# AT Block 2 - Task 2
# Mesh Destruction

**Matteo Bolognesi**
**19025911**

*University of the West of England*

July 20, 2023

**The chosen task for block 2 was "Mesh Destruction". It was required to research and develop a system able to slice meshes and fill the holes generated this way with the correct materials. As a stretch goal, the system needed to be deployed into a game to showcase its potential.**

## 1 Introduction

The aim of this task is to develop a system to slice meshes and reconstructs them accordingly to the new shapes created. To implement it, is required to understand how meshes are built, how to obtain the intersection points with the cutting plane and how to reconstruct the meshes accordingly, filling all the holes created during the process. Unfortunately, this task was not implemented due to personal reasons and lack of time. What follows would have been the first steps of the final implementation. The intent of this document is not to write a guide on how to implement the system but provide an idea of the approach towards the task. All the examples provided are untested therefore might not be perfectly accurate or cover all the possible corner cases.

## 2 Related Work

The intent was to create a system able to work with simple solid shapes at first and, once achieved, expand it to work also on solid complex shapes.

## 3 Method

To cut a mesh alongside a plane some data needs to be collected. It is required to have access to the intersection points of the mesh with the plane, the vertices position for each triangle of the mesh, their UVs and normals and the plane orientation.

### 3.1 Contact Points Collection

The contact points can be collected in many different ways. The intersection plane can be instantiated runtime, can be a game object always present in the game or can be defined with a minimum of three points in space. Considering the simplest case possible, where the plane that intersects the mesh is a game object, it is possible to use the function OnCollisionStay(Collisions collisions) to access the list of contact points at every frame. The implementation of a simple button allows to get the contact points at any given time. This can easily be achieved with one of the two Unity's input systems.

### 3.2 Plane Orientation

Since the plane is a game object, it is possible to access the mesh's normals of the plane to determine its orientation. In case 3 or more points simply defined the plane, the cross-product can be used to calculate the normals.

### 3.3 Mesh Cutting

Cutting the mesh will result in two separate objects each with a portion of the old mesh plus the newly generated mesh to fill the hole. As the first step, a new

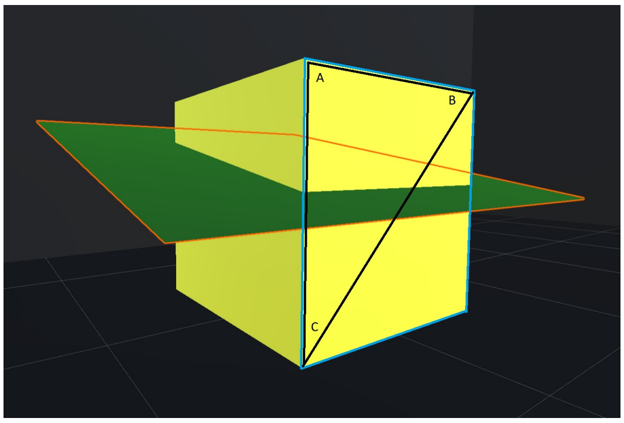


**Figure 1:** *Vertices Orientation*

object needs to be instantiated and both the MeshRenderer and MeshFilter components need to be added to it. Then, each triangle of the mesh can be in one of three different cases. It can be completely above the cutting plane, completely below the cutting plane or intersecting the cutting plane. In the first case, the triangle needs to be added to the list of triangles of the original object. In the second case, the triangle needs to be added to the list of triangles of the newly created game object. In the third case, the triangle can have two out of three vertices or one out of three vertices on the same side of the cutting plane (Fig.1). Shooting two raycasts from the single vertex towards the other two, it is possible to obtain the position of the interception with the cutting plane (Fig.2). Those two points can be used to create the triangles needed to be assigned to the triangle lists. In the case of the single vertex, the triangle is already created and can be added to one of the lists. In the other case, the intersection points and the vertices can be used to form two triangles to add to the lists (Fig.3). In Unity, the order of the vertices passed to the MeshFilter's triangle list indicates to the engine how to render the triangles. Using the orientation of the cutting plane collected in the previous steps, it is possible to determine the order of the vertices to pass to the function (clockwise or counter-clockwise). If below the plane, they need to be CW, otherwise CCW.

## 3.4 Filling the Holes

To fill the holes created, for each side of the cut, a triangle needs to be created. The vertices that define this triangle are the vertices of the sides plus the barycenter. To calculate the barycenter, the vectors representing the vertices of the cut need to be added together through vectorial sum and then the result needs to be divided by the number of vertices used in the sum (Fig.4). Then the vertices of the triangles generated this way need to be passed to the list of triangles of the mesh.

This method does not work with complex solid

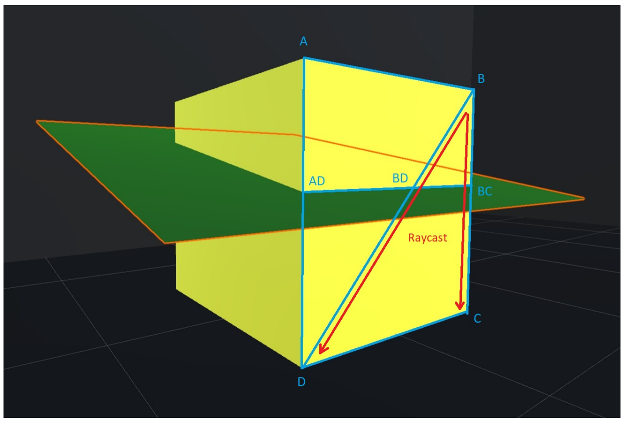


**Figure 2:** *Raycast Direction*

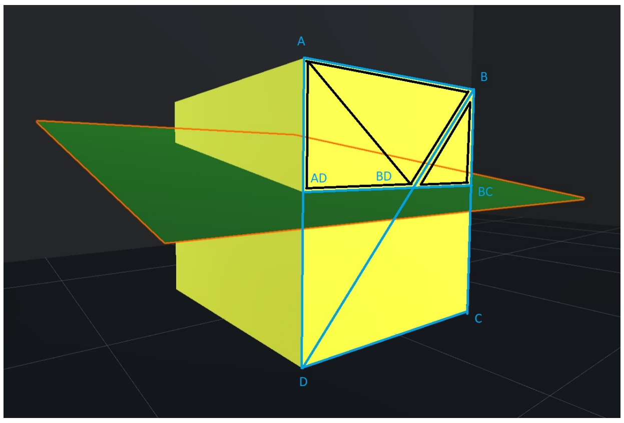
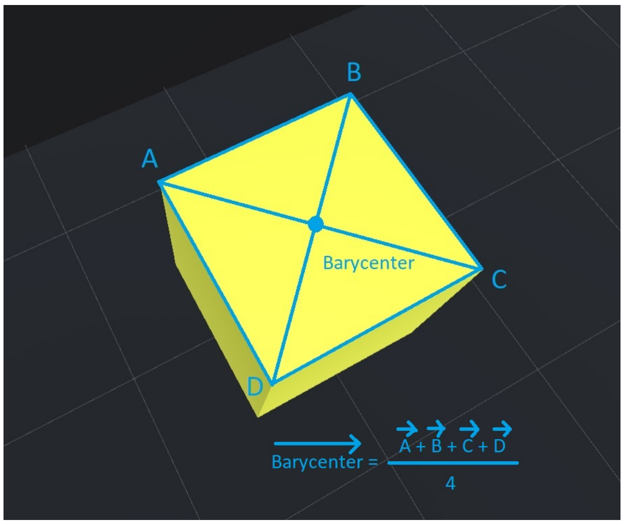


**Figure 3:** *New Triangles*



$$\overrightarrow{Barycenter} = \frac{\vec{A} + \vec{B} + \vec{C} + \vec{D}}{4}$$

**Figure 4:** *Barycenter of the mesh cut*

shapes because the barycenter can be outside the solid's volume. Applying this method to those shapes would lead to triangles spiking outside the cut plane. To avoid this problem a different algorithm needs to be used but no major research on the topic was conducted. Probably one of the ways is to divide the complex shape into simple ones and iterate this process of each of them.

## 4 Conclusion

The task required to create a system to cut meshes and fill the holes product of the cut. Unfortunately, no artefact was made for this task but a general explanation of how it would have been approached has been provided.