Manual Geometry Algorithms v2.0

Contact information:

- **Mail:** jobberwockysoftware@gmail.com
- **Forum:** https://forum.unity.com/threads/geometry-algorithms.409854/
- **Facebook:** https://www.facebook.com/JobberwockySoftware/
- **Twitter:** https://twitter.com/JobberwockySoft

Changelog

- Version 1.x => 2.0
    - All references to the old asset name have been removed, this means all namespaces have been updated: all namespaces with **TriangulationForUnity** in it are now changed to use **GeometryAlgorithms** instead.
    - All Generator objects have been removed and changed to API objects. Thus, e.g. instead of calling **new HullGenerator()**, you now use **new HullAPI().**
    - The new API objects all inherit from the abstract **ThreadingAPI** class. Now, every method can be used with threading by using its async counterpart. This means that if you want to run the **Triangulation2D** method in a separate thread you have to call the **Triangulation2DAsync** method.
    - Input parameters are set by using an **IParameters** object. Each method only uses this object to setup its parameters.
    - Code that was on the MiConvexhull and Triangle.Net libraries is placed back in their respective dlls. Previously, I tried to refactor both libraries into one single library, but it was very difficult to update this when for example a new version of one of the libraries came out.

Methods & Objects

This part of the manual gives a brief overview of the various methods and objects that exist in Geometry Algorithms. You can also have a look at various code examples that are available. These can be found in the "Example" folder and should be self-explanatory.

API objects

The API objects contain the methods to perform the geometrical operations. There are three different API objects available:

- **TriangulationAPI** contains methods to generate 2D and 3D triangulations
- **HullAPI** contains methods to generate 2D and 3D hulls
- **VoronoiAPI** contains methods to generate 2D and 3D Voronois.

Geometry

The Geometry object stores the results of the methods in the API objects. This object is similar to the Unity Mesh object, but it can be extended and inherited by other objects to store additional information. In addition, it is possible to create a Unity mesh from this object.

Async

Threading is supported by all methods. You have to call the async version of a certain method. For example, to run the **Triangulation2D** method in a different thread, you have to use the **Triangulation2DAsync** method. The async methods do not return an object, but you have to provide a callback function as an input parameter to process the results. In general, I would only recommend using threading if you are going to do very time-consuming calculations that will freeze the main thread, because threading is not supported for every platform (e.g. WebGL), and performance-wise the non-async methods are better with smaller calculations. In the "Example" folder, there is a specific example to demonstrate the async methods.

Triangulation

Triangulations can be generated by creating a **TriangulationAPI** object and call the **Triangulation2D** or **Triangulation3D** method. If you want to triangulate 2.5D data then you should use the **Triangulate2D** method since it will take into account the height value. The triangulation methods return a Unity Mesh, if you do not want a Unity object then you call the methods **Triangulation2DRaw** or **Triangulation3DRaw,** which return a **Geometry** object.

Triangulation2D

Code examples of the **Triangulation2D** method can be found in the **ExampleGeometry2D** and **ExampleInteractiveVoronoi2D** scripts in the "Example" folder. Both convex and concave triangulations (with holes) are supported and shown in the examples.

The input parameters are provided by declaring a **Triangulation2DParameters** object and setting its parameters. The following parameters are available:

- **Points,** is an array of Vector3 points that define regular points of a shape. Array can be null if the **Boundary** parameter is defined.
- **Boundary**, is an array of Vector3 points that defines the boundary of a shape. These points need to be in the correct order. This parameter allows you to create concave shapes. Value can be null if the **Points** parameter is defined.
- **Holes**, is an array containing arrays of Vector3 points where each array represents a different hole. Again, for this parameter the points need to be in the correct to represent a hole. Value can be null.
- **Delaunay**, is an boolean that determines whether the triangulation should be strictly Delaunay. It is possible that if it is set to true that additional points are added during the triangulation to meet the Delaunay criteria.
- **Side**, is an enum that determines which side should be triangulated. There are three options: Front, Back, Double.

Triangulation3D

Code examples of the **Triangulation3D** method can be found in the **ExampleGeometry3D** script in the "Example" folder. Only convex triangulations are supported.

The input parameters are provided by declaring a **Triangulation3DParameters** object and setting its parameters. The following parameters are available:

- **Points,** is an array of Vector3 points which are used to create the 3D triangulation
- **BoundaryOnly,** is a boolean that determines whether only the outside of the 3D triangulation should be returned. If it is false then also each triangle of every tetrahydron is returned.
- **Side**, is an enum that determines which side should be triangulated. There are three options: Front, Back, Double.

Hull

Hulls can be generated by creating a **HullAPI** object and the **Hull2D** or **Hull3D** method. The hull methods return a Unity Mesh, if you do not want a Unity object then you call the methods **Hull2DRaw** or **Hull3DRaw,** which return a **Geometry** object. For the 2D case, hulls are defined as a line, and for the 3D case, hulls are defined as a set of triangles.

Hull2D

Code examples of the **Hull2D** method can be found in the **ExampleGeometry2D** script in the "Example" folder. Both convex and concave hulls are supported. Make sure that you have defined sufficient points if you want to find a good-fitting concave hull.

The input parameters are provided by declaring a **Hull2DParameters** object and setting its parameters. The following parameters are available:

- **Points,** is an array of Vector3 points for which a hull/boundary is calculated.
- **Concavity**, is the maximum length that is allowed between each point. Smaller values will result in a more concave hull, but also increases the calculation time.

Hull3D

Code examples of the **Hull3D** method can be found in the **ExampleGeometry3D** script in the "Example" folder. Only convex hulls are supported.

The input parameters are provided by declaring a **Hull3DParameters** object and setting its parameters. The following parameters are available:

- **Points,** is an array of Vector3 points used to determine the 3D hull.

Voronoi diagram

Voronoi diagrams can be generated by creating a **VoronoiAPI** object and use the **Voronoi2D** or **Voronoi3D** method. The Voronoi methods return a Unity Mesh, if you do not want a Unity object then you call the methods **Voronoi2DRaw** or **Voronoi3DRaw,** which return a **Geometry** object.

Voronoi2D

Code examples of the **Voronoi2D** method can be found in the **ExampleGeometry2D** and **ExampleInteractiveVoronoi2D** scripts in the "Example" folder.

The input parameters are provided by declaring a **Voronoi2DParameters** object and setting its parameters. The following parameters are available:

- **Points,** is an array of Vector3 points used to calculate the Voronoi diagram.

Voronoi3D

Code examples of the **Voronoi3D** method can be found in the **ExampleGeometry3D** script in the "Example" folder.

The input parameters are provided by declaring a **Voronoi3DParameters** object and setting its parameters. The following parameters are available:

- **Points,** is an array of Vector3 points used to calculate the Voronoi diagram.