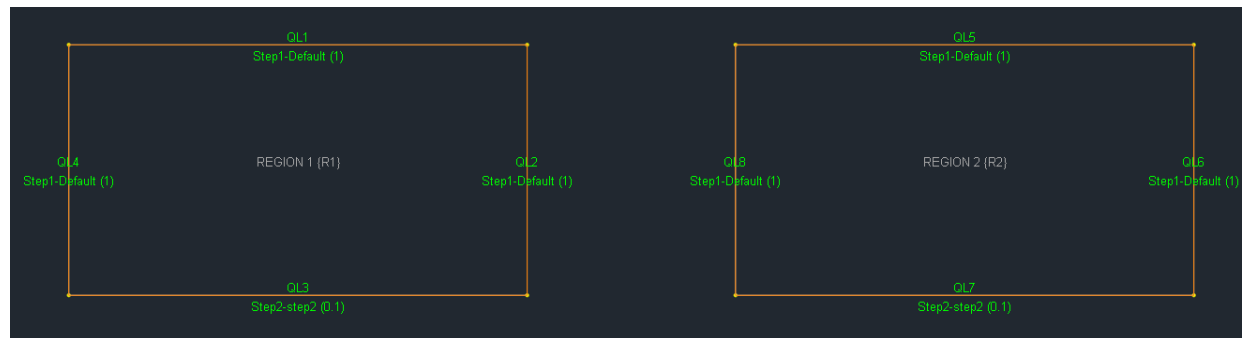
A PLANE-type region is defined by a sequence of curves that belongs to one and the same flat surface. Different plane regions in a mesh model may lie on different planes. It is important to point out that the boundary curves may be specified in arbitrary order. Their actual order will be restored automatically during the preprocessing stage. This is a very convenient feature in QuadSdk.

A region can have one or more dependents such as holes, internal points, and internal lines (aka. trees).

An additional parameter in a region called REF (refinement coefficient) can be specified with a value between [0.0 and 0.35]. This parameter can, to some extent, regulate the density of mesh. Here is how the REF value works:

1. If STEP sizes for all curves comprising the boundary of a region are set to the same or close values, then all parts of the sub-region will be meshed more or less the same density. In such a case, the value of parameter REF is not important and can be set to 0.
2. If there are significant variations in the step sizes along the region, the mesh will have transitional areas between parts of fine mesh (originating near those curves that have smaller step sizes) and coarse mesh (originating near curves with large step sizes). In that case the user may specify how far the parts of the fine mesh can "penetrate" into the interior of the sub-region. As a general rule, the higher the value of REF, the more space will be occupied by fine mesh.

The following (Figure 0.1) illustrates two regions with REF value of 0 at the left and REF value of 0.35 at the right.



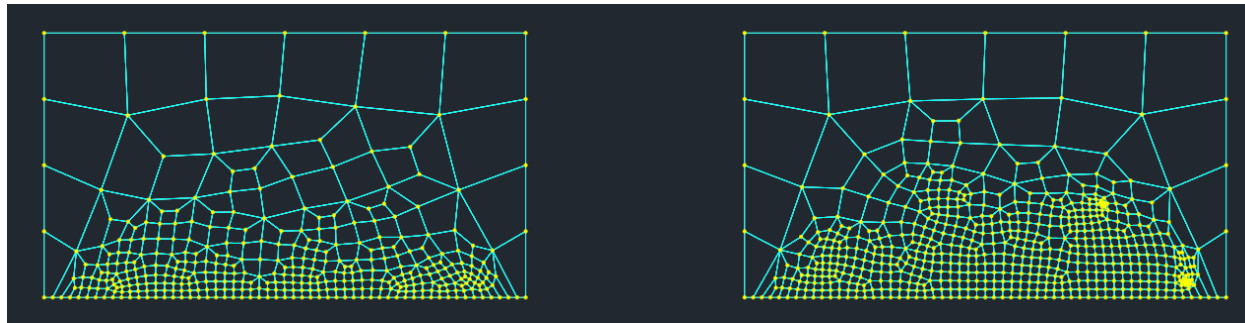
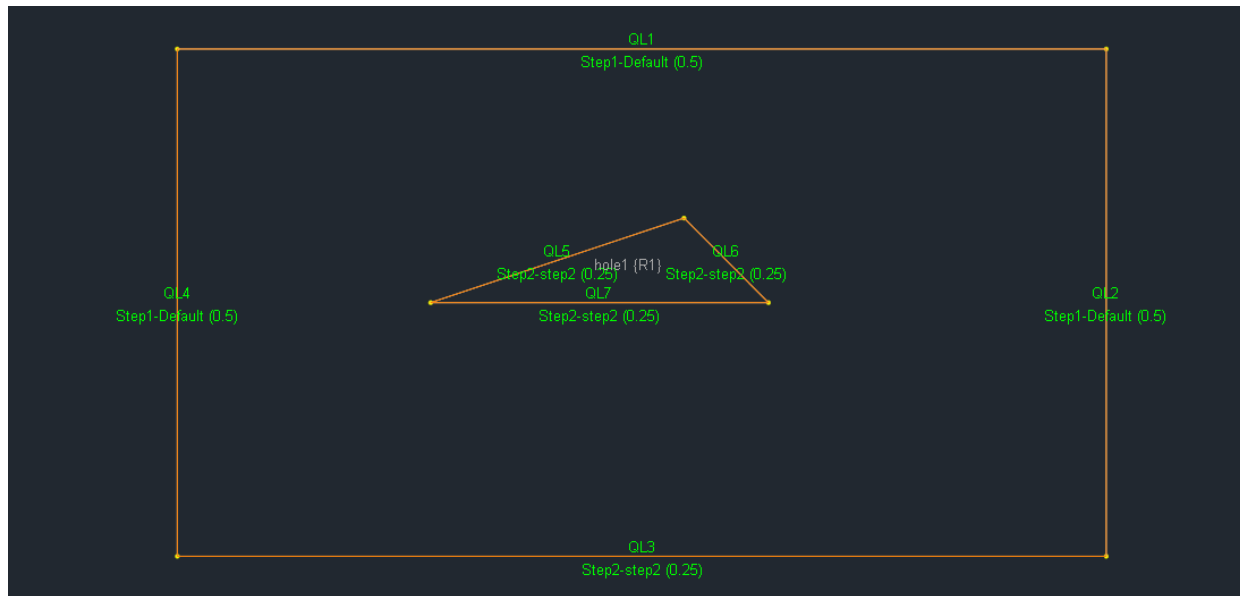


Figure 0.1

Holes

A **hole** in a plane region is defined by a closed, non-intersecting sequence of curves that belong to the same plane as the region. The boundary curves may appear in arbitrary order. Their actual order will be restored automatically during the preprocessing stage.

The following (Figure 0.2) is an example where a triangular hole exists inside a rectangular plane region.



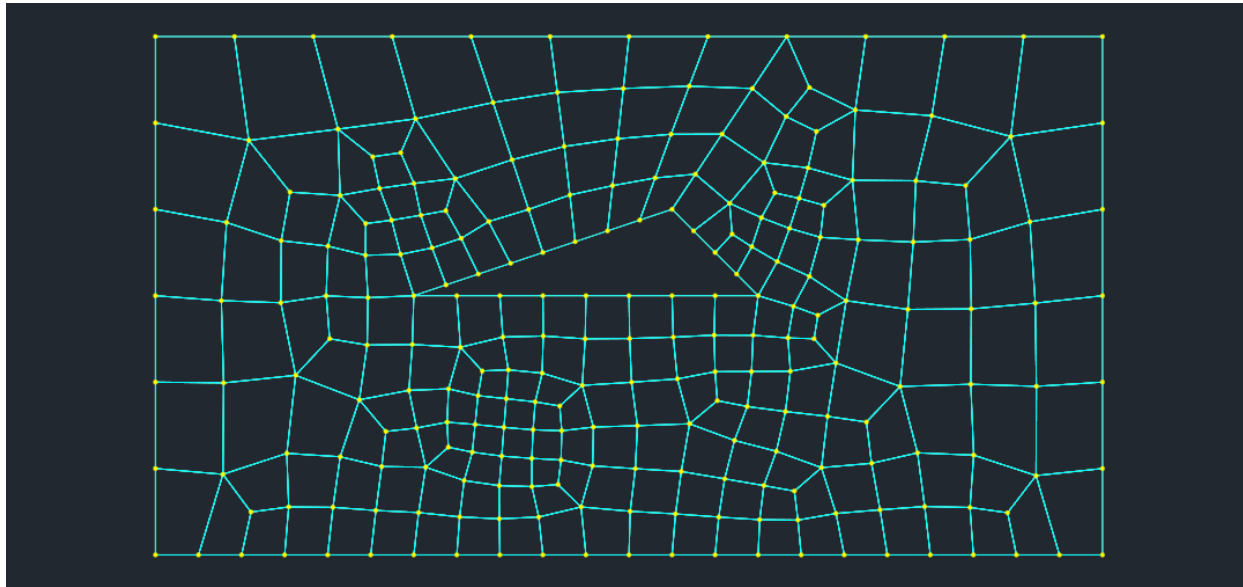


Figure 0.2

Internal Points

An **internal point** (PINT) defines a location that the mesh generation engine is to include a node in the specified location. It is well known that areas of large stress gradients often arise around locations of boundary constraints and point loads. Accurate FEA modeling requires zones of very fine elements to be generated around these PINT-points. Two optional numericals, STEP and RAD, allow the user to set a surrounding refinement area:

STEP: a desirable average edge length of the quadrilaterals around the point

RAD: a desirable radius of the refinement area around the point.

In the following example (Figure 0.3), we have two internal points: one with STEP=0 and RAD=0, and another with STEP=0.1 and RAD=0.25.

