CRANFIELD UNIVERSITY

RÉMI NABONNE

COMPUTATIONAL FLUID DYNAMICS DATA MANIPULATION IN VIRTUAL REALITY

SCHOOL OF AEROSPACE, TRANSPORT AND MANUFACTURING
Computational & Software Techniques in Engineering

MSc in Software Engineering for Technical Computing
Academic Year: 2016–2017

Supervisor: Dr Antonis F. Antoniadis
August 2017

CRANFIELD UNIVERSITY

SCHOOL OF AEROSPACE, TRANSPORT AND MANUFACTURING
Computational & Software Techniques in Engineering

MSc in Software Engineering for Technical Computing

Academic Year: 2016–2017

RÉMI NABONNE

Computational Fluid Dynamics data manipulation in Virtual Reality

Supervisor: Dr Antonis F. Antoniadis
August 2017

This thesis is submitted in partial fulfilment of the requirements for the degree of MSc in Software Engineering for Technical Computing.

Abstract

Because Computational Fluid Dynamics often implies working with huge, three dimensional and multi-parameter data sets, the visualization of those data can be quite hard to handle and unintuitive to manipulate on the screen of a computer. The Virtual Reality technology allows the user to visualize the same results but in a three dimensional virtual environment and removes the weakness of the plain screen. The goal of this project was to develop an intuitive interface in order to interact easily with some computational fluid dynamics renderings in a three dimensional virtual environment. To reach it, two pieces of software were used: ParaView and Unity3D. The first one is an open-source application for data visualization but ended up being really complex to implement. In order to transfer the data from ParaView to a Unity3D application, the ParaUnity plug-in has been used, allowing the implementation of a new visualization application in Unity3D. Unity is a cross-platform game engine used to develop video-games and simulations, and one of the most documented technologies through the internet. It made the development a lot easier than on ParaView, and allowed the implementation of basic interaction functions, and some advanced functions from ParaView.

Contents

List of Figures

List of Abbreviations

2D	2 dimensional
3D	3 dimensional
BOOM	Binocular Omni-Orientation Monitor
CAVE	Cave Automatic Virtual Environment
CFD	Computational Fluid Dynamics
CSTE	Computational & Software Techniques in Engineering
HMD	Head Mounted Display
HUD	Head Up Display
IDE	Integrated Development Environment
IR	InfraRed
OS	Operating System
RE	Real Environment
SATM	School of Aerospace, Technology and Manufacturing
TCL	Tool Command Language
VE	Virtual Environment
VR	Virtual Reality
VTK	Visualization ToolKit
W7	Windows 7
W10	Windows 10

Acknowledgements

I would like to thank my family without whom I would never have been to Cranfield University, and for their support. Thanks as well to ISEP engineering school along with Cranfield Universtiy for offering me such an opportunity. Finally, I would like to thank Lorenzo D'Eri and Ventura Romero Mendo for enduring my morning mood during these last 6 months, and of course Dr. Antonis F. Antoniadis for giving us the opportunity to work on this really interesting subject.

Chapter 1

Introduction

This project has been done with the cooperation of Lorenzo D'Eri and Ventura Romero Mendo, from the MSc CSTE. In this regard the work has been separated in 3 different implementations:

- User input (Ventura Romero Mendo)
- Menu, object scaling Paraview to Unity data import (Lorenzo D'Eri)
- Paraview functions (Rémi Nabonne)

It is recommended to read the 3 reports in order to see the project from a wider angle.

1.1 Objectives

The main objective of the last months has been to successfully implement some Paraview visualization functions in VR, such as the Clip or Slice function, and to improve them with an Undo function. The latest versions of ParaView has been implemented with a plain VR visualization mode, with only a grabbing and scaling interaction available. A lot of time has been spent on trying to implement the already existing functions of ParaView into its own VR mode, but given the complexity of the source code, it has been decided to develop another visualization application on Unity, with the help of a ParaView plug-in called ParaUnity.

1.2 Management

1.2.1 Gantt Diagram

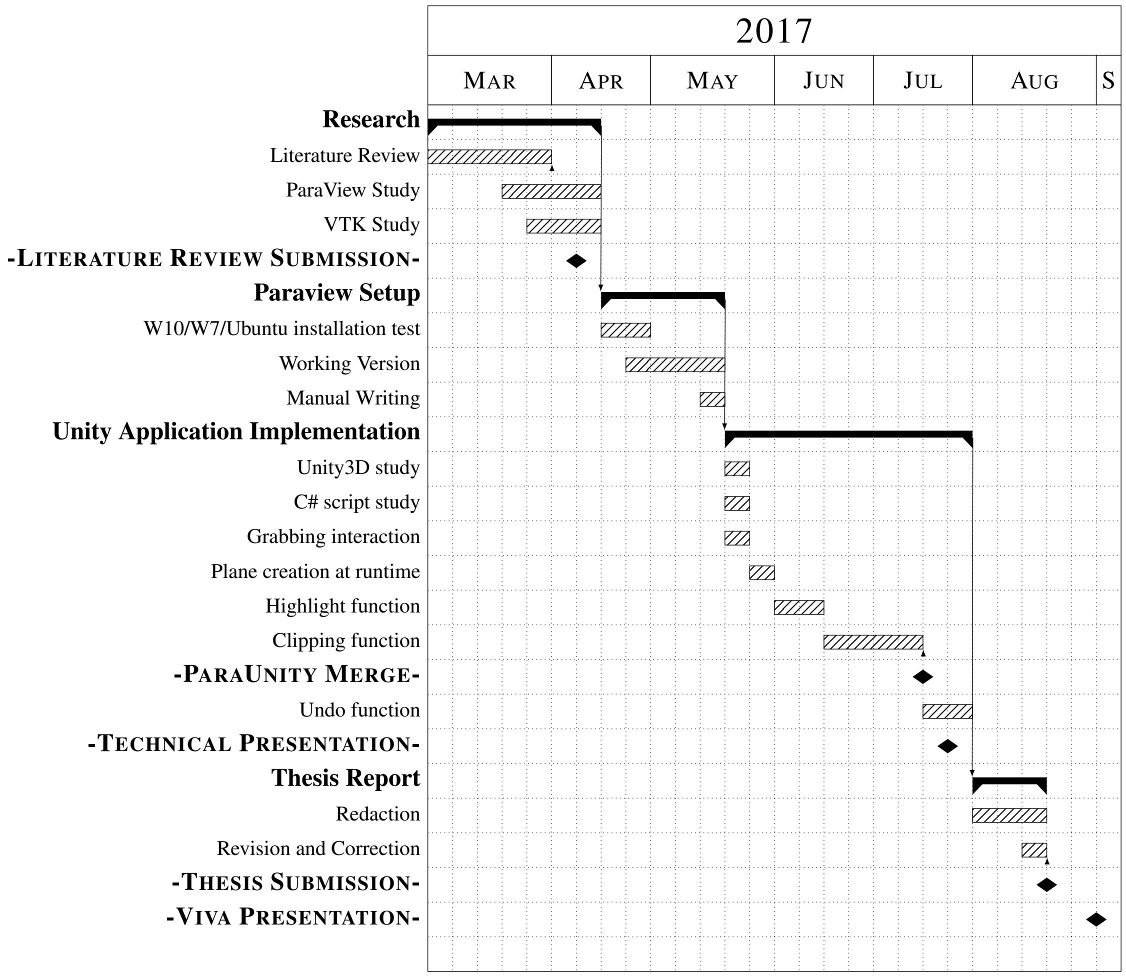

Figure 1.1: Gantt chart

This Gantt Diagram shows the time management of the whole project. Given that VR

is a pretty new technology, the documentation on it is quite limited. Most of the **literature review** has been dedicated to the documentation on the ParaView application and its 3D computer graphics & visualization library VTK which stands for Visualization ToolKit.
As stated earlier, ParaView is a huge application and nearly no support is provided, so the **Setup** has taken more time than expected thanks to the huge number of dependencies that makes it well suited for the visualization of any kind of scientific data visualization. The same amount of time has been spent on trying to make the application work on Ubuntu, before realizing that there was no official support for SteamVR on this OS, wich is the environment needed by the HTC Vive in order to work. A "'How to compile ParaView from its sources"' guide has been writen, describing each step to follow in order to run the software and modify it.
Thanks to the huge **Unity** community it has been possible to learn how to create a 3D application in less than two weeks and to implement it with some basic functionalities such as the grabbing interaction. The part that took most of the time to realize was the implementation of the Clip function since it has needed some more documentation on how the meshes on Unity are working. The **ParaUnity Merge** refers to the fusion of my work with the work of Lorenzo D'Eri in order to have a proper application to show. From this point it has been possible to test the interaction functions on meshes imported from the ParaView application.

1.2.2 Git

Git is a version control system really useful for projects on which several people are working on. It tracks changes in computer files for each user and allows them to work on the same files at the same time without overwriting on each other. Since 3 students were working on this project, Git and its web based version has been of a great help, especially for the fusion of the Unity application.

Chapter 2

Prerequisites

Before diving right into the project explanation, it is necessary to understand the technologies used and the tools that were used to build the application.

2.1 Literature Review

Virtual Reality is a technology that simulates the physical presence of the user in a 3D 360-degree VE created with the help of an interactive software linked to a hardware. These past few years, the ways of using VR have spread widely from the military training simulations, to the video game industry. However, not a lot of applications for the visualization of CFD data in VR can be found.

In this literature review we will see the VR capabilities, before discovering the handling of movement in a VE. Then we will see the limits of the technology before presenting the different tools to be used in order to successfully create an interface for the immersive visualization of CFD data.

2.1.1 Virtual Reality Capabilities

The developer can be the creator of a custom designed, 3D VE in which the user is comfortable with because his body is being part of the environment. The interface of a VE is nearly transparent, intuitive and it is called ’Anthropomorphic’ 3D interface [1]. The fact of being printed in a VE makes the handling of the 3D display way more consistent than a 3D render printed on a 2D computer screen as the user’s movement will help him establish spatial cues.

Furthermore, the VR interface can be built to provide a ”real time” data exploration [1], allowing the user to travel through space, time, or even both.

2.1.2 Movement

[3, 4] There are many ways of tracking the movement of the user, and by combining them together it creates a lot of possibilities for the interface building.

The movement is defined by a direction and a speed, and can be computed with the help of the VR kit.

There are three most immersive ways to keep track of the direction in a situation where the user is virtually flying:

- The *Pointing mode* keeps track of the direction of the controller only. It is used for case in which the user is being moved in the VE, he will then move towards where he points his controller, while he will be able to look around using the headset sensors.

- In *Crosshairs mode*, the controller is represented in the VE as a cursor that the user can move around him. The direction is defined by the vector that goes from the headset to the cursor. It implies that the user will need to keep the cursor at sight in order to control the direction thus preventing him from looking elsewhere.

- The *Gaze directed mode* keeps track of the direction of the headset only.

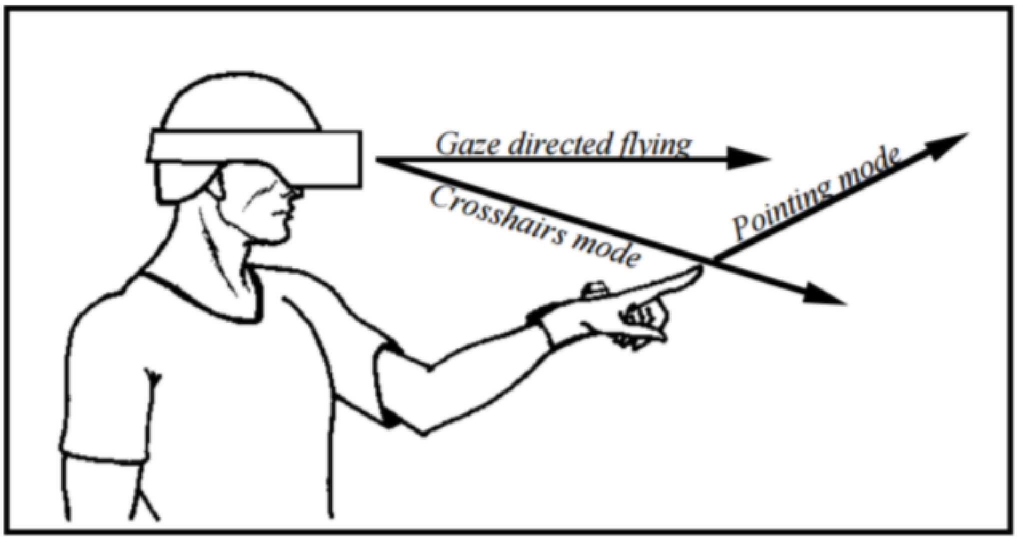

Figure 2.1: Main Flying Modes

However, the most immersive way to render movement in a VE is by letting the user physically move and record his position with cameras/sensors in order to create a VE in which he is walking. The headset's sensors allows the user to look around him while walking and freeing the controllers for other purposes (being able to interact with virtual objects in the VE or with a HUD).

The VE is then limited to a certain scale of the real environment because even though he is able to go through virtual walls, he can't in the real world. In order to fix this problem, the controller may be used to point to a location where the user wants to be teleported to. Once the movement has been tracked, it is possible to customize the $\frac{realmovement}{virtualmovement}$ ration in order to have more or less realistic rendering [3].

2.1.3 Interaction

[3, 4, 5] If the movement in the VE is only relying on the user's spatial position in the RE, it is possible to use the controllers for other purposes such as navigating through a HUD, manipulating virtual objects or scaling the VE. Regarding the **selection of an object or menu item**, there are some ways to render an intuitive interface:

- The *Local Selection* allows the user to move the cursor to a zone spatially dedicated to the selection of the item.
- The *At-a-distance Selection* displays a virtual laser beam corresponding to the direction the controller is heading toward. The goal is to select an object pointing at the item.
- The *Gaze directed Selection*, as for the movement, the user must look directly at the item he wants to select.
- The *Voice Selection*, less intuitive than the other options, it consists in saying a word related to the object of the selection. It implies to have a voice recognition device and to make the user learn a list of vocal commands.

As for the **manipulation/scaling of an object** that has just been selected, there are 3 main ways:

- The *Hand specified* control, make the user virtually holding the selected object. The orientation and movement of the hand/controller allows the user to manipulate the virtual object. It is the most immersive method.
- The *Physical controls*, an additional controller is dedicated to the manipulation of the virtual object, allowing the best precision of all methods.

- The *Virtual controls*, same as previous method, but the controller is virtually displayed in the VE.

2.1.4 Virtual Reality Limits

[1] As the interface is completely different from the one of a classic desktop and mouse computer, it has been necessary to reinvent the human-computer interactions all over again. In addition to that, the VE rendering is a high-cost process because it needs to print two simultaneous images, corresponding to the eyes of the user in order to render a 3D environment. It means that a High quality VR interface needs to do a lot of computations.
In addition to that, the images need to be refreshed as quickly as possible to keep a good illusion. In fact, the comfort deadline of the human eye is 24fps, and can notice a frame rate change until 120fps. To render as many images during such a little time is also a a computationally-costly process, because the movement, position of controller, direction of the head, must be refreshed on each frame. The reaction time between a user action and its virtual resolution needs to be under 0.1 second to allow a fast and accurate environment manipulation [1].
However, one psychological limit is the human brain, because some users can develop simulation sickness, that is similar to motion sickness. Sometimes it is induced by the internal ear and eyes of the user sending different information on the position and movement to its brain. Another cause is having a screen at less than 10cm from the eyes. The user can feel uncomfortable with the VR display during the simulation and sometimes after [2].

2.1.5 Virtual Reality Devices

There are several ways of displaying a VE:

- The CAVE is a VR system composed of a cubic room where the VE is projected on the walls. It is not a 3D environment but it's still at 360 degree motion free [6, 7]
- The BOOM is a headset fixed to a mechanical arm, the user is then in a VE similar to the one displayed by a HMD but his movement is limited to the length of the mechanical arm [7].

[10] Nowadays, the HTC Vive HMD is considered as the best VR headset, providing a lot of sensors and using the room scale tracking technology. The Vive hardware is composed of the HMD (32 IR sensors), 2 controllers (19 IR sensors each) and two base stations. It allows a 110 degree sight angle with a refresh rate of 90Hz at a resolution of 1080x1200.

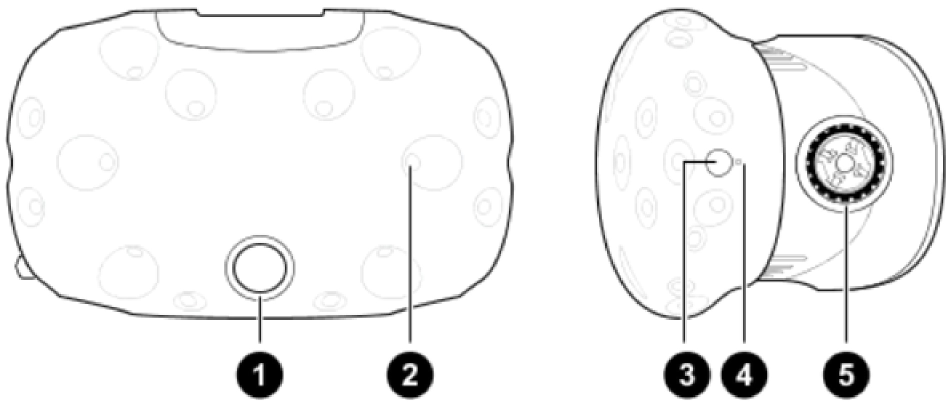

Figure 2.2: HTC Vive Headset

The headset is basically the window on the VE, its IR sensors are tracked by the base stations, but it has as well a gyroscope and an accelerometer to keep track of the rotation of the user's head.

On the Figure 2.2 it is possible to see the location of some of the **front IR sensors (2)**.The **front camera (1)** allows to identify any object in a room. On the left side of the headset, the **I/O button (3)** is right next to the **status light (4)**. You can adjust the distance between the two inner screens with the **control wheel (5)**.

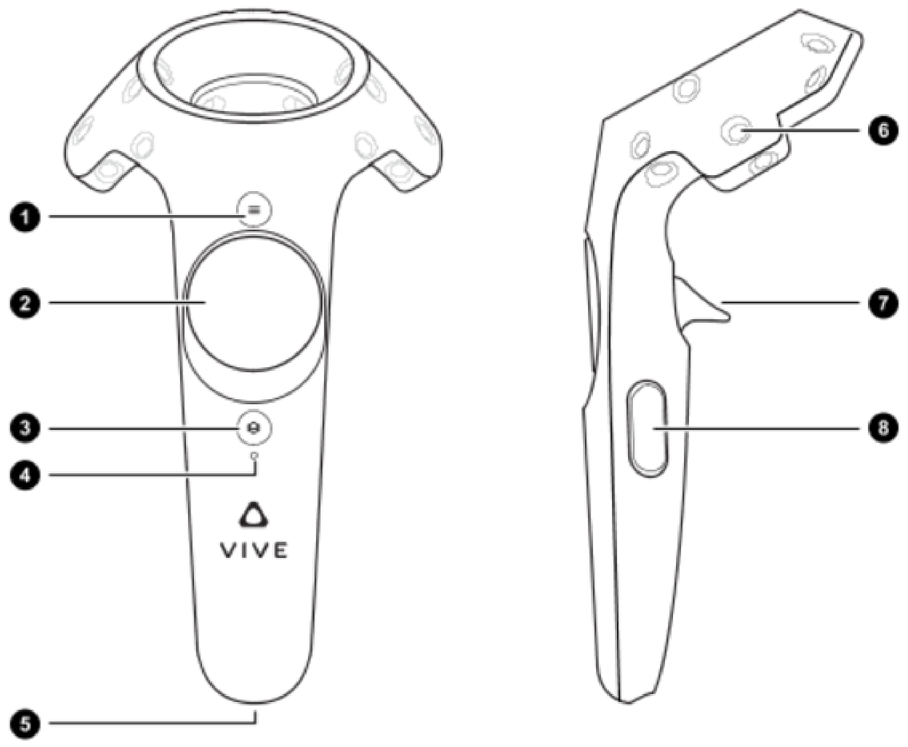

Figure 2.3: HTC Vive Controller

The controllers are the main component for interaction with objects in the VE, its IR sensors are tracked by the base stations as well.

On the Figure 2.3 it is possible to see the location of some of the **IR sensors (6)**. The **Menu Button (1)** used to open an application menu. Right below this button is the **touchpad (2)** with haptic feedback. The **System button (3)** above the **status light (4)** allows the user to open the Steam menu. The controllers use a battery that can be charged through the **micro-USB port (5)**. The **trigger button (7)** is located below the controller and **2 grip buttons (8)** are located on each side.

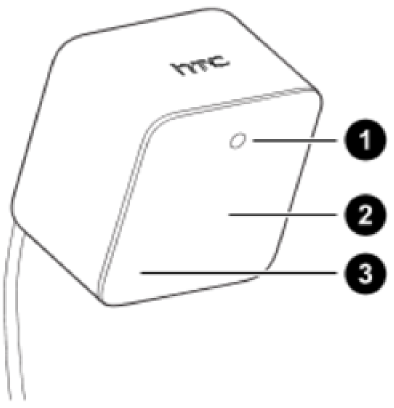

Figure 2.4: HTC Vive Base Station

The base stations are placed at two corners of the VE in order to always have the headset and the controllers at sight. They send signals to the controllers with a built-in IR generator right behind the **front panel (2)** through wich passes the **status light (1)**. A built-in **channel indicator** can be seen behind the lower right part of the front panel.

2.2 Technologies Used

2.2.1 ParaView

ParaView [8] is an open source application for scientific data visualization which internal structure is based on several layers.

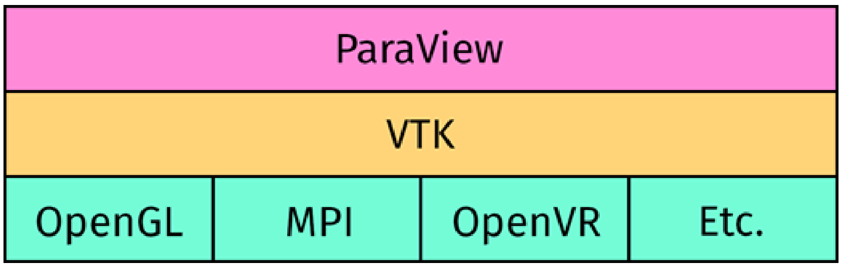

Figure 2.5: ParaView Layers

VTK is a huge library for 3D computer graphics, image processing and visualization of scientific data. It is written in C++ (kernel) and allows code wrapping in Python, Java and TCL for customization purposes or plug-in development. It has been co-developed by ParaView and Kitware [9] community.
ParaView also uses other specific purpose libraries such as OpenGL for graphics, MPI for parallelisation and efficiency improvement and OpenVR for VR capabilities.
The VR mode of ParaView only came out on the version 5.3.0 released on March 10 2017 and was only implemented with the basic VR interactions such as scaling and grabbing.

2.2.2 Unity3D

Unity [11] is a cross-platform game engine for video-game and simulation development. It is developed by Unity Technologies and written in C C++ languages and supports 2D and 3D graphics. Although it is an Object Oriented IDE, it allows to handle the behavior of objects with scripts written in C# or JavaScript. The latest version of Unity is the 2017.1 released on 11/07/2017.

Chapter 3

Unity Application

The hardest part of this project has been in understanding how do meshes on Unity works in order to develop a Clip function. Therefore, it's important to explain in depth how a Unity object works.

3.1 Basic Interaction

3.1.1 GameObject

The **GameObject** in Unity is the class to describe all entities. Despite the few variables it contains, it comes with the base Component **Transform**. A Component is a class that must be attached to a GameObject in order to extends its capabilities. For example the Transform Component will provide the GameObject with its position, rotation and scale information. These parameters can be modified in the IDE before each simulation, or at runtime using a **script** Component.

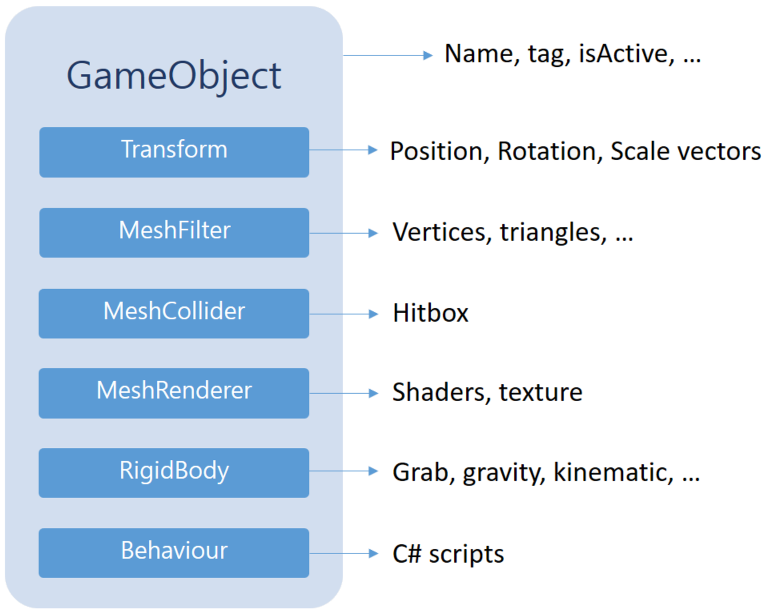

Figure 3.1: GameObject and Components

[12] When it comes to the manipulation of meshes, there are several Components to take into account:

- The **MeshFilter** is one of the two base classes for instantiating a mesh. It will define its vertices, triangles, texture coordinates and pass them to the MeshRenderer Component in order to render the mesh in the VE. The next section will go deeper into the meshes explanations.

- The **MeshCollider** is defining the hitbox of the GameObject. It willl detect a collision between a defined space around the GameObject and another GameObject. This Component is necessary in order to implement basic interactions, since it is not possible to grab an object if the controller collider is not within the GameObject one.

- The **MeshRenderer** is the second base class for instantiating a mesh. It is taking in account the mesh parameters given by the MeshFilter and the parameters of the scene in order to render the mesh in a realistic way.

- The **RigidBody** is the class that adds physics to the GameObject so they can be affected by gravity and react to collision with other GameObjects with a RigidBody attached to them. This Component has a scripting API that allows an in depth customization of the GameObject physics. For a scientific data visualization, the *useGravity* parameter is useless, so it can be disabled, and gravity won't affect the GameObject. For the project, this Component has been only used to implement the grabbing interaction, because you can't grab a GameObject if there is no RigidBody attached to it. Another important parameter to keep in mind: *isKinematic* will give the object its physics such as a movement along the trajectory a force give him. If gravity is enabled, the object will fall to the ground, but if not, only the collision with other rigid GameObjects will make it move in the VE. Given that we don't want the CFD data to move when we hit it with a controller, this parameter has been disabled as well.

3.1.2 Grabbing Interaction

The most simple interaction with a VE to implement in Unity is the grabbing one. The pre-made function **OnTriggerEnter(Collider collider)** is called whenever a GameObject collider enters the collider of the GameObject the script is attached to. In order to grab an object, this function must be implemented in a script attached to a controller. It will store the collided GameObject into a **pickup** variable so the controller will be able to interact with it from the script. When the GameObject leaves the controller collider, the pickup variable is reset to null.

```
private void OnTriggerEnter(Collider collider)
{
  pickup = collider.gameObject;
}
private void OnTriggerExit(Collider collider)
{
  pickup = null;
}
```

Then the controller can press a trigger button while the colliders meet to grab the object. In order to move a GameObject, it is necessary to modify the **Transform** Component variables of the grabbed object. For this project, while an object is being grabbed, it becomes part of the controller, so the position, rotation and scale of the grabbed object are the same as in the controller Transform Component.

```
  if (pickup != null && controller.GetPressDown(triggerButton
    ↪ ))
    {
      //Debug.Log("Trigger Button is down");
      pickup.transform.parent = this.transform;
    }
    if (pickup != null && controller.GetPressUp(triggerButton
      ↪ ))
    {
      //Debug.Log("Trigger Button is up");
      pickup.transform.parent = null;
    }
}
```

3.2 Advanced Interactions

3.2.1 Meshes in Unity

A mesh is a collection of triangles that defines the shape of an object in 3D computer graphics and solid modeling. Each triangle has 3 vertices and each vertex can be part of several triangles.

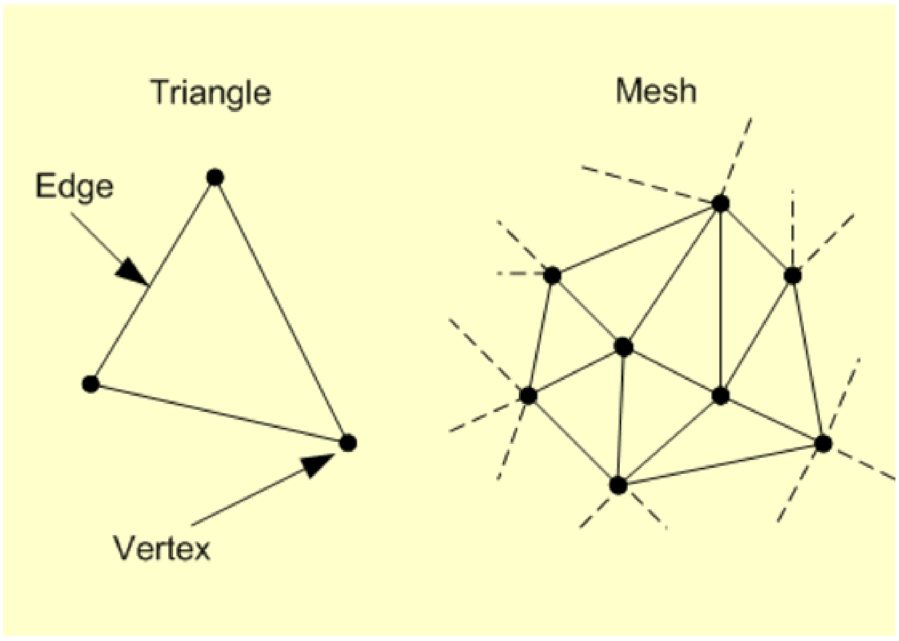

Figure 3.2: Mesh, Triangles and Vertices

In Unity3D, it is the MeshFilter that determines the vertices and triangles of the mesh. Given that a VE is in 3D, the coordinates for the vertices have 3 components along the 3 axes. The vertices coordinates are stored in a **Vector3 array**. The type Vector3 is used to represent 3D vectors and points and is used to pass 3D positions and directions.
The triangles are a bit more complex to manipulate, because their coordinates are not stored into an array. However, they are stored into an **integer array** that contains the indices of its vertices into the Vector3 vertices array.

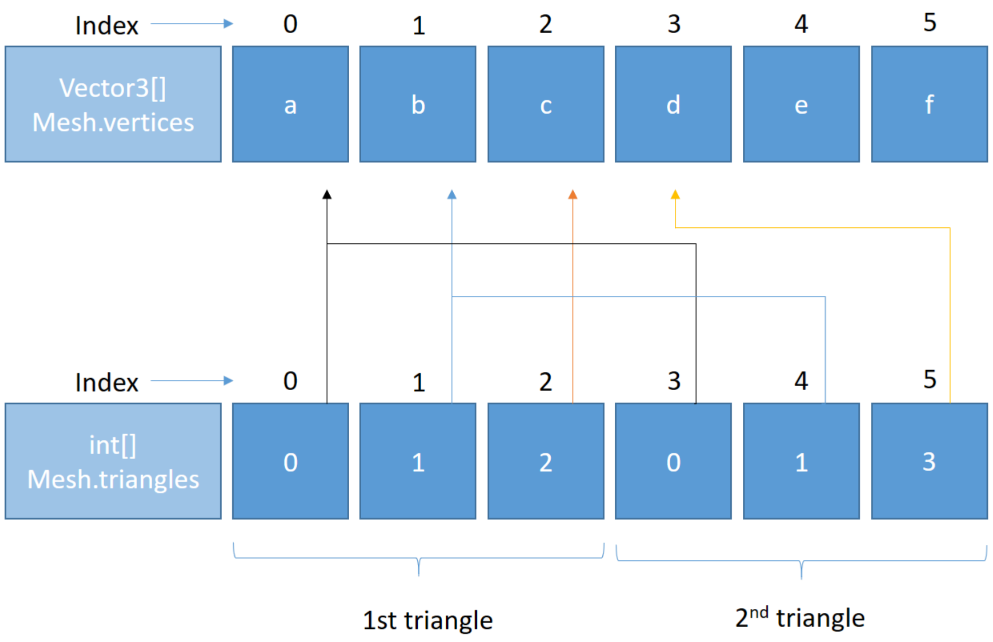

Figure 3.3: Triangles and Vertices Relation

As a triangle has 3 vertices, the integer triangle array defines a new triangle every 3 elements. A simple way of getting the coordinates of a triangle's vertices when knowing its order 'N' in the triangle array is to loop from the element N*3 to the element (N*3)+2 in order to get the vertices indices in the vertices array.

```
//Getting the Mesh Component of the GameObject and its
   ↪ vertices and triangles
Mesh mesh = gameObject.GetComponent<MeshFilter>().mesh;
Vector3[] vertices = mesh.vertices;
int[] triangles = mesh.triangles;

//3rd triangle
int N = 3;
//A 3 elements array of Vector3 to store the vertices
```

```
    ↪ coordinates
Vector3[] triangleNVertices = new Vector3[3];

//Index of the triangleNVertices
int k = 0;
//Loop through 3 integers
for(int i = N*3; i < (N*3)+3;  i++)
{
  triangleNVertices[k] = vertices[triangles[i]];
  k++;
}
```

3.2.2 Highlight function

This function has been developed in order to have a visual representation of the vertices that would get deleted by the Clip function. It uses a plane to place where the user wants the triangles to be highlighted, as each triangle vertices that the plane cuts will be stored into an array and linked with a pink line. Once the user wants to highlight, he needs to press the grip button (Figure 2.3, (8)). The source code can be seen in the Appendix, but a workflow diagram of the function (Figure 3.4) is best suited to explain the algorithm.

When the function is called by pressing the grip button (Figure 2.3, (8)), the first thing it does is checking if the cutting plane collider is triggered by the collider of another GameObject. If not, the plane cannot cut anything and the function is exited.
Then it checks if the function has already been called since the application is running. If not, the initialization part creates the objects that will be used by the algorithm:

