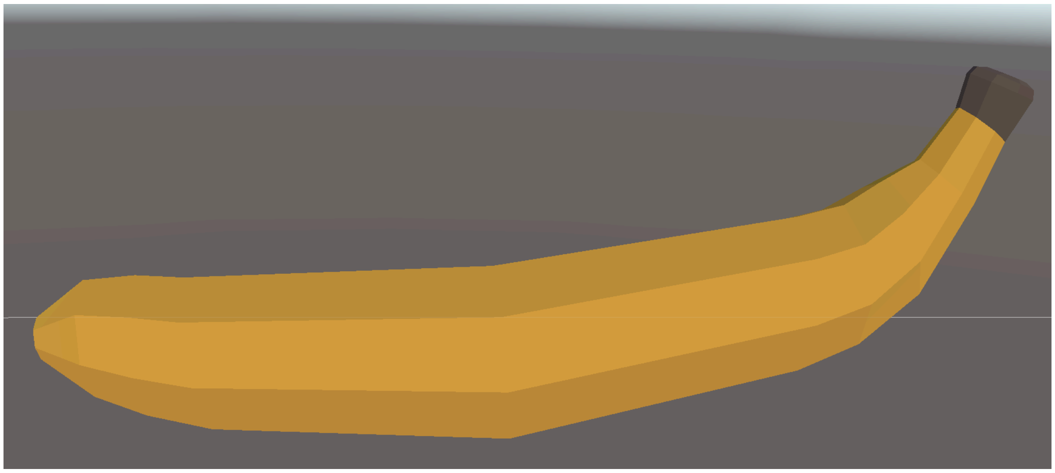
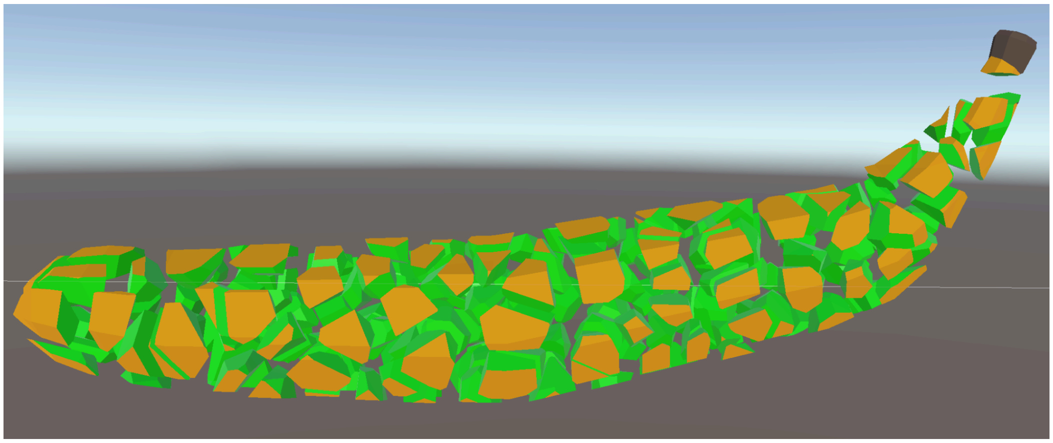
Useful Links:

Mesh Demolisher on GitHub: https://github.com/hanzemeng/MeshDemolisher
Mesh Demolisher on Unity Asset Store: https://assetstore.unity.com/packages/slug/288927
Mesh Demolisher's online documentation: Mesh Demolisher Documentation

Purpose:

Mesh Demolisher demolishes meshes into smaller pieces. Below is an example that demonstrates Mesh Demolisher's feature:





However, Mesh Demolisher has many requirements on the input mesh, so please read the next section carefully.

The Mesh Demolisher Class:

The MeshDemolisher class is used to demolish meshes. It can be created by:

```
MeshDemolisher meshDemolisher = new MeshDemolisher();
```

To demolish a mesh, call the following function:

```
meshDemolisher.Demolish(targetGameObject, breakPoints, interiorMaterial);

// targetGameObject is of type GameObject. targetGameObject is assumed to
// have a MeshFilter and a MeshRenderer attached.

// breakPoints is of type List<Transform>. The world position of every
// Transform in breakPoints is used to break the targetGameObject.

// interiorMaterial is of type Material. interiorMaterial fills the new
// faces created in the demolishing process.
```

After the `Demolish` call, `meshDemolisher` returns:

```
List<GameObject> brokenGameObjects = meshDemolisher.Demolish(...);

// Every GameObject in brokenGameObjects has 4 components attached:
// the GameObject component;
// the Transform component;
// the MeshRenderer component with the original material from
// targetGameObject and the interiorMaterial supplied by the users;
// the MeshFilter with the broken mesh.

// The targetGameObject is not modified in the Demolish call.
// In fact, breakPoints and interiorMaterial are not modified either.
```

Before making the `Demolish` call, the users may want to verify if the input is valid.
They can do so by calling:

```
bool isValid = meshDemolisher.VerifyDemolishInput(targetGameObject,
breakPoints);
```

`VerifyDemolishInput` checks whether all of the following properties are met:

- The world position of every `Transform` in `breakPoints` is not too far (see the Troubleshooting section) from the world origin.
- `targetGameObject`'s `Transform` does not have a negative scale (negative positions and rotations are fine).
- The world position of every vertex in `targetGameObject`'s mesh is not too far (see the Troubleshooting section) from the world origin.
- `targetGameObject`'s mesh has only one sub mesh.
- `targetGameObject`'s mesh encloses at least one volume.

- `targetGameObject`'s mesh does not have a triangle that intersects another triangle in the mesh (a triangle may intersect another triangle at their shared vertex or shared edge).

However, there are a few more things that VerifyDemolishInput does not check that the users should check to ensure correct demolish result:

- `targetGameObject`'s mesh must not contain parts that are flat.
- `targetGameObject`'s mesh must enclose only one volume.
- `targetGameObject`'s mesh's UV layout must be that every face of the mesh corresponds to one continuous chunk of UV. If a face corresponds to two or more chunks of UV, the broken game objects will have wrong UV.

Algorithm Overview:

The MeshDemolisher class works by:

1. Construct a Delaunay tetrahedralization from the `breakPoints`.
2. Break `targetGameObject`'s mesh into a set of tetrahedrons via constrained Delaunay tetrahedralization.
3. Extract the Voronoi diagram from the Delaunay tetrahedralization in step 1; calculate the intersection of the Voronoi diagram with the set of tetrahedrons from step 2.

Step 1 and step 2 are carried out by the DelaunayTetrahedralization class. Step 3 is carried out by the ClippedVoronoi class.

If MeshDemolisher encounters a bug or does not terminate, it is almost always caused by step 2.

Troubleshooting:

If MeshDemolisher encounters a bug or does not terminate, it is almost always caused by step 2.

To solve the problem, first call VerifyDemolishInput to make sure the input mesh is valid.
If the VerifyDemolishInput call resulted in a message about “A input point is too far from the world space origin”, either move the `targetGameObject` closer to the world space origin, or go to CDT_Field.cs and increase the `RANGE`.

This is because the algorithm used in step 1 and 2 (see the Algorithm Overview section) assumes every input vertex is strictly inside a big tetrahedron. The big tetrahedron is defined as:

```
float RANGE = 2000f;
int p0 = CreateNewPoint(new Vector3(-RANGE,-RANGE,-RANGE));
int p1 = CreateNewPoint(new Vector3(0,-RANGE, RANGE));
int p2 = CreateNewPoint(new Vector3(RANGE,-RANGE,-RANGE));
int p3 = CreateNewPoint(new Vector3(0,RANGE,0));

// details in CDT_Field.cs
```

If the VerifyDemolishInput call resulted in a message about “Input triangles do not enclose a volume” or “Input triangles intersect”, then `targetGameObject`’s mesh is too complex for MeshDemolisher to demolish
Note that `targetGameObject`’s mesh may appear to be closed and not self-intersecting, but this may be an illusion.

A mesh in Unity almost always contains more vertices than what the geometric object appears to have. For example, Unity’s built-in mesh for a cube has 24 vertices, but the cube only appears to have 8 vertices.
The extra vertices store additional information for rendering purposes, but MeshDemolisher must remove the duplicate vertices to function correctly. The following is how duplication is detected:

```
public int Compare(Vector3 a, Vector3 b)
{
    if((a-b).magnitude < Constant.EPSILON_F)
    {
        return 0;
    }
    ...
}

// the compare function is defined in CustomComparator.cs
// Constant.EPSILON_F is defined in GeometricObject.cs
```

If the duplication detection is malfunctioning, either manipulate the mesh or adjust `Constant.EPSILON_F`. Note that MeshDemolisher does not round the vertices’ positions.
Also note that MeshDemolisher uses the vertices’ world positions (so position, rotation, scale of the `targetGameObject` could affect the result).

Even if the VerifyDemolishInput call passes and the users manually checked other requirements (see the Mesh Demolisher Class), the MeshDemolisher may still misbehave.
If MeshDemolisher terminates with an error, then there is a bug in the algorithm (it is unlikely to be fixed).
If MeshDemolisher does not appear to terminate, then the input mesh is too complex. In general, reduce curves in `targetGameObject`’s mesh as much as possible.

Contact Author:

Email: hanzemeng2001518@gmail.com

References:

The following is a selected list of papers that are referenced (in fact, the 3 steps in Algorithm Overview corresponds to the 3 papers):

- H. Ledoux, "Computing the 3D Voronoi Diagram Robustly: An Easy Explanation," 4th International Symposium on Voronoi Diagrams in Science and Engineering (ISVD 2007), Glamorgan, UK, 2007, pp. 117-129, doi: 10.1109/ISVD.2007.10.
- Lorenzo Diazzi, Daniele Panozzo, Amir Vaxman, and Marco Attene. 2023. Constrained Delaunay Tetrahedrization: A Robust and Practical Approach. ACM Trans. Graph. 42, 6, Article 181 (December 2023), 15 pages. https://doi.org/10.1145/3618352
- Dong-Ming Yan, Wenping Wang, Bruno Lévy, Yang Liu. Efficient Computation of 3D Clipped Voronoi Diagram. Geometric Modeling and Processing, University of Cantabria, Jun 2010, Castro Urdiales, Spain. pp.269-282, ff10.1007/978-3-642-13411-1_18ff. ffinria-00547794f

The following is a selected list of GitHub repositories that are referenced (by quite a lot):

- https://github.com/MarcoAttene/CDT
- https://github.com/MarcoAttene/Indirect_Predicates

Many solutions from Stack Overflow, Mathematics Stack Exchange, Unity Forum, Microsoft Learn, Unity Documentation are referenced (I don’t remember the details).