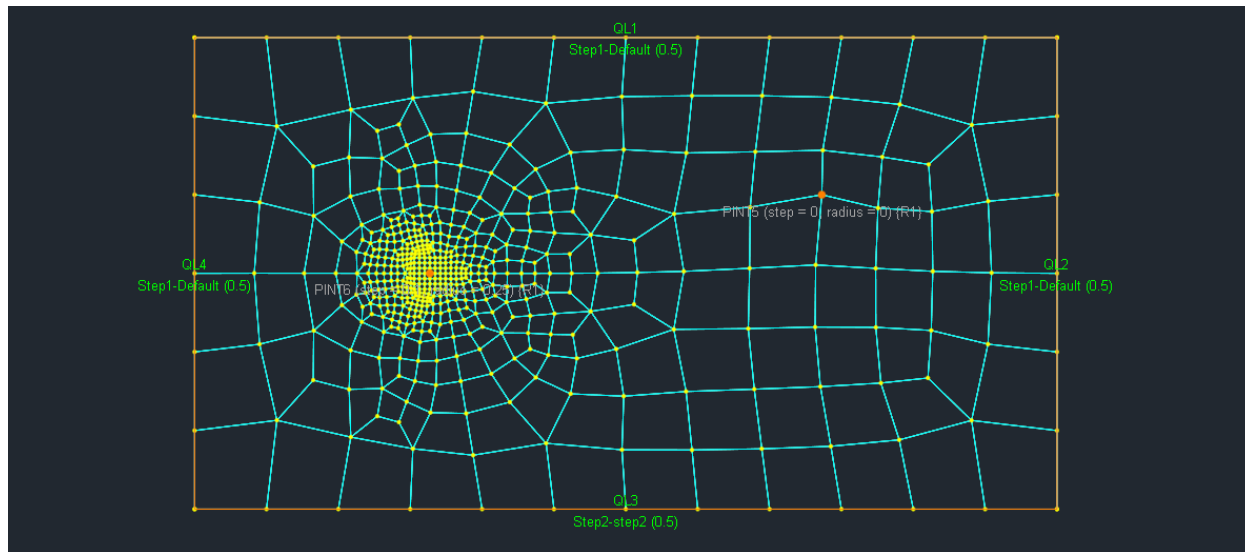
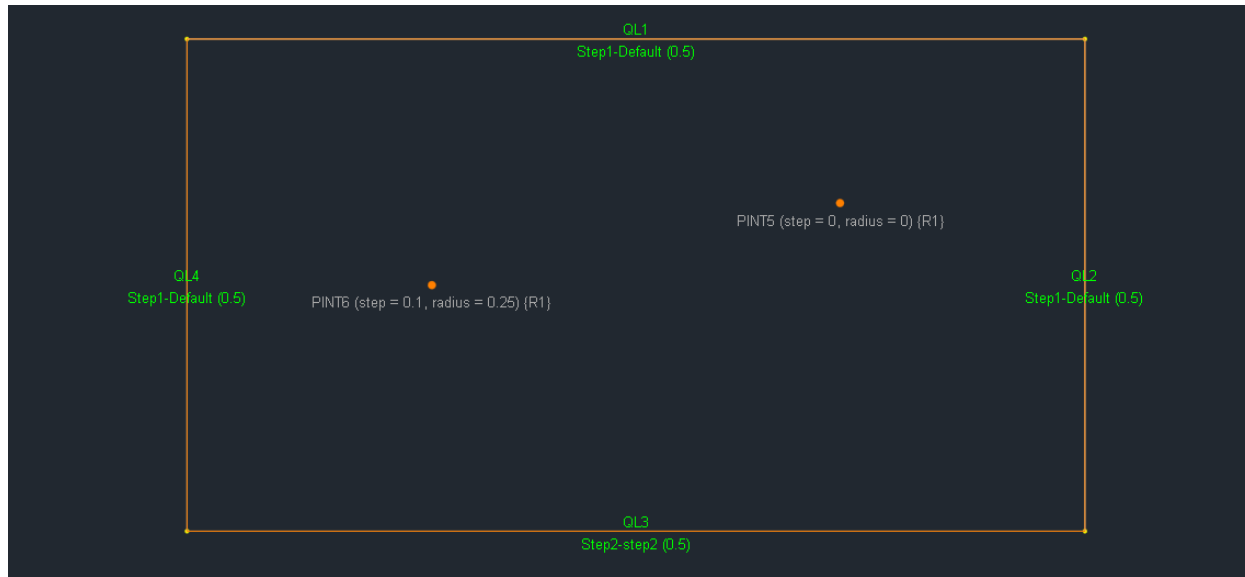
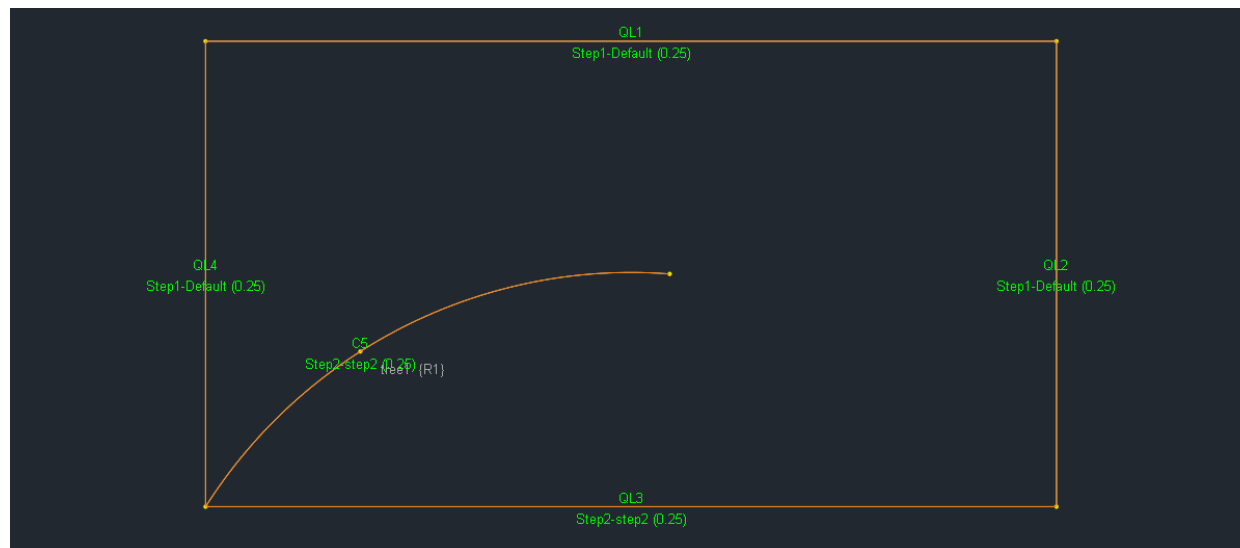


Figure 0.3

Trees

A **tree** (a combination of internal lines) in a region defines a sequence of curves that the mesh generation engine is to include nodes along these curves. Conceptually, a tree is like a physical tree which starts from a root and then branch out. A tree may represent curves, walls, or other boundaries in a structural model. It is important to point out that the curves in a tree must be all connected (no isolated curves). Tree curves must not form internal (closed) contours.

In the following example (Figure 0.4), a tree is defined by a single arc originating from the lower left corner of the rectangular plane region.



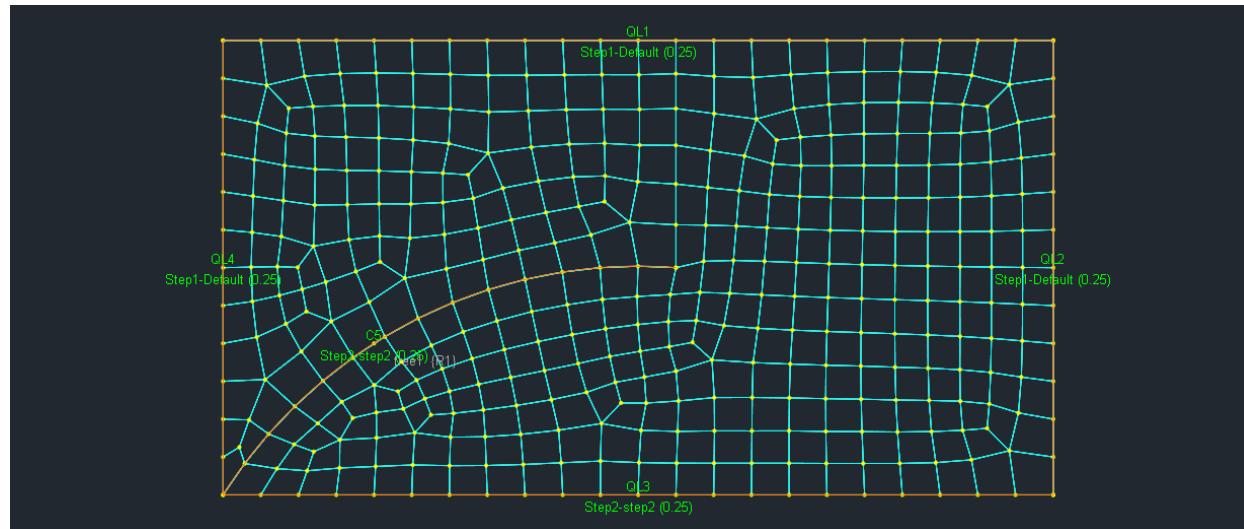


Figure 0.4

Cylinder Region

A CYLINDER-type region is defined by a closed, non-intersecting sequence of curves that belongs to one and the same cylindrical surface formed when a straight line in 3D space travels such that each new position of the line is parallel to its previous position. The cylindrical surface orientation is defined by the directional angles in degrees of the straight line: AngleX, AngleY, and AngleZ. An additional parameter in a region called REF (refinement coefficient) can be specified. For more information, please refer to PLANE region section earlier in this chapter. It is important to point out that the boundary curves may be specified in arbitrary order. Their actual order will be restored automatically during the preprocessing stage. This is a very convenient feature in QuadSdk.

As an example, the following (Figure 0.5) is a mesh model for cylindrical tunnel. All curves have the STEP size of 1 ft. The angleX=90 deg, angleY=90 deg, and angleZ=0 deg are the directional angles (in degrees) of the straight line that defines the cylindrical surface orientation. The mesh refinement coefficient REF is specified as 0.0.

