1. Introduction.

Thank you for purchasing this asset! I hope it will come in handy and easy to use. Make sure to check video: https://youtu.be/Yk3Iug9VvEs

2. Using the asset.

1.1. Before using the asset make sure your project have ShatterableGlass tag. Usually Tags are imported from unitypackage. Check https://docs.unity3d.com/Manual/Tags.html for detailed information.

1.2. Checkout `ShatterableGlass\Demo\Scenes\Demo` scene.

1.3. If you do not need any demo related assets you can simply remove `ShatterableGlass\Demo` folder completely. Prefabs in `ShatterableGlass\Prefabs` will remain functional.

1.4. Instantiate any prefab from `ShatterableGlass\Prefabs` anywhere in the scene you want. **Make sure that every parent in prefab's hierarchy scaled `{1, 1, 1}`.** Script uses `Transform.lossyScale` to determine glass's size and `HitPoint` in world, witch **may cause weird behavior**. Warning message will appear in console, if this may happen. Check https://docs.unity3d.com/ScriptReference/Transform-lossyScale.html for detailed information.

1.5. Options:

- **Sectors** - number of areas between rays drown from `HitPoint` to glass edge.
- **Fragments per sector** – number of fragments generated per sector.
- **Simplify threshold** – If sector area less then total glass area multiplied by this value, sector will be replaced with single triangle.
- **Fragments with edges** – Generate those green lines around glass fragment?
- **Fragment edge material** – Material applied to those sides.
- **Shatter but not break** – Unbreakable glass.
- **Slightly rotate fragments** – Gives a bit more realistic effect. Applied only if `ShatterButNotBreak` is checked.
- **Destroy fragments** – Destroy fragments after some time? Applied only if `ShatterButNotBreak` is not checked.
- **After (seconds)** – Time, after fragments will be destroyed. Time a bit randomized for each fragment. Applied only if `DestroyGibs` is not checked.
- **Fragments on a separate layer** – Check if you want glass fragments to be on separate layer. Applied only if `ShatterButNotBreak` is not checked.
- **Fragments layer index** – Index of separate layer. Applied only if `GibsOnSeparateLayer` is checked.
- **Break force** – Force applied to glass fragments.
- **Thickness** – Thickness of the glass fragments. Affects colliders and edge sizes.
- **Adopt fragments** – Glass fragments will have same parent as original glass does. Checking this **may help if not every parent in hierarchy have scale of {1, 1, 1}**.

1.6. Glass shattering may be achieved in 2 ways:

- By using `SendMessage`
- By using `public methods`.

Check `ShatterableGlass\Demo\Scripts\Gun.cs` and `ShatterableGlass\Demo\Scripts\Trigger.cs` for example of both usages.

When calling `Shatter3D()` `ShatterableGlassInfo` object must be passed as an a argument. `ShatterableGlassInfo` consists of `HitPoint` and `HitDirrection` in world coordinates.

You can call `Shatter2D()` To bypass 3D calculations. This is useful for triggers and buttons. Argument is `Vector2` point.

Note: `Vector2` point must be within glass's bounds. Trying to shatter glass outside of the glass will look weird. It is recommended to call `Shatter2D(Vector2.zero)`.

2. Theory of operation.

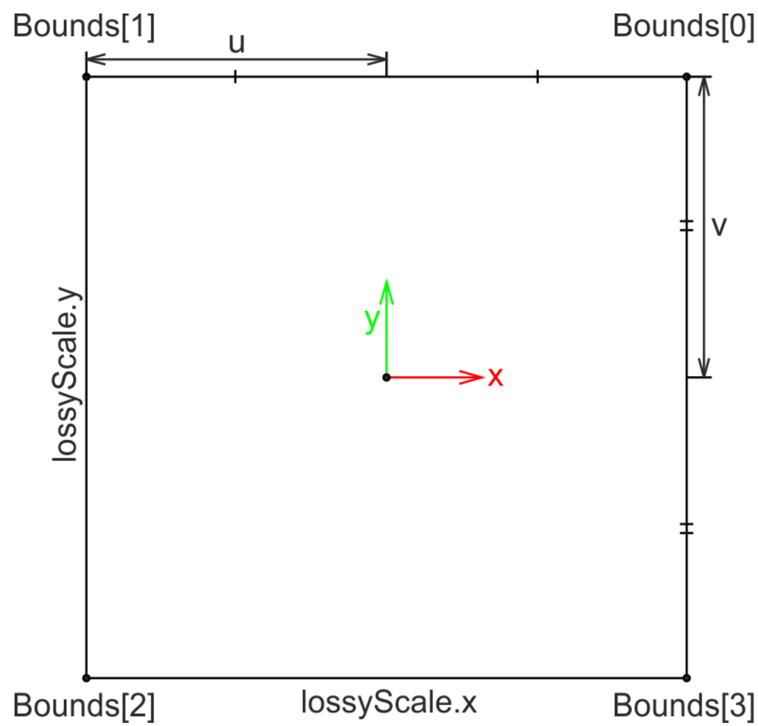

Figure 2.1. Dimensions of the glass.

Dimensions of the glass are determined by its scale in world space. However, all computations performed in local coordinates. Pivot point of the glass model counts as local center.

`Shatter3D()` transforms global `HitPoint` point, usually generated by `Physics.Raycast()` into glass local coordinates. In order to do that I calculate location of `HitPoint` relatively to glass using `transform.TransformPoint()` and simple geometry.

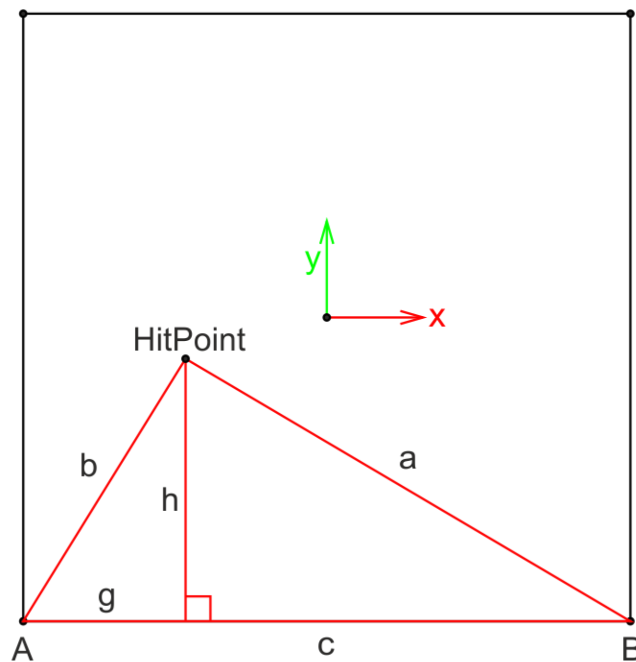

Figure 2.2. `HitPoint` calculation.

Point `A` is `{-0.5, -0.5, 0}` and point `B` is `{0.5, -0.5, 0}` in local cooorditates.
Triangle lengths calculated using `Vector3.Distance()`.

Triangle height h calculated through triangle area.
Side g calculated via Pythagorean theorem.

Note: Those calculations may seem pointless, but keep in mind that glass actually in 3D space and it may be rotated and scaled.

After `HitPoint` calculated `Shatter()` function is called.
First of all a number of `BaseLines` are created. `BaseLine` is a line, created from `HitPoint` to the edge of the glass. `BaseLine` divided into number of point, determined by `FiguresPerSector`. `BaseLine` divided using `Mathf.Lerp()`. To make shatter net look circular parameter t of `Mathf.Lerp()` multiplied by `BaseLine Roll`.

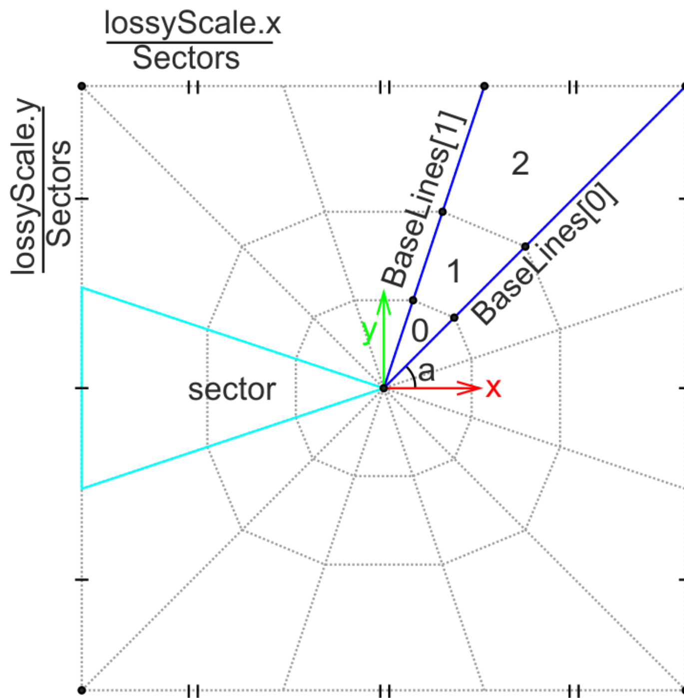

Figure 2.3. `BaseLines` layout.

`Roll` is coefficient, witch is equal to 1 when line angle is 45° and 0 when angle line is 0°. Line angle calculated using `Mathf.Atan2`. Line moved to first quadrant to simplify calculations.

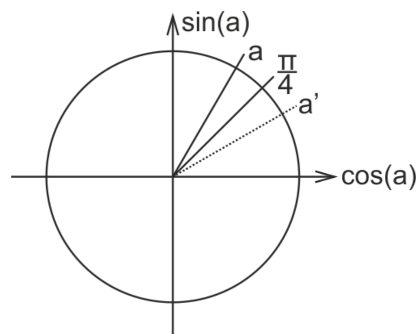

Figure 2.4. `Roll` angle calculation.
As `Mathf.Atan2()` returns angle in radians, `Roll` calculations is performed in radians as well. 45° is π/4. If angle is greater then π/4 it is gets mirrored around π/4. Resulted angle divided by π/4 to get required coefficient.
Given coefficient interpolates t between 1 and (√2)/2.

After `BaseLines` created begins process of creating `Figures` between them.

Firstly area of the Sector is calculated. If area of the sector is too small (smaller than `Area * SimplifyThreshold`) then whole sector replaced by single triangle. This is done to stop generate really small meshes and colliders.
If sector considered big enough `FiguresPerSector` generated. First figure in the sector is always a triangle, all the others is trapezes.

When all figures are generated begins mesh generation.
`Mesh` defined by:

- Array of vertices;
- Array of UV's;
- Array of triangles.

Size of array depends on figure type (triangle or trapeze) and edge generation needs.

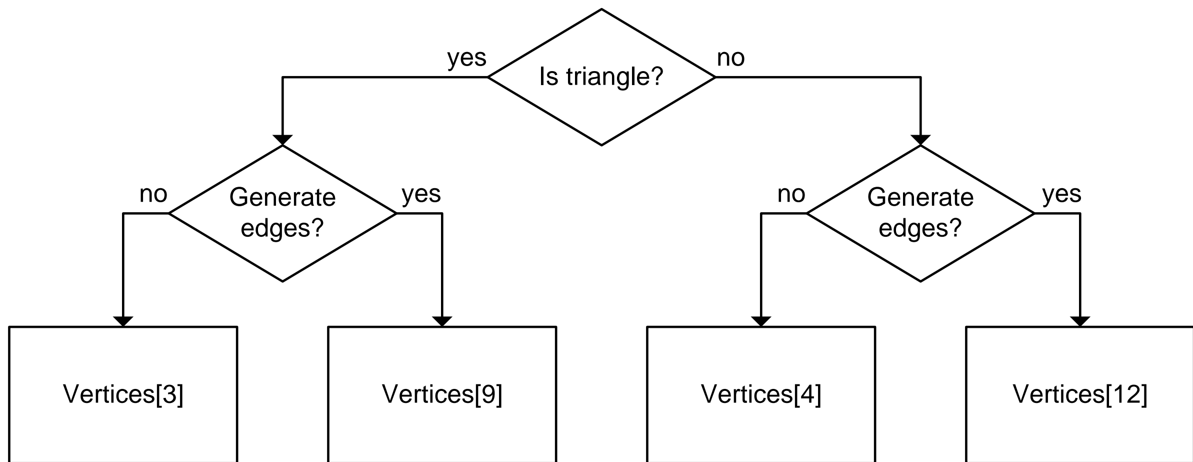

Figure 2.5. Vertices array sizes.

UV array size **must** be the same size as vertices array (this is the way unity designed), although we assign only 3 or 4 of them.
Triangle defined as triplet of indices in Vertices array. Trapeze represented as 2 triangles. Glass edges generation requires 2 triangles per side. Figure vertex indices are shown below.

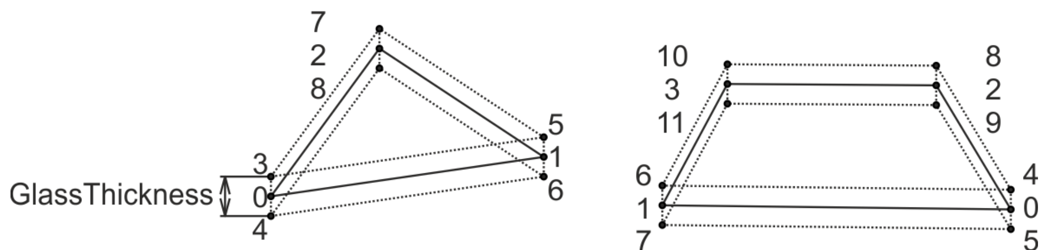

Figure 2.6. Figure indices.

Mesh is splitted into 2 submeshes. First one is the main mesh while second submesh generated only if edge generation enabled. One Material applied to one submesh. Triangle array is always constant, but depends on vertices array size.

After mesh is generated its collider may be created. As Unity do not support `RigBody` with `MeshCollider` in the way we need, small `BoxCollider` created per glass fragment. Idea is to find out radius of figure's incircle and found *insqare* of that incircle. In this way glass fragments may have small colliders, but none of them will stuck in each other.

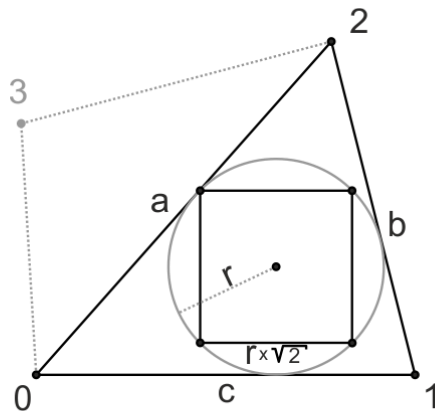

Figure 2.7. Collider size calculation.

Note: same method applied to trapeze as it can be represented as 2 triangles. Collider calculated only for one of them.

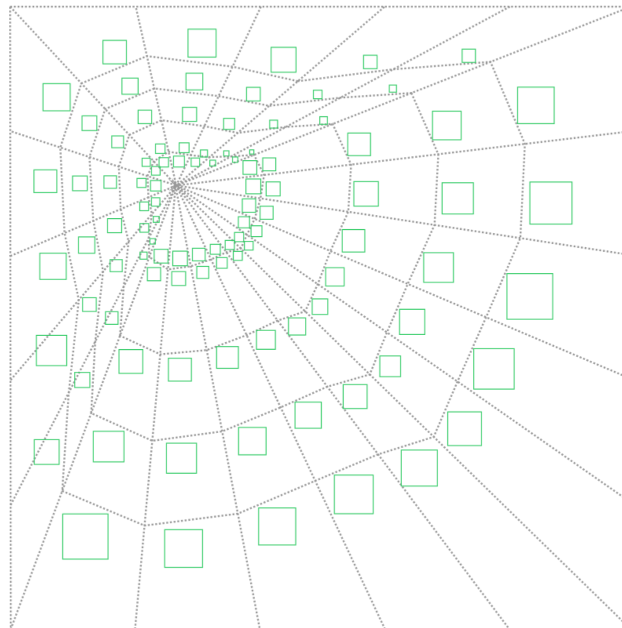

Figure 2.8. Generated colliders.

Node: Actually it is possible to add `MeshCollider` Component with the same `sharedMesh` as `MeshFilter` and make it convex. But in this way Collider will spawn highly stuck in each other witch can cause Assertion failed: Invalid AABB aabb and performance issues.

This is more of a theoretical overview of the algorithm, there are more little details in the code comments.

If you have any questions, ideas or bug reports feel free to contact me: batyastudios@gmail.com.