DOCUMENTATION FOR TETRAHEDRALIZER VERSION 1.0.0

Useful Links:

Tetrahedralizer on GitHub: https://github.com/hanzemeng/Tetrahedralizer
Tetrahedralizer on Unity Asset Store: https://assetstore.unity.com/packages/slug/306196
Tetrahedralizer's online documentation: https://docs.google.com/document/d/1TmDsCK4SiXGq7Y7xVC6l-rAW5G5WxgNKjIlzR9Algpc/edit?usp=sharing

Purpose:

Tetrahedralizer converts triangle meshes into tetrahedral meshes. As the name suggests, a tetrahedral mesh represents a 3D model using tetrahedrons, which are more suitable for geometric computation than using triangles. Tetrahedralizer is able to convert arbitrarily complex triangle meshes that may include intersecting triangles, degenerate triangles, and non-manifold edges.

Tetrahedralizer is a rather low-level package. The users should have prior experience working with meshes to make the best use of Tetrahedralizer.
I do my best to explain everything Tetrahedralizer has to offer, but some parts may still be confusing.

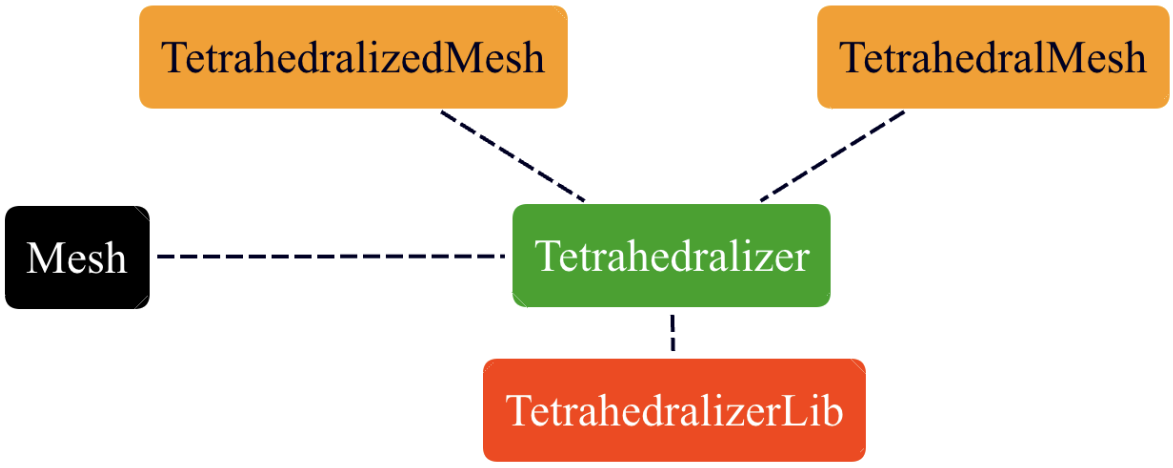
Requirements:

The operating system is Windows (64-bit) or macOS (64-bit or silicon). If one wishes to use this package in another operating system, read this documentation first and focus on The TetrahedralizerLib section.

The CPU must be in little-endian (most CPU is little-endian by default).
The CPU must conform with IEEE 754 (if a CPU is manufactured in the 21st century, it probably is).

Roadmap:

The picture below describes the relationship between the major classes in this package.



The black class, Mesh, is from Unity Engine. It is the input to Tetrahedralizer.

The oranges classes are scriptable objects. They store results computed by Tetrahedralizer.

The green class, Tetrahedralizer, is the core of this package. It implements the algorithm used to convert Mesh into TetrahedralizedMesh and TetrahedralizedMesh to TetrahedralMesh.

The red “class”, TetrahedralizerLib, is a dynamic library written in C++. It is used by Tetrahedralizer.

The expected workflow is that the user first create an instance of Tetrahedralizer,
then supply the mesh to convert to the tetrahedralizer and it will output an instance of TetrahedralizedMesh,
then supply the tetrahedralizedMesh to the tetrahedralizer and it will output an instance of TetrahedralMesh.
The tetrahedralMesh now contains the converted tetrahedral mesh. The rest is up to the user.

The rest of the document goes over the classes in more detail.

The Mesh Class:

Please refer to Unity Documentation.

The Tetrahedralized Mesh Class:

This class is a scriptable object that stores information needed to tetrahedralize a mesh.
The user may wish to skip this section if they are not planning to modify the package itself.

Below describes the fields of this class:

```
public Mesh originalMesh;

public List<ListInt32> originalVerticesMappings;
public List<NineInt32> newVerticesMappings;
public List<Int32> tetrahedrons;

/*
originalMesh is a reference to the original mesh that is being
tetrahedralized.


originalVerticesMappings stores indices of vertices from originalMesh
that share the same position.
Note that ListInt32 is a wrapper class for List<Int32>.

For example, say originalMesh's 2,39,444 vertices are in the same
position, then originalVerticesMappings[2] contains a ListInt32
object with its list containing {2,39,444}.

We can think of originalVerticesMappings as a map that only keeps the
unique positions in originalMesh.vertices.


newVerticesMappings stores new vertices generated by the algorithm.
The new vertices are represented using indices from
originalVerticesMappings (not from originalMesh).
Note that NineInt32 is a struct that contains nine Int32.

If 0xFFFFFFFF == newVerticesMappings[i][5], this means the ith new
vertex is the intersection of a line and a plane.
The line is defined as
newVerticesMappings[i][0],newVerticesMappings[i][1].
The plane is defined as
newVerticesMappings[i][2],newVerticesMappings[i][3],newVerticesMappin
gs[i][4].

If 0xFFFFFFFF != newVerticesMappings[i][5], this means the ith new
vertex is the intersection of three planes.
The first plane is defined as
```

```
newVerticesMappings[i][0],newVerticesMappings[i][1],newVerticesMappings[i][2].
The second plane is defined using the same pattern but starts at 3.
The third plane is defined using the same pattern but starts at 6.


tetrahedrons stores the tetrahedrons that represent the tetrahedral mesh.
For example, let 0==i%4, then
tetrahedrons[i+0],tetrahedrons[i+1],
tetrahedrons[i+2],tetrahedrons[i+3]
are the 4 indices that represent a tetrahedron.

If tetrahedrons[i] < originalVerticesMappings.Count, then
tetrahedrons[i] corresponds to one of the vertices from originalMesh.

If tetrahedrons[i] >= originalVerticesMappings.Count, then
tetrahedrons[i] corresponds to
newVerticesMappings[tetrahedrons[i]-originalVerticesMappings.Count].

The tetrahedrons are oriented using the left-hand rule.
That is, if one curls their left hand around tetrahedrons[i+0],
tetrahedrons[i+1], and tetrahedrons[i+2], then their thumb points
toward tetrahedrons[i+3].
*/
```

The Tetrahedral Mesh Class:

This class is a scriptable object that stores the tetrahedral mesh.
The tetrahedral mesh is stored in one of two ways:
Way one is storing only the shape information; we know the position of every vertex, but nothing else (no uv, color, etc.).
Way two is storing position as well as every type of vertex data that the original mesh has.

A tetrahedralMesh behaves very differently depending on which storing scheme is used, so please pay close attention to what we are about to read.
Below describes the fields of this class:

```
public List<Vector3> vertices;
```

```
public List<Int32> tetrahedrons;

public List<Int32> vertexAttributeDescriptors;

public List<Color32> colors;
public List<Vector4> uvs0; // note that uvs are stored in Vector4
public List<Vector4> uvs1;
public List<Vector4> uvs2;
public List<Vector4> uvs3;
public List<Vector4> uvs4;
public List<Vector4> uvs5;
public List<Vector4> uvs6;
public List<Vector4> uvs7;

public List<Int32> facetsSubmeshes;


/*
When storing scheme one is used, vertices and tetrahedrons are
populated with data, other fields are empty.

vertices stores the position of each vertex.

Note that a regular mesh often has multiple vertices at the same
position (for accurate normal and tangent), this is not the case for
vertices.
vertices may still contain duplicate values due to floating-point
rounding, but we don't intentionally create multiple copies of the
same vertex.

tetrahedrons stores indices that represent tetrahedrons.
For example, let 0==i%4, then
tetrahedrons[i+0],tetrahedrons[i+1],
tetrahedrons[i+2],tetrahedrons[i+3]
are the 4 indices that represent a tetrahedron.

The tetrahedrons are oriented using the left-hand rule.
That is, if one curls their left hand around tetrahedrons[i+0],
tetrahedrons[i+1], and tetrahedrons[i+2], then their thumb points
```

toward ***tetrahedrons***[i+3].
Note that it is possible to have degenerate tetrahedrons (4 vertices on the same plane).

When storing scheme two is used, ***tetrahedrons*** is empty, other fields are populated with data.

Because a tetrahedron has 4 triangle faces,
every 12 vertices correspond to a tetrahedron.
For example, let 0==i%12, then
vertices[i+0],***vertices***[i+1],***vertices***[i+2]
is the position of the first face of the i/12 tetrahedron.
Suppose the original mesh has uv1, then
uvs1[i+3],***uvs1***[i+4],***uvs1***[i+5]
is the uv1 of the second face of the i/12 tetrahedron.

Pop quiz:
how to access the color of the fourth face of the i/12 tetrahedron?
highlight the line below to check the answer.

(If highlighting doesn't work, try copying the line and pasting it somewhere.)

Every kind of vertex data from the original mesh is generated except boneWeights.

Note that each triangle face is pointing out. That is, if one curls their left hand around ***vertices***[i+0],***vertices***[i+1],***vertices***[i+2], then their thumb points away from the center of the tetrahedron.

Also note that for every 0==i%12:
vertices[i+0] == ***vertices***[i+4] == ***vertices***[i+6],
vertices[i+1] == ***vertices***[i+3] == ***vertices***[i+10],
vertices[i+5] == ***vertices***[i+8] == ***vertices***[i+11],
vertices[i+2] == ***vertices***[i+7] == ***vertices***[i+9].
Basically, every vertex in a tetrahedron corresponds to 3 vertices. The duplicate vertices are needed for other vertex data such as

normal, tangent, and uv.

vertexAttributeDescriptors describes each of the vertex attributes. Each element is supposed to be of type VertexAttributeDescriptor, but Unity doesn't know how to serialize it. Therefore each vertexAttributeDescriptor is converted to Int32 and can be converted back when needed.

facetsSubmeshes describes which submesh a given triangle face belongs to.
For example, let 0==i%4, then ***facetsSubmeshes***[i+0]
is the submesh for the first face of the i/4 tetrahedron.

Suppose the original mesh has ***n*** submesh, 0<=***facetsSubmeshes***[i]<=n for all i. If n==***facetsSubmeshes***[i], then the ith face is likely an interior face and is not part of the original mesh. Otherwise, ***facetsSubmeshes***[i] is the same as the original mesh's triangle's submesh. (I have no idea how to phrase the last sentence better.)

Regardless of the storing scheme, one can get a reference of a vertex data by using the dot operator (for example, tetrahedralMesh.***vertices***).

A more costly way is to duplicate the vertex data and store the result into a user-provided list. Here is the syntax:

```
tetrahedralMesh.GetVertices(IList<Vector3> target);
tetrahedralMesh.GetColors(IList<Color32> target);
tetrahedralMesh.GetUVs(Int32 channel, IList<Vector4> target);
```

Checkout TetrahedralMeshDrawer.cs for examples on how to use a tetrahedral mesh.
*/

The Tetrahedralizer Class:

This class converts Mesh into TetrahedralizedMesh and TetrahedralizedMesh to TetrahedralMesh.

A tetrahedralizer can be created by:

```
Tetrahedralizer tetrahedralizer = new Tetrahedralizer();
```

To convert a mesh into a tetrahedralizedMesh:

```
tetrahedralizer.MeshToTetrahedralizedMesh(Mesh mesh,
TetrahedralizedMesh tetrahedralizedMesh);

// mesh is the input.
// tetrahedralizedMesh is cleared and has the output stored in it.
```

To convert a tetrahedralizedMesh into a tetrahedralMesh:

```
tetrahedralizer.TetrahedralizedMeshToTetrahedralMesh(
TetrahedralizedMesh tetrahedralizedMesh,
TetrahedralMesh tetrahedralMesh);

// tetrahedralizedMesh is the input.
// tetrahedralMesh is cleared and has the output stored in it.

// contact the author if one has a shorter name for this function.
```

A tetrahedralizer has settings that can be adjusted:

```
tetrahedralizer.SetSettings(Tetrahedralizer.Settings settings);

/*
where settings is defined as:
settings = new Tetrahedralizer.Settings(bool remapVertexData, double
degenerateTetrahedronRatio);

If remapVertexData is true, then the output tetrahedralMesh will use
storing scheme two, which contains every kind of vertex data from the
original mesh.
Otherwise, the output tetrahedralMesh will use storing scheme one,
which only has the position for every vertex.
```

```
degenerateTetrahedronRatio is the ratio to remove tetrahedrons that
are too small.
Let the average volume of each generated tetrahedron be v_a. Then, if
a tetrahedron's volume is less than v_a * degenerateTetrahedronRatio,
that tetrahedron is removed.
Set degenerateTetrahedronRatio to 0 if one wishes to keep every
generated tetrahedron.
*/
```

The TetrahedralizerLib:

TetrahedralizerLib is a dynamic library written in C++. Two versions of the library are included in the package.
One version supports macOS for both Intel 64-bit and Apple silicon chips.
The other supports Windows for Intel 64-bit chips.

The source code is provided on GitHub. If one wish to compile the source code, here are the requirements:

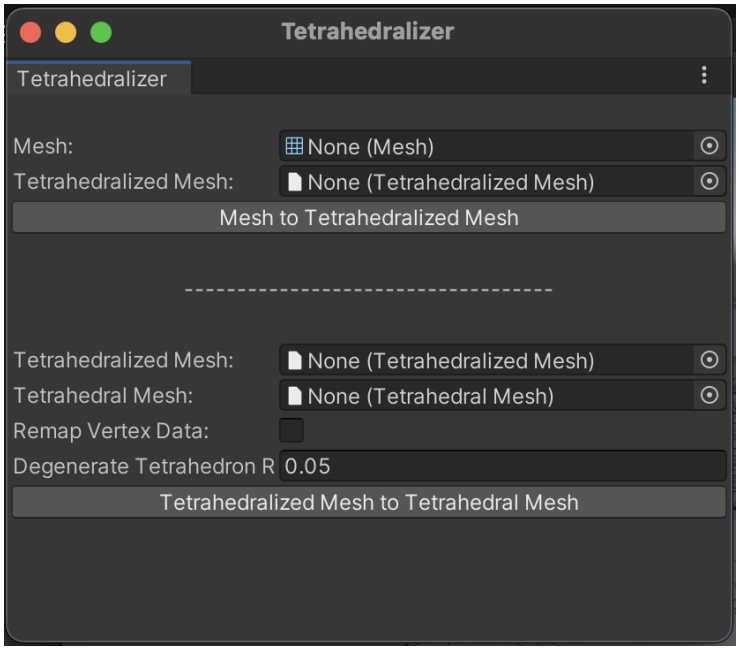
- Use C++20.
- Use -ffp-model=strict, -ffp-contract=off, -frounding-math.
- Optionally use -O2.

To use the new library, open Scripts/Tetrahedralizer.cs and assign TETRAHEDRALIZER_LIBRARY_NAME to the name of the new library.

Unity Editor Modification:

The Unity Editor has been updated with the intention of simplifying development.
All editor scripts are in Scripts/Editor/. The users are welcome to edit them.

In the menu bar, select Window/Tetrahedralizer to open a window that looks like:



The top part is about converting Mesh into TetrahedralizedMesh and
the bottom part is about converting TetrahedralizedMesh to TetrahedralMesh.

In the project window, right click and select
Create/ScriptableObjects/Tetrahedralizer/Tetrahedralized_Mesh to create a Tetrahedralized Mesh.
A Tetrahedral Mesh is created in the same way.

Limitations:

- Skinned meshes need to be converted to regular meshes before processing; the output will also be regular meshes.
- The tetrahedral mesh may contain many slivers (flat or almost flat tetrahedrons). One way to reduce them is to run a mesh refinement algorithm, which currently is not implemented.
- The shape of the converted tetrahedral mesh should closely match the shape of the original triangle mesh, but some parts may be missing or having excess. I have yet to find a better algorithm.
- The current algorithm for generating vertex data is not great. For example, the uv coordinates usually get messed up. I have yet to find a better algorithm.

Miscellaneous:

- Tetrahedralizer's cover image and other art assets are designed and created by artist michuudo. Please check him out on X and Bluesky.

- Thank you for using Tetrahedralizer and reading its documentation. Consider reviewing it on the Unity Asset Store.

Author Information:

Email: hanzemeng2001518@gmail.com
Unity Asset Store: https://assetstore.unity.com/publishers/91066
LinkedIn: https://www.linkedin.com/in/han-zemeng-a227b3147/

References:

- Diazzi, L., & Attene, M. (2021). Convex Polyhedral Meshing for Robust Solid Modeling. ACM Transactions on Graphics (SIGGRAPH Asia 2021), 40(6).
- M. Attene. Indirect Predicates for Geometric Constructions. In Elsevier Computer-Aided Design (2020).
- Alec Jacobson, Ladislav Kavan, Olga Sorkine-Hornung. Robust Inside-Outside Segmentation using Generalized Winding Numbers. ACM Transaction on Graphics 32(4) [Proceedings of SIGGRAPH], 2013.
- Efficient Approximate Energy Minimization via Graph Cuts. Y. Boykov, O. Veksler, R.Zabih. IEEE TPAMI, 20(12):1222-1239, Nov 2001.
- What Energy Functions can be Minimized via Graph Cuts? V. Kolmogorov, R.Zabih. IEEE TPAMI, 26(2): 147-159, Feb 2004.
- An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision. Y. Boykov, V. Kolmogorov. IEEE TPAMI, 26(9):1124-1137, Sep 2004.
- Many pages from Microsoft Learn, Unity Documentation, and Unity Forum are referenced.