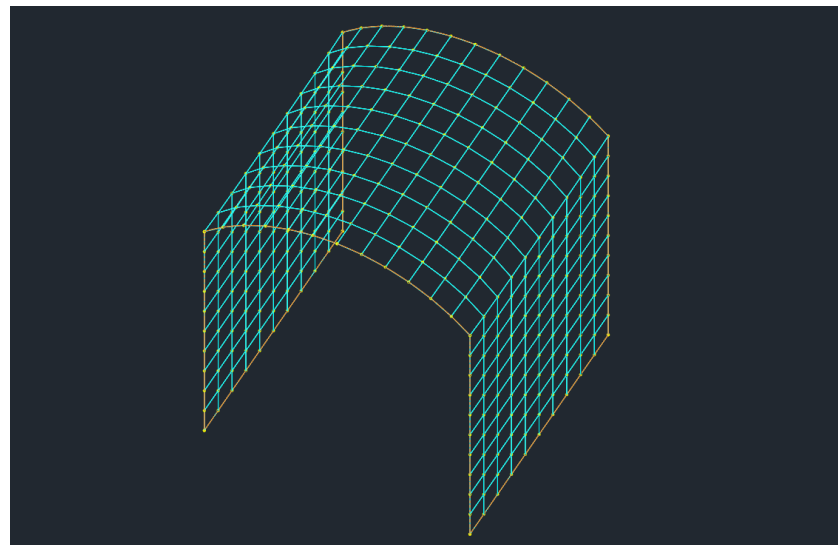
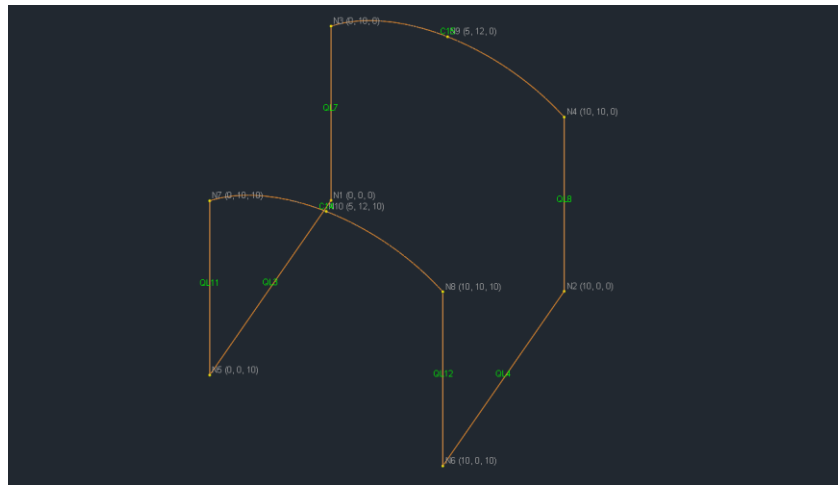


Figure 0.5

Sphere Region

A SPHERE-type region is defined by 2, 3 or 4 edges, all belonging to one and the same spherical surface. The curves in each edge must be arc (C-type) curves. The center coordinates of the sphere are specified as CenterX, CenterY, and CenterZ. An additional parameter in a region called REF (refinement coefficient) can be specified. For more information, please refer to PLANE region section earlier in this chapter. It is important to point out that the boundary edges as well as the curves in each edge may be specified in arbitrary order. Their actual order will be restored automatically during the preprocessing stage. This is a very convenient feature in QuadSdk.

As an example, the following (Figure 0.6) is a mesh model for a quarter of a sphere. The model is defined by two edges (with one arc curve in each edge); each curve has a STEP size of 0.1 ft. The center of the sphere is (0, 0, 0). The mesh refinement coefficient REF is specified as 0.0.

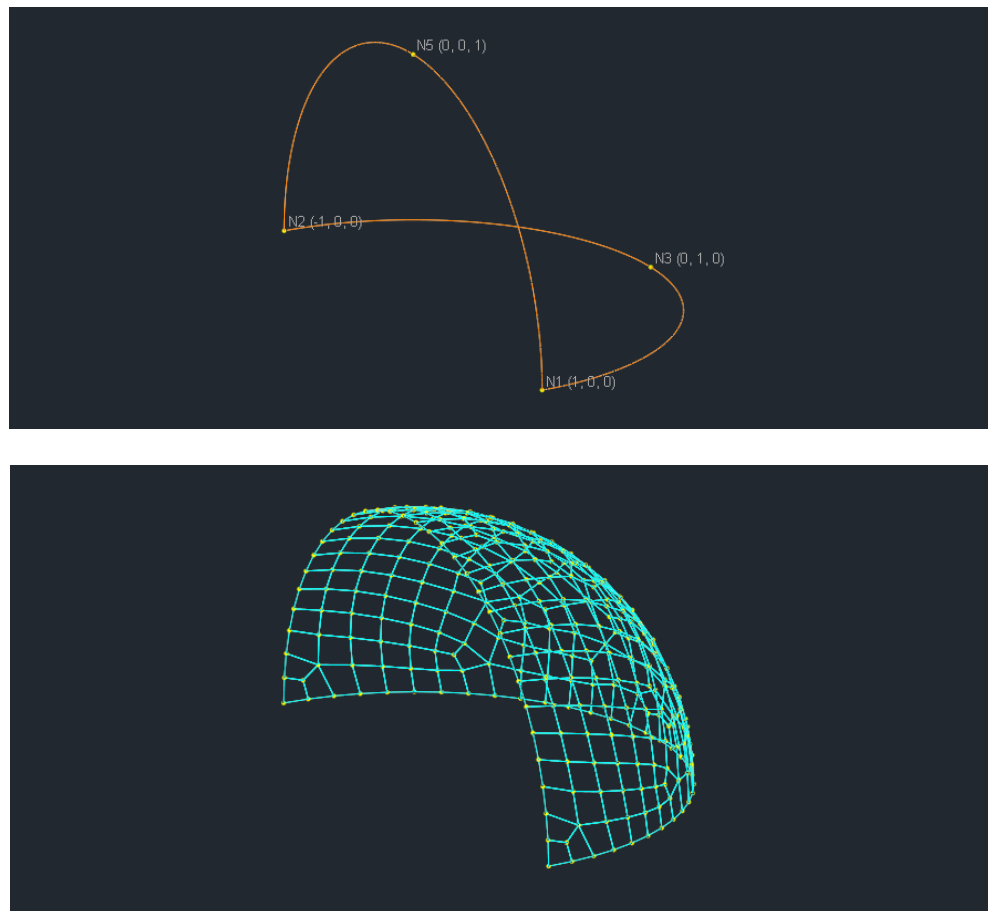
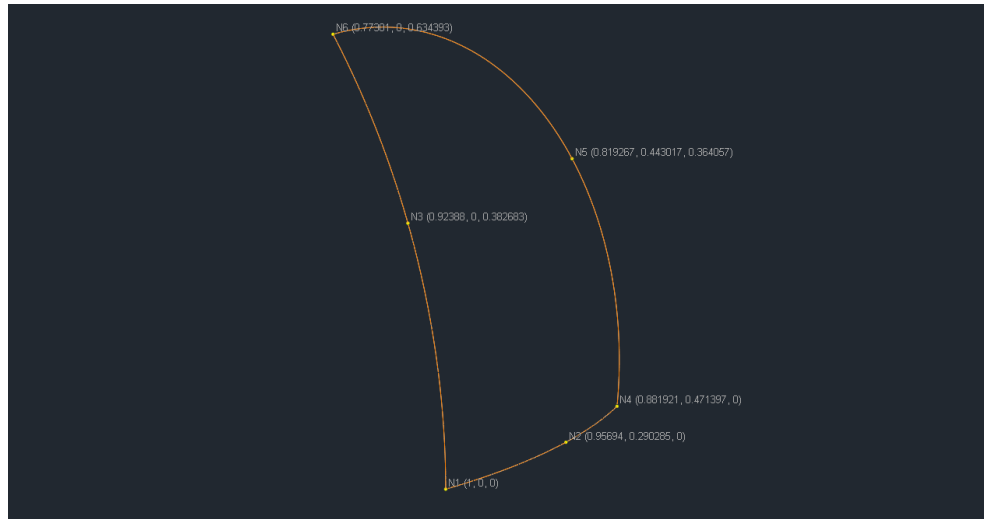


Figure 0.6

As another example, the following (Figure 0.7) is a mesh model for a small patch of a sphere. The model is defined by three edges (with one arc curve in each edge); each curve has a STEP size of 0.1 ft. The center of the sphere is (0, 0, 0). The mesh refinement coefficient REF is specified as 0.0.



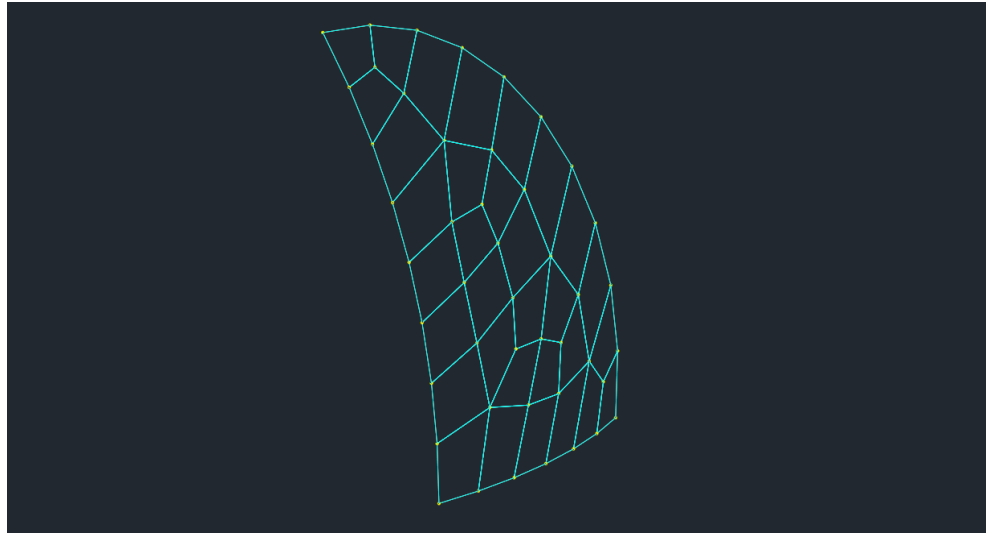


Figure 0.7

Rev Region

A REV-type region is defined by exactly 4 edges, two of which must be C-type curves representing the "upper" and the "lower" arcs of the surface of revolution correspondingly. The upper and lower arcs belong to parallel planes that are orthogonal to the axis of revolution. Two other edges, describing the "left" and the "right" sides of the patch, define the actual shape of the surface of revolution. They may consist of unequal numbers of curves. The corresponding start and end points of the curves must belong to one and the same plane orthogonal to the axis of revolution. An additional parameter in a region called REF (refinement coefficient) can be specified. For more information, please refer to PLANE region section earlier in this chapter. It is important to point out that the boundary edges as well as the curves in each edge may be specified in arbitrary order. Their actual order will be restored automatically during the preprocessing stage. This is a very convenient feature in QuadSdk.

As an example, the following (Figure 0.8) is a mesh model for a half of a cone. The model is defined by four edges: upper and lower arcs, left and right sides. Each of the curves has a STEP size of 0.1 ft. The mesh refinement coefficient REF is specified as 0.0.

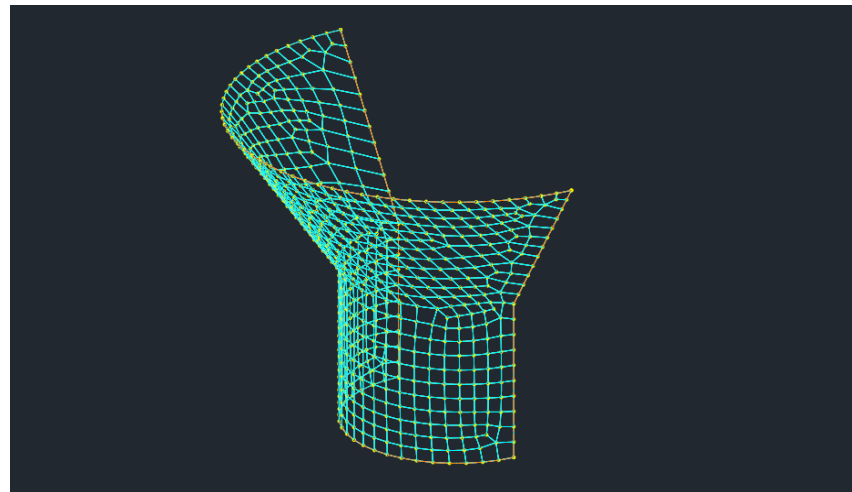
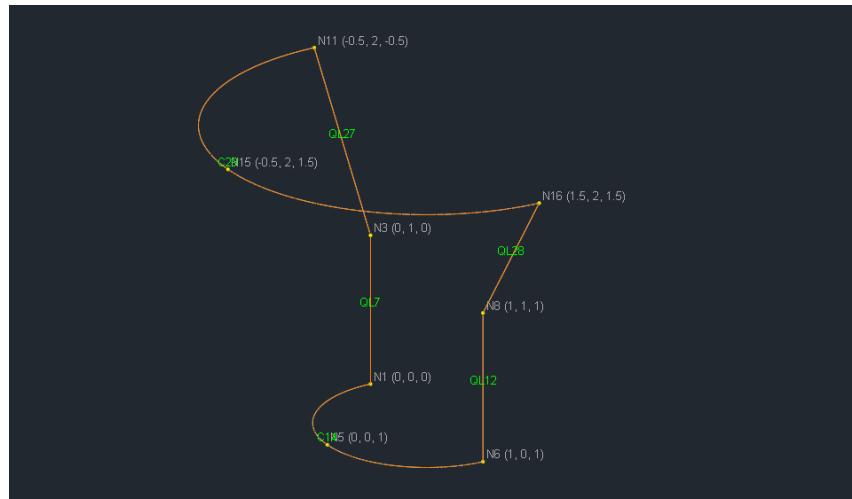


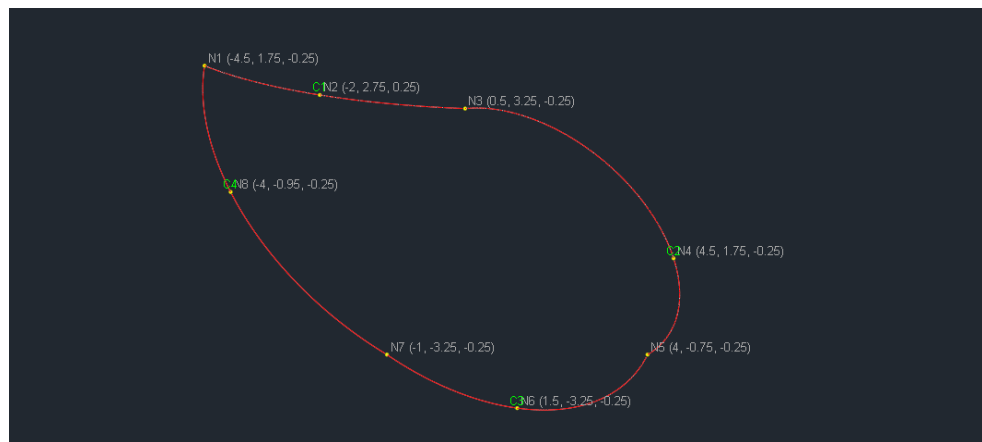
Figure 0.8

General Patch Region

A GENERAL PATCH-type region is defined by 2, 3, or 4 edges. One edge may consist of one or more curves. The latter option is useful in cases when the user wants an edge to be divided into several parts with different step sizes; or when a particular edge contains one or several cusps. An additional parameter in a region called REF (refinement coefficient) can be specified. For more information, please refer to PLANE region section earlier in this chapter. It is important to point out that the boundary edges as well as the curves in each edge may be specified in arbitrary order. Their actual order will be restored automatically during the preprocessing stage. This is a very convenient feature in QuadSdk.

General-type patches have to be used in situations where no other information about the patch (i.e. actual type of surface) is available except of its boundary curves.

As an example, the following (Figure 0.9) is a mesh general model. The model consists of four arc edges. Each of the curves has a STEP size of 0.25 ft. The mesh refinement coefficient REF is specified as 0.0.



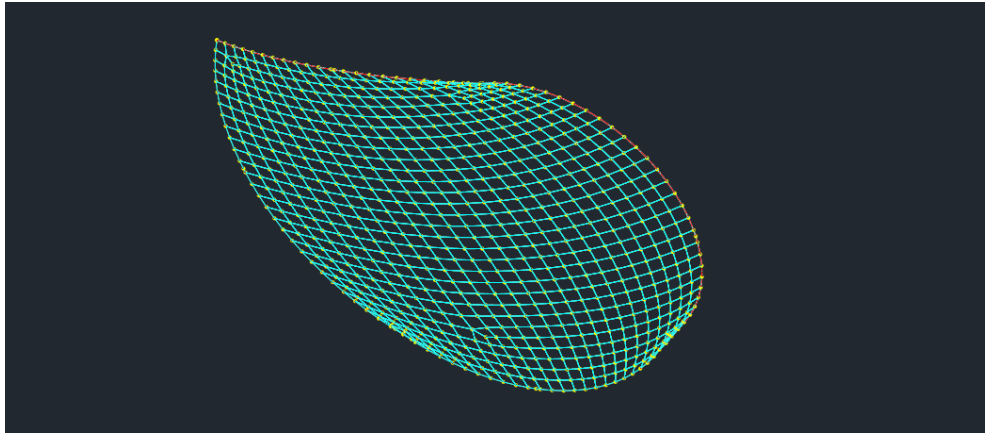


Figure 0.9

Chapter 1 - .NET Interface

The .NET Framework interface is a 64-bit Class Library DLL `cgiQuadSdkCli.dll`. It is written in C++/CLI and makes calls to the native Windows DLL `cgiQuadSdk.dll` which has a dependency on another native DLL called `QuadSurfaceDll.dll`. QuadSdk supports .NET Framework version 4.0-4.8. QuadSdk also supports .NET Core 5.0-10.0 `cgiQuadSdkClass` is the one and only interface to set input, perform meshing, and retrieve output. The release version of `QuadSurfaceDll.dll`, `cgiQuadSdk.dll`, and `cgiQuadSdkCli.dll` must be placed in the executable directory. For .NET Core, `ijwhost.dll` is a **shim** for finding and loading the runtime. All C++/CLI libraries are linked to this shim, such that `ijwhost.dll` is found/loaded when the C++/CLI library is loaded. Therefore, make sure a copy of `ijwhost.dll` is placed in the executable directory. *For distribution, you may need to install Visual C++ 2015~2026 x64 redistributables as `cgiQuadSdkCli.dll` and `cgiQuadSdkCoreCli.dll` link dynamically with the visual C++ runtime library.*

The best way to learn how you use these data structures and interfaces is to study the examples included in the C# console application project `cgiQuadSdkTestCSharp`. These examples are taken from the User's Manual QuadMaker, which is an interactive quadrilateral meshing software by CGI.

The following lists all the interface functions exposed by `cgiQuadSdkClass` which is in the namespace `cgiQuadSdkCli` for .NET Framework version and `cgiQuadSdkCoreCli` for .NET Core version.

```
public class cgiQuadSdkClass : IDisposable
{
    public void setStatusMessageFunction(cgiQuadSdkClass.StatusMessageDelegate fnStatusMsg);

    public void setProjPath(string projPath);
    public void getProjPath(ref StringBuilder projPath);

    public void setStandardEnglishUnits();
    public void setStandardMetricUnits();
    public void setConsistentEnglishUnits();
    public void setConsistentMetricUnits();

    public void setNodes(List<cgiNodeCli> listNd);
    public void getNodes(ref List<cgiNodeCli> listNd);
    public void setCurves(List<cgiCurveCli> listBm);
    public void getCurves(ref List<cgiCurveCli> listBm);

    public void getSingleNode(ref cgiNodeCli item, int nId);
    public void getSingleCurve(ref cgiCurveCli item, int nId);
}
```

```

public void addPlaneRegion(cgiMeshPlaneRegionCli region);
public void addCylinderRegion(cgiMeshCylinderRegionCli region);
public void addSphereRegion(cgiMeshSphereRegionCli region);
public void addRevRegion(cgiMeshRevRegionCli region);
public void addGeneralRegion(cgiMeshGeneralRegionCli region);

public void getPlaneRegion(ref cgiMeshPlaneRegionCli region, int regionId);
public void getCylinderRegion(ref cgiMeshCylinderRegionCli region, int regionId);
public void getSphereRegion(ref cgiMeshSphereRegionCli region, int regionId);
public void getRevRegion(ref cgiMeshRevRegionCli region, int regionId);
public void getGeneralRegion(ref cgiMeshGeneralRegionCli region, int regionId);

public void getPlaneRegions(ref List<cgiMeshPlaneRegionCli> vRegion);
public void getCylinderRegions(ref List<cgiMeshCylinderRegionCli> vRegion);
public void getSphereRegions(ref List<cgiMeshSphereRegionCli> vRegion);
public void getRevRegions(ref List<cgiMeshRevRegionCli> vRegion);
public void getGeneralRegions(ref List<cgiMeshGeneralRegionCli> vRegion);

public bool exportToQuadMaker();
public bool exportToReal3D();
public int checkInputData(ref string szMsg);
public void setSolverParameters(int max_subregions, int max_curves_per_tree, int max_points_per_tree, int max_nodel_3d);
public void getSolverParameters(ref int max_subregions, ref int max_curves_per_tree,
                                ref int max_points_per_tree, ref int max_nodel_3d);

public int solve();
public void alignShell4LocalZWithReferencePoint(cgiPointCli referencePoint);
public void getResults(ref List<cgiNodeCli> vNd, ref List<cgiShell4Cli> vShell4);

public void clearNodes();
public void clearCurves();
public void clearRegions();
public void clear();

public bool createStructure();
}

```

The following lists classes and enums used by `cgiQuadSdkClass`

```
public enum cgiDofEnum
{
    X,
    Y,
    Z,
    OX,
    OY,
    OZ,
}
```

```
public enum cgiErrorEnum
{
    ERROR_NONE,
    ERROR_LOADING_DLL,
    ERROR_FINDING_DLL_FUNCTIONS,
    ERROR_CREATING_INPUT_FILE,
    ERROR_SOLVER,
    TOO_MANY_SUBREGIONS,
    TOO_MANY_INTERNAL_POINTS,
    TOO_MANY_TREES_IN_REGION,
    TOO_MANY_HOLES_IN_REGION,
    TOO_MANY_CURVES_IN_HOLE,
    TOO_MANY_CURVES_IN_TREE,
    CURVES_NOT_CONNECTED_IN_TREE,
    REGION_BOUNDARY_NOT_CLOSED,
    NODE_NOT_FOUND,
    CURVE_NOT_FOUND,
    ERROR_EVALUAION_LIMIT,
}
```

```
public class cgiPointCli
{
    public double x;
    public double y;
    public double z;

    public cgiPointCli(double _x, double _y, double _z);
    public cgiPointCli();
    public void setCoordinates(double _x, double _y, double _z);
}
```

```

}

public class cgiNodeCli
{
    public int iId;
    public double x;
    public double y;
    public double z;

    public void setId(int _iId);
    public void setCoordinates(double _x, double _y, double _z);
}

public class cgiCurveCli
{
    public int iId;
    public double step;
    public List<int> nodeIds;

    public cgiCurveCli();
    public void setId(int id);
    public void setStep(double _step);
    public void setNodes(List<int> list);
    public void setNodes(int node1, int node2, int node3);
    public void setNodes(int node1, int node2);
    public void addNode(int _nodeId);
}

public class cgiShell4Cli
{
    public int iId;
    public int node1;
    public int node2;
    public int node3;
    public int node4;

    public void setId(int id);
    public void setNodes(int nodeId1, int nodeId2, int nodeId3, int nodeId4);
}

public class cgiMeshInternalPointCli
{
    public int nodeId;
}

```

```

    public int regionId;
    public double step;
    public double radius;

    public cgiMeshInternalPointCli();
    public void setNodeId(int id);
}

public class cgiMeshHoleCli
{
    public string name;
    public int iId;
    public List<int> curveIds;

    public cgiMeshHoleCli();
    public void setId(int id);
    public void setCurveIds(List<int> list);
    public void addCurveId(int curveId);
}

public class cgiMeshTreeCli
{
    public string name;
    public int iId;
    public List<int> curveIds;

    public cgiMeshTreeCli();
    public void setId(int id);
    public void setCurveIds(List<int> list);
    public void addCurveId(int curveId);
}

public class cgiMeshEdgeCli
{
    public List<int> curveIds;
    public cgiMeshEdgeCli();
    public void setCurveIds(List<int> list);
    public void addCurveId(int curveId);
}

public class cgiMeshRegionBaseCli
{
    public int iId;

```

```

    public string name;
    public double refValue;
    public bool disabled;

    public void setId(int _iId);
    public List<cgiMeshEdgeCli> getEdges();
    protected cgiMeshRegionBaseCli();
}

public class cgiMeshPlaneRegionCli : cgiMeshRegionBaseCli
{
    public List<cgiMeshInternalPointCli> internalPoints;
    public List<cgiMeshHoleCli> holes;
    public List<cgiMeshTreeCli> trees;

    public cgiMeshPlaneRegionCli();
    public void setEdge(cgiMeshEdgeCli edge);
    public void addInternalPoint(cgiMeshInternalPointCli internalPt);
    public void addHole(cgiMeshHoleCli h);
    public void addTree(cgiMeshTreeCli t);
}

public class cgiMeshCylinderRegionCli : cgiMeshRegionBaseCli
{
    public double angleX;
    public double angleY;
    public double angleZ;

    public cgiMeshCylinderRegionCli();
    public void setEdge(cgiMeshEdgeCli edge);
}

public class cgiMeshSphereRegionCli : cgiMeshRegionBaseCli
{
    public double centerX;
    public double centerY;
    public double centerZ;

    public cgiMeshSphereRegionCli();
    public void setEdges(List<cgiMeshEdgeCli> list);
}

public class cgiMeshRevRegionCli : cgiMeshRegionBaseCli

```

```
{  
    public void setEdges(List<cgiMeshEdgeCli> list);  
}  
  
public class cgiMeshGeneralRegionCli : cgiMeshRegionBaseCli  
{  
    public void setEdges(List<cgiMeshEdgeCli> list);  
}
```

Example Model

The following (Figure 1.1) is a four-sided mesh model that includes an opening, a tree of two lines, an internal point.

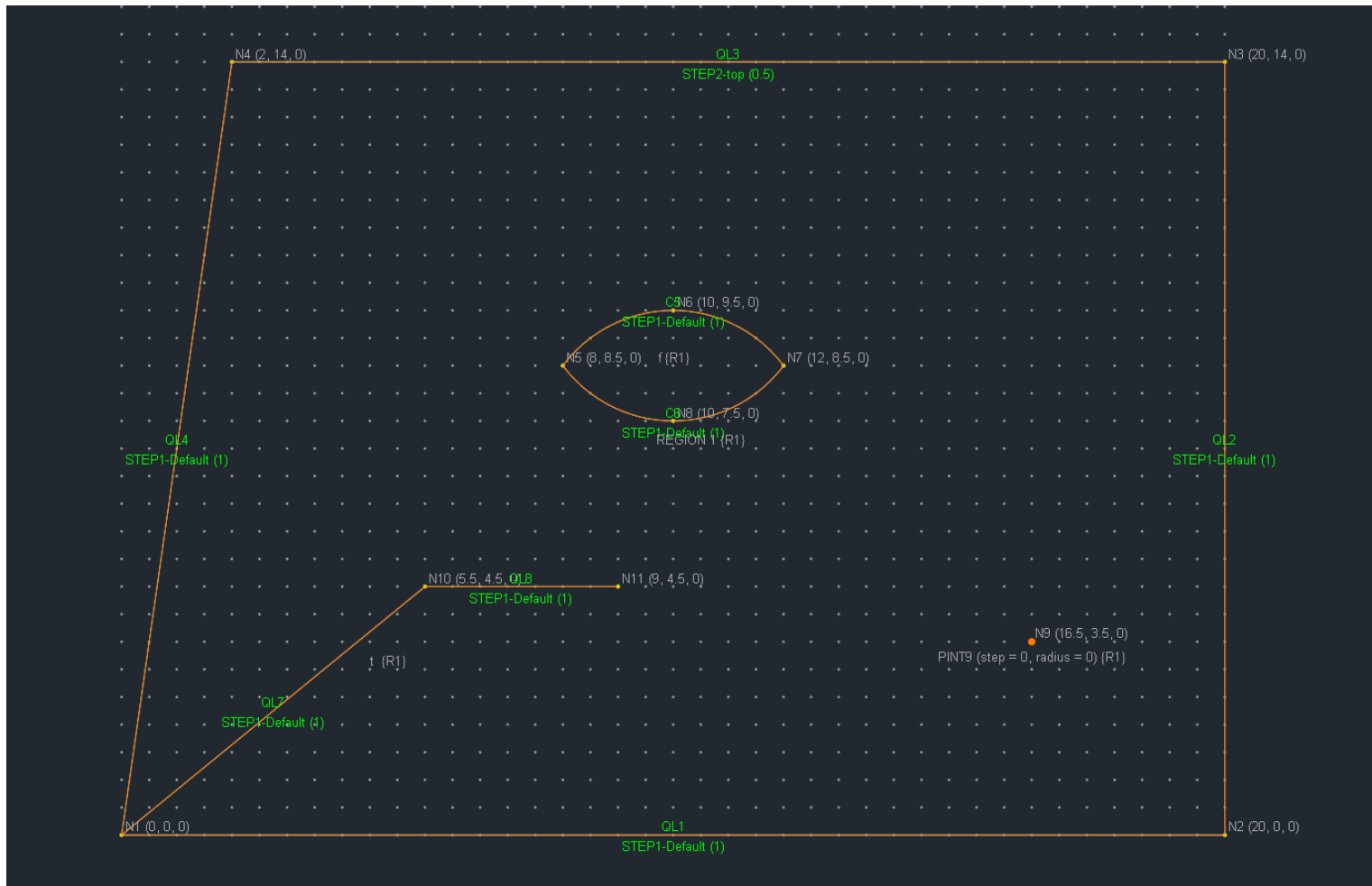


Figure 1.1

The following is the entire C# code that will create the mesh model above, then solve it, and then retrieve the mesh results.

```
using System;
using System.Collections.Generic;
using System.Text;
using cgiQuadSdkCli;

namespace cgiQuadSdkTestCSharp
{
    public class demo
    {
        static void StatusCallback(string s)
        {
            Console.WriteLine("{0}", s);
        }

        static public void verify()
        {
            RunModel();
        }

        static public void RunModel()
        {
            cgiQuadSdkClass solver = new cgiQuadSdkClass();
            solver.createStructure();
            cgiQuadSdkClass.StatusMessageDelegate statusMsg = new cgiQuadSdkClass.StatusMessageDelegate(StatusCallback);
            solver.setStatusMessageFunction(statusMsg);
            solver.setProjPath(@"C:\CGInc\QuadSdk\SampleCode\Models\dotNet\demo_");

            // LENGTH=ft;    DIMENSION=in;
            solver.setStandardEnglishUnits();

            // define nodes
            List<cgiNodeCli> listNode = new List<cgiNodeCli>();
            cgiNodeCli node1 = new cgiNodeCli();
            node1.setId(1);
            node1.setCoordinates(0, 0, 0);
            listNode.Add(node1);
            cgiNodeCli node2 = new cgiNodeCli();
            node2.setId(2);
            node2.setCoordinates(20, 0, 0);
            listNode.Add(node2);
            cgiNodeCli node3 = new cgiNodeCli();
            node3.setId(3);
            node3.setCoordinates(20, 14, 0);
            listNode.Add(node3);
            cgiNodeCli node4 = new cgiNodeCli();
```

```

node4.setId(4);
node4.setCoordinates(2, 14, 0);
listNode.Add(node4);
cgiNodeCli node5 = new cgiNodeCli();
node5.setId(5);
node5.setCoordinates(8, 8.5, 0);
listNode.Add(node5);
cgiNodeCli node6 = new cgiNodeCli();
node6.setId(6);
node6.setCoordinates(10, 9.5, 0);
listNode.Add(node6);
cgiNodeCli node7 = new cgiNodeCli();
node7.setId(7);
node7.setCoordinates(12, 8.5, 0);
listNode.Add(node7);
cgiNodeCli node8 = new cgiNodeCli();
node8.setId(8);
node8.setCoordinates(10, 7.5, 0);
listNode.Add(node8);
cgiNodeCli node9 = new cgiNodeCli();
node9.setId(9);
node9.setCoordinates(16.5, 3.5, 0);
listNode.Add(node9);
cgiNodeCli node10 = new cgiNodeCli();
node10.setId(10);
node10.setCoordinates(5.5, 4.5, 0);
listNode.Add(node10);
cgiNodeCli node11 = new cgiNodeCli();
node11.setId(11);
node11.setCoordinates(9, 4.5, 0);
listNode.Add(node11);
solver.setNodes(listNode);

// define curves
List<cgiCurveCli> listCurve = new List<cgiCurveCli>();
cgiCurveCli curve1 = new cgiCurveCli();
curve1.setId(1);
curve1.setStep(1);
curve1.setNodes(1, 2);
listCurve.Add(curve1);
cgiCurveCli curve2 = new cgiCurveCli();
curve2.setId(2);
curve2.setStep(1);
curve2.setNodes(2, 3);
listCurve.Add(curve2);
cgiCurveCli curve3 = new cgiCurveCli();
curve3.setId(3);
curve3.setStep(0.5);
curve3.setNodes(3, 4);
listCurve.Add(curve3);
cgiCurveCli curve4 = new cgiCurveCli();
curve4.setId(4);

```

```

curve4.setStep(1);
curve4.setNodes(4, 1);
listCurve.Add(curve4);
cgiCurveCli curve5 = new cgiCurveCli();
curve5.setId(5);
curve5.setStep(1);
curve5.addNode(5);
curve5.addNode(6);
curve5.addNode(7);
listCurve.Add(curve5);
cgiCurveCli curve6 = new cgiCurveCli();
curve6.setId(6);
curve6.setStep(1);
curve6.addNode(5);
curve6.addNode(8);
curve6.addNode(7);
listCurve.Add(curve6);
cgiCurveCli curve7 = new cgiCurveCli();
curve7.setId(7);
curve7.setStep(1);
curve7.setNodes(1, 10);
listCurve.Add(curve7);
cgiCurveCli curve8 = new cgiCurveCli();
curve8.setId(8);
curve8.setStep(1);
curve8.setNodes(10, 11);
listCurve.Add(curve8);
solver.setCurves(listCurve);

// define edges for planeRegion1
cgiMeshEdgeCli planeRegion1_edge1 = new cgiMeshEdgeCli();
planeRegion1_edge1.addCurveId(1);
planeRegion1_edge1.addCurveId(2);
planeRegion1_edge1.addCurveId(3);
planeRegion1_edge1.addCurveId(4);

// define internal points for planeRegion1
cgiMeshInternalPointCli planeRegion1_internalPoint1 = new cgiMeshInternalPointCli();
planeRegion1_internalPoint1.nodeId = 9;
planeRegion1_internalPoint1.regionId = 1;
planeRegion1_internalPoint1.step = 0;
planeRegion1_internalPoint1.radius = 0;

// define trees for planeRegion1
cgiMeshTreeCli planeRegion1_tree1 = new cgiMeshTreeCli();
planeRegion1_tree1.iId = 1;
planeRegion1_tree1.name = "t";
planeRegion1_tree1.addCurveId(7);
planeRegion1_tree1.addCurveId(8);

// define holes for planeRegion1
cgiMeshHoleCli planeRegion1_hole1 = new cgiMeshHoleCli();

```

```

planeRegion1_hole1.iId = 1;
planeRegion1_hole1.name = "f";
planeRegion1_hole1.addCurveId(5);
planeRegion1_hole1.addCurveId(6);

cgiMeshPlaneRegionCli planeRegion1 = new cgiMeshPlaneRegionCli();
planeRegion1.iId = 1;
planeRegion1.name = "REGION 1";
planeRegion1.refValue = 0;

// set edges for planeRegion1
planeRegion1.setEdge(planeRegion1_edge1);

planeRegion1.addInternalPoint(planeRegion1_internalPoint1);

planeRegion1.addTree(planeRegion1_tree1);

planeRegion1.addHole(planeRegion1_hole1);

solver.addPlaneRegion(planeRegion1);

solver.exportToQuadMaker(); // export input mesh model to QuadMaker

int ret = (int)cgiErrorEnum.ERROR_NONE;
string msg = "";
ret = solver.checkInputData(ref msg);
if (ret != (int)cgiErrorEnum.ERROR_NONE)
{
    Console.WriteLine("Invalid Input : " + msg);
    return;
};

ret = solver.solve(); // solve
if (ret != (int)cgiErrorEnum.ERROR_NONE)
{
    Console.WriteLine("Error encountered");
    return;
};

solver.exportToReal3D(); // export generated mesh to a Real3D file

List<cgiNodeCli> result_nodes = new List<cgiNodeCli>();
List<cgiShell4Cli> result_shells = new List<cgiShell4Cli>();
solver.getResults(ref result_nodes, ref result_shells);
for (int i = 0; i < result_nodes.Count; i++)
{
    var n = result_nodes[i];
    Console.WriteLine("node " + n.iId + ", " + n.x + ", " + n.y + ", " + n.z);
}

for (int i = 0; i < result_shells.Count; i++)
{

```

```

        var s = result_shells[i];
        Console.WriteLine("shell " + s.iId + ", " + s.node1 + ", " + s.node2 + ", " + s.node3 + ", " + s.node4);
    }
}
}
}
}

```

The following are the detailed explanations for the above code.

To start, you create a `cgiQuadSdkClass` object that represents the one and only mesh model like this:

```

cgiQuadSdkClass solver = new cgiQuadSdkClass();
solver.createStructure();

```

You can supply one optional callback function for the solver to notify your application the progress of solution:

```

cgiQuadSdkClass.StatusMessageDelegate statusMsg = new cgiQuadSdkClass.StatusMessageDelegate(StatusCallback);
solver.setStatusMessageFunction(statusMsg);

```

The callback function takes a string parameter. Here is a simple example:

```

static void StatusCallback(string s)
{
    Console.WriteLine("{0}", s);
}

```

During the solution, some intermediate files will be generated. We can specify the location of these files like this:

```

solver.setProjPath(@"C:\CGInc\QuadSdk\SampleCode\Models\dotNet\demo_");

```

Four unit systems are available in the library. They are Standard English unit, Standard Metric unit, Consistent English unit and Consistent Metric unit. The following units are used for length and section dimension for each of the unit systems:

```

solver.setStandardEnglishUnits();// LENGTH=ft; DIMENSION=in;

```

```

solver.setConsistentEnglishUnits();// LENGTH=in; DIMENSION=in;

solver.setStandardMetricUnits ( ) ;// LENGTH=m; DIMENSION=mm;

solver.setConsistentMetricUnits ( ) ;// LENGTH=m; DIMENSION=m;

```

Each node is defined by specifying a unique node id (integer number), x, y and z coordinates. All nodes must then be assigned to the model by calling `solver.setNodes()`

```

// define nodes
List<cgiNodeCli> listNode = new List<cgiNodeCli>();
cgiNodeCli node1 = new cgiNodeCli();
node1.setId(1);
node1.setCoordinates(0, 0, 0);
listNode.Add(node1);
cgiNodeCli node2 = new cgiNodeCli();
node2.setId(2);
node2.setCoordinates(20, 0, 0);
listNode.Add(node2);
//...
cgiNodeCli node10 = new cgiNodeCli();
node10.setId(10);
node10.setCoordinates(5.5, 4.5, 0);
listNode.Add(node10);
cgiNodeCli node11 = new cgiNodeCli();
node11.setId(11);
node11.setCoordinates(9, 4.5, 0);
listNode.Add(node11);
solver.setNodes(listNode);

```

Each curve is defined by specifying a unique curve id (integer number), two or more node ids, and a curve step in length unit. There are three types of curves: two-node straight line, three-node arc, and four-or-more-node spline. All curves must then be assigned to the model by calling `solver.setCurves()`. In the following, curve1 and curve2 are two-node straight lines, and curve5 and curve6 are three arcs.

```

// define curves
List<cgiCurveCli> listCurve = new List<cgiCurveCli>();
cgiCurveCli curve1 = new cgiCurveCli();

```

```

curve1.setId(1);
curve1.setStep(1);
curve1.setNodes(1, 2);
listCurve.Add(curve1);
cgiCurveCli curve2 = new cgiCurveCli();
curve2.setId(2);
curve2.setStep(1);
curve2.setNodes(2, 3);
listCurve.Add(curve2);
//...
cgiCurveCli curve5 = new cgiCurveCli();
curve5.setId(5);
curve5.setStep(1);
curve5.addNode(5);
curve5.addNode(6);
curve5.addNode(7);
listCurve.Add(curve5);
cgiCurveCli curve6 = new cgiCurveCli();
curve6.setId(6);
curve6.setStep(1);
curve6.addNode(5);
curve6.addNode(8);
curve6.addNode(7);
listCurve.Add(curve6);
//...
solver.setCurves(listCurve);

```

Next, we will define edges to be used as boundary for regions.

```

// define edges for planeRegion1
cgiMeshEdgeCli planeRegion1_edge1 = new cgiMeshEdgeCli();
planeRegion1_edge1.addCurveId(1);
planeRegion1_edge1.addCurveId(2);
planeRegion1_edge1.addCurveId(3);
planeRegion1_edge1.addCurveId(4);

```

A plane region can also include internal points, holes, and trees.

```

// define internal points for planeRegion1
cgiMeshInternalPointCli planeRegion1_internalPoint1 = new cgiMeshInternalPointCli();
planeRegion1_internalPoint1.nodeId = 9;
planeRegion1_internalPoint1.regionId = 1;

```

```

planeRegion1_internalPoint1.step = 0;
planeRegion1_internalPoint1.radius = 0;

// define trees for planeRegion1
cgiMeshTreeCli planeRegion1_tree1 = new cgiMeshTreeCli();
planeRegion1_tree1.iId = 1;
planeRegion1_tree1.name = "t";
planeRegion1_tree1.addCurveId(7);
planeRegion1_tree1.addCurveId(8);

// define holes for planeRegion1
cgiMeshHoleCli planeRegion1_hole1 = new cgiMeshHoleCli();
planeRegion1_hole1.iId = 1;
planeRegion1_hole1.name = "f";
planeRegion1_hole1.addCurveId(5);
planeRegion1_hole1.addCurveId(6);

```

We can then define one or more regions. Each region must be added to the model by calling `addPlaneRegion()`, `addCylinderRegion()`, `addSphereRegion()`, `addRevRegion()`, or `addGeneralRegion()`.

```

cgiMeshPlaneRegionCli planeRegion1 = new cgiMeshPlaneRegionCli();
planeRegion1.iId = 1;
planeRegion1.name = "REGION 1";
planeRegion1.refValue = 0;

// set edges for planeRegion1
planeRegion1.setEdge(planeRegion1_edge1);

planeRegion1.addInternalPoint(planeRegion1_internalPoint1);

planeRegion1.addTree(planeRegion1_tree1);

planeRegion1.addHole(planeRegion1_hole1);

solver.addPlaneRegion(planeRegion1);

```

It might be a good idea to export the mesh model just defined to QuadMaker so you can visually inspect (or even solve) the mesh model in QuadMaker.

```

solver.exportToQuadMaker(); // export input mesh model to QuadMaker

```

Next, we can check the input, solve the mesh model like the following:

```

int ret = (int)cgiErrorEnum.ERROR_NONE;
string msg = "";
ret = solver.checkInputData(ref msg);
if (ret != (int)cgiErrorEnum.ERROR_NONE)
{
    Console.WriteLine("Invalid Input : " + msg);
    return;
};

ret = solver.solve(); // solve
if (ret != (int)cgiErrorEnum.ERROR_NONE)
{
    Console.WriteLine("Error encountered");
    return;
};

```

It might also be a good idea to export the generated mesh to Real3D so you can visually inspect the generated mesh.

```

solver.exportToReal3D(); // export generated mesh to a Real3D file

```

Finally, we can retrieve the generated nodes and elements (shells) like in the following:

```

List<cgiNodeCli> result_nodes = new List<cgiNodeCli>();
List<cgiShell4Cli> result_shells = new List<cgiShell4Cli>();
solver.getResults(ref result_nodes, ref result_shells);
for (int i = 0; i < result_nodes.Count; i++)
{
    var n = result_nodes[i];
    Console.WriteLine("node " + n.iId + ", " + n.x + ", " + n.y + ", " + n.z);
}

for (int i = 0; i < result_shells.Count; i++)
{
    var s = result_shells[i];
    Console.WriteLine("shell " + s.iId + ", " + s.node1 + ", " + s.node2 + ", " + s.node3 + ", " + s.node4);
}

```

The following (Figure 1.2) is the generated mesh viewed in QuadMaker

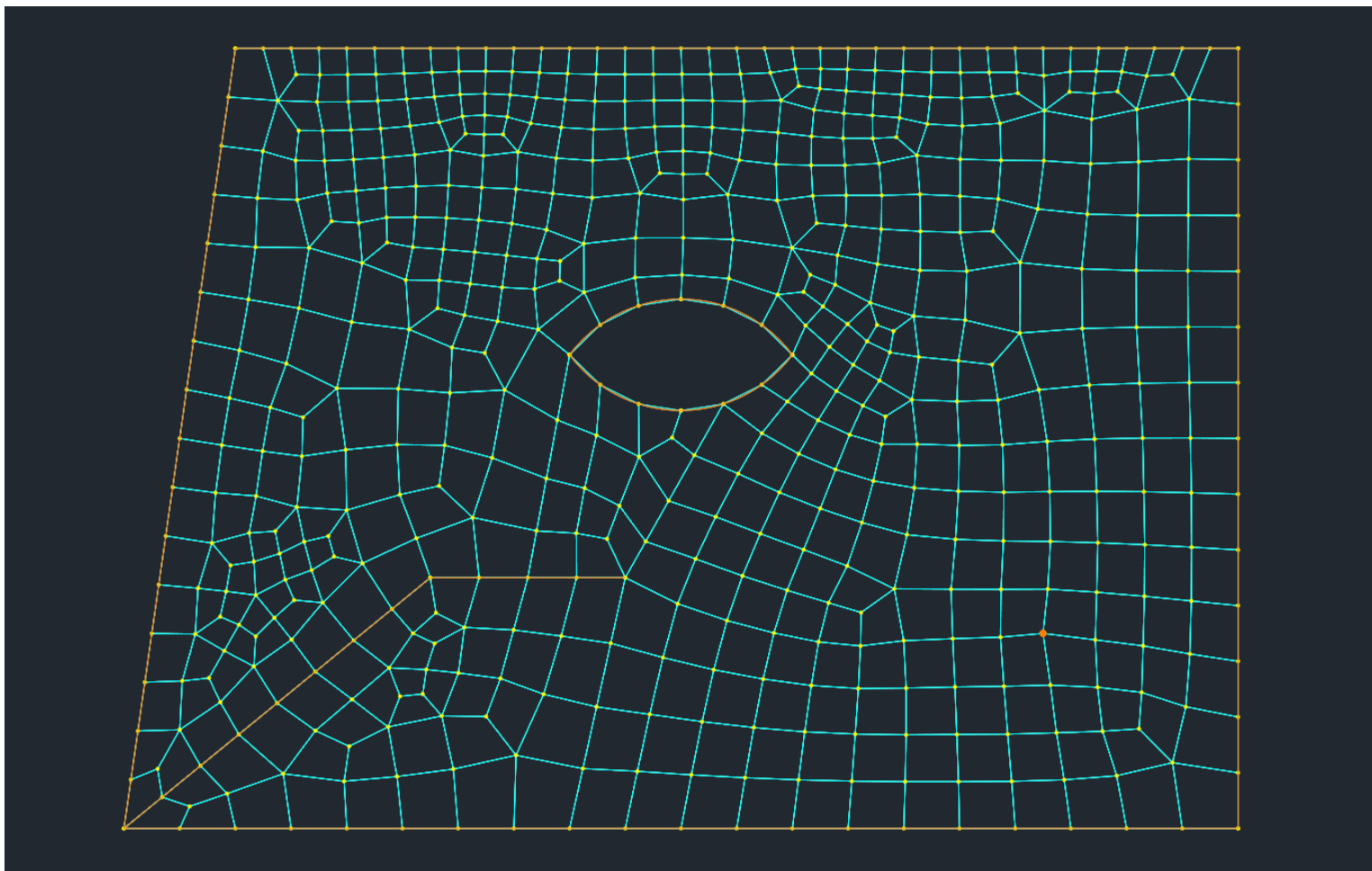


Figure 1.2

Chapter 2 - C++ Interface

The C++ library contains two 64-bit native Windows DLLs `cgiQuadSdk.dll` and `QuadSurfaceDll.dll`. The interface includes the following header files: `_cgiDefines.h`, `_cgiIStructure.h`, `_cgiPlatform.h` and `cgiQuadSdk.h`. It also includes a `cgiQuadSdk.lib` for dynamically linking to your projects. QuadSdk supports Visual Studio 2015 through 2026. It is important to point out that “Multi-threaded DLL” and “Multi-threaded Debug DLL” runtime library must be used for the release and debug build mode in your application respectively. The corresponding release or debug version of `cgiQuadSdk.dll` and `QuadSurfaceDll.dll` must be copied to the output folders of your application. *For distribution, you may need to install Visual C++ 2015~2026 x64 redistributables as `cgiQuadSdk.dll` link dynamically with the visual C++ runtime library.*

The `_cgiDefines.h` defines all input and output data structures. `_cgiIStructure.h` is the one and only interface to set input, perform meshing, and retrieve output. The best way to learn how you use these data structures and interfaces is to study the examples included in the C++ console application project `cgiQuadSdkTest`. These examples are taken from the User’s Manual QuadMaker, which is an interactive quadrilateral meshing software by CGI.

The following lists all the interface functions exposed by `cgiIStructure`:

```
// function callbacks to notify the clients
typedef void (*fnSTATUSMSG)(LPCTSTR sz);

namespace cgiQuadSdkNamespace
{
    struct CGIQUADSDK_API cgiIStructure
    {
        virtual void setStatusMessageFunction(fnSTATUSMSG fnStatusMsg) = 0;

        virtual void setProjPath(const TCHAR projPath[]) = 0;
        virtual void getProjPath(TCHAR projPath[])const = 0;

        // LENGTH=ft;
        virtual void setStandardEnglishUnits() = 0;
        // LENGTH=m;
        virtual void setStandardMetricUnits() = 0;
        // lb; in; rad
        virtual void setConsistentEnglishUnits() = 0;
        // N; m; rad
        virtual void setConsistentMetricUnits() = 0;

        virtual void setNodes(const std::vector<cgiNode>& vNd) = 0;
        virtual void getNodes(std::vector<cgiNode>& vNd)const = 0;
        virtual void setCurves(const std::vector<cgiCurve>& vBm) = 0;
        virtual void getCurves(std::vector<cgiCurve>& vBm)const = 0;

        virtual void getSingleNode(cgiNode& node, int nId)const = 0;
    };
}
```

```

virtual void getSingleCurve(cgiCurve& beam, int nId)const = 0;

virtual void addPlaneRegion(const cgiMeshPlaneRegion& region) = 0;
virtual void addCylinderRegion(const cgiMeshCylinderRegion& region) = 0;
virtual void addSphereRegion(const cgiMeshSphereRegion& region) = 0;
virtual void addRevRegion(const cgiMeshRevRegion& region) = 0;
virtual void addGeneralRegion(const cgiMeshGeneralRegion& region) = 0;

virtual void getPlaneRegion(cgiMeshPlaneRegion& region, int regionId)const = 0;
virtual void getCylinderRegion(cgiMeshCylinderRegion& region, int regionId)const = 0;
virtual void getSphereRegion(cgiMeshSphereRegion& region, int regionId)const = 0;
virtual void getRevRegion(cgiMeshRevRegion& region, int regionId)const = 0;
virtual void getGeneralRegion(cgiMeshGeneralRegion& region, int regionId)const = 0;

virtual void getPlaneRegions(std::vector<cgiMeshPlaneRegion>& vRegion)const = 0;
virtual void getCylinderRegions(std::vector<cgiMeshCylinderRegion>& vRegion)const = 0;
virtual void getSphereRegions(std::vector<cgiMeshSphereRegion>& vRegion)const = 0;
virtual void getRevRegions(std::vector<cgiMeshRevRegion>& vRegion)const = 0;
virtual void getGeneralRegions(std::vector<cgiMeshGeneralRegion>& vRegion)const = 0;

virtual bool exportToQuadMaker()const = 0;
virtual bool exportToReal3D()const = 0;
virtual int checkInputData(TCHAR szMsg[MAX_LINE_SIZE], int bufferLength)const = 0;
virtual void setSolverParameters(int max_subregions, int max_curves_per_tree, int max_points_per_tree, int max_nodel_3d) = 0;
virtual void getSolverParameters(int& max_subregions, int& max_curves_per_tree, int& max_points_per_tree, int& max_nodel_3d) = 0;
virtual int solve() = 0;
virtual void alignShell4LocalZWithReferencePoint(const cgiPoint& referencePoint) = 0;
virtual void getResults(std::vector<cgiNode>& vNd, std::vector<cgiShell4>& vShell4)const = 0;

virtual void clearNodes() = 0;
virtual void clearCurves() = 0;
virtual void clearRegions() = 0;
virtual void clear() = 0;

};

CGIQUADSDK_API cgiIStructure* CreateStructure();

}

```

Here is a C++ code example to set up, solve, and retrieve results for the same mesh model used in the .NET interface section. For brevity, we are not going to repeat the detailed explanations here.

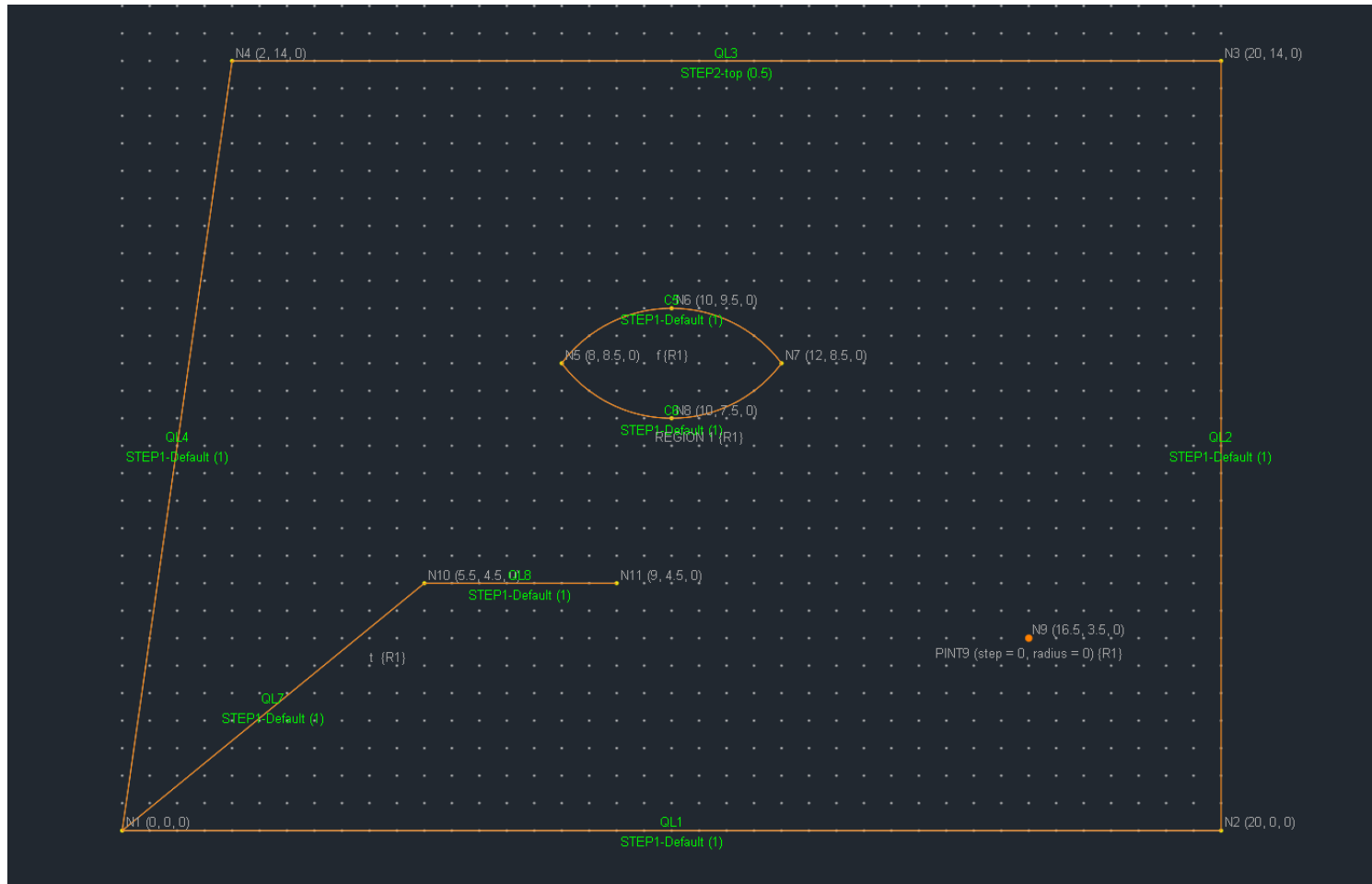


Figure 2.1

```

#include "_cgiIStructure.h"
#include <iostream>
#include <iomanip>
#include <vector>

using namespace std;
using namespace cgiQuadSdkNamespace;

#ifdef _UNICODE
#define COUT wcout
#else
#define COUT cout
#endif

static void StatusMsg(LPCTSTR sz)
{
    COUT << sz << endl;
}

void verify_demo()
{
    COUT << _T("-----") << endl;
    COUT << _T("----- Verifying Demo Cylinder Roof -----") << endl;
    COUT << _T("-----") << endl;

    // create a structural model
    cgiIStructure* pStructure = CreateStructure();

    // message functions, can be set null in which case no messages will be printed during solution
    pStructure->setStatusMessageFunction(StatusMsg);

    pStructure->setProjPath(_T("c:\\CGInc\\QuadSdk\\SampleCode\\Models\\Native\\demo_")); // folder path + file name without extension

    pStructure->setStandardEnglishUnits(); // length unit = ft
    //pStructure->setStandardMetricUnits(); // length unit = m

    // define nodes
    vector<cgiNode> nodes;
    cgiNode node1;
    node1.setID(1);
    node1.setCoordinates(0, 0, 0);
    nodes.push_back(node1);
    cgiNode node2;
    node2.setID(2);
    node2.setCoordinates(20, 0, 0);
    nodes.push_back(node2);
    cgiNode node3;
    node3.setID(3);
    node3.setCoordinates(20, 14, 0);
}

```

```

nodes.push_back(node3);
cgiNode node4;
node4.setId(4);
node4.setCoordinates(2, 14, 0);
nodes.push_back(node4);
cgiNode node5;
node5.setId(5);
node5.setCoordinates(8, 8.5, 0);
nodes.push_back(node5);
cgiNode node6;
node6.setId(6);
node6.setCoordinates(10, 9.5, 0);
nodes.push_back(node6);
cgiNode node7;
node7.setId(7);
node7.setCoordinates(12, 8.5, 0);
nodes.push_back(node7);
cgiNode node8;
node8.setId(8);
node8.setCoordinates(10, 7.5, 0);
nodes.push_back(node8);
cgiNode node9;
node9.setId(9);
node9.setCoordinates(16.5, 3.5, 0);
nodes.push_back(node9);
cgiNode node10;
node10.setId(10);
node10.setCoordinates(5.5, 4.5, 0);
nodes.push_back(node10);
cgiNode node11;
node11.setId(11);
node11.setCoordinates(9, 4.5, 0);
nodes.push_back(node11);
pStructure->setNodes(nodes);

// define curves
vector<cgiCurve> curves;
cgiCurve curve1;
curve1.setId(1);
curve1.setStep(1);
curve1.setNodes(1, 2);
curves.push_back(curve1);
cgiCurve curve2;
curve2.setId(2);
curve2.setStep(1);
curve2.setNodes(2, 3);
curves.push_back(curve2);
cgiCurve curve3;
curve3.setId(3);
curve3.setStep(0.5);
curve3.setNodes(3, 4);
curves.push_back(curve3);

```

```

cgiCurve curve4;
curve4.setId(4);
curve4.setStep(1);
curve4.setNodes(4, 1);
curves.push_back(curve4);
cgiCurve curve5;
curve5.setId(5);
curve5.setStep(1);
curve5.addNode(5);
curve5.addNode(6);
curve5.addNode(7);
curves.push_back(curve5);
cgiCurve curve6;
curve6.setId(6);
curve6.setStep(1);
curve6.addNode(5);
curve6.addNode(8);
curve6.addNode(7);
curves.push_back(curve6);
cgiCurve curve7;
curve7.setId(7);
curve7.setStep(1);
curve7.setNodes(1, 10);
curves.push_back(curve7);
cgiCurve curve8;
curve8.setId(8);
curve8.setStep(1);
curve8.setNodes(10, 11);
curves.push_back(curve8);
pStructure->setCurves(curves);

// define edges for planeRegion1
cgiMeshEdge planeRegion1_edge1;
planeRegion1_edge1.addCurveId(1);
planeRegion1_edge1.addCurveId(2);
planeRegion1_edge1.addCurveId(3);
planeRegion1_edge1.addCurveId(4);

// define internal points for planeRegion1
cgiMeshInternalPoint planeRegion1_internalPoint1;
planeRegion1_internalPoint1.nodeId = 9;
planeRegion1_internalPoint1.regionId = 1;
planeRegion1_internalPoint1.step = 0;
planeRegion1_internalPoint1.radius = 0;

// define trees for planeRegion1
cgiMeshTree planeRegion1_tree1;
planeRegion1_tree1.iId = 1;
_tcsncpy(planeRegion1_tree1.szName, _T("t"), LABEL_SIZE - 1);
planeRegion1_tree1.addCurveId(7);
planeRegion1_tree1.addCurveId(8);

```

```

// define holes for planeRegion1
cgiMeshHole planeRegion1_hole1;
planeRegion1_hole1.iId = 1;
_tcsncpy(planeRegion1_hole1.szName, _T("f"), LABEL_SIZE - 1);
planeRegion1_hole1.addCurveId(5);
planeRegion1_hole1.addCurveId(6);

cgiMeshPlaneRegion planeRegion1;
planeRegion1.iId = 1;
_tcsncpy(planeRegion1.szName, _T("REGION 1"), LABEL_SIZE - 1);
planeRegion1.refValue = 0;

// set edges for planeRegion1
planeRegion1.setEdge(planeRegion1_edge1);

planeRegion1.addInternalPoint(planeRegion1_internalPoint1);

planeRegion1.addTree(planeRegion1_tree1);

planeRegion1.addHole(planeRegion1_hole1);

pStructure->addPlaneRegion(planeRegion1);

pStructure->exportToQuadMaker(); // export input mesh model to QuadMaker

int ret = ERROR_NONE;

TCHAR szMsg[MAX_LINE_SIZE];
ret = pStructure->checkInputData(szMsg, MAX_LINE_SIZE - 1);
if (ret != ERROR_NONE) {
    COUT << _T("Invalid Input: ") << szMsg << endl;
    return;
}

ret = pStructure->solve(); // solve

if (ret != ERROR_NONE) {
    COUT << _T("Error encountered");
    return;
};

pStructure->exportToReal3D(); // export generated mesh to a Real3D file

std::vector<cgiNode> resultNodes;
std::vector<cgiShell4> resultShells;
pStructure->getResults(resultNodes, resultShells);

for (const auto& node : resultNodes) {
    cout << "node " << node.iId << ", " << node.pt.xyz[X] << ", " << node.pt.xyz[Y] << ", " << node.pt.xyz[Z] << endl;
}

for (const auto& shell : resultShells) {

```

```
    cout << "shell " << shell.iId << ", " << shell.nodeIds[0] << ", " << shell.nodeIds[1] << ", "
        << shell.nodeIds[2] << ", " << shell.nodeIds[3] << endl;
}
}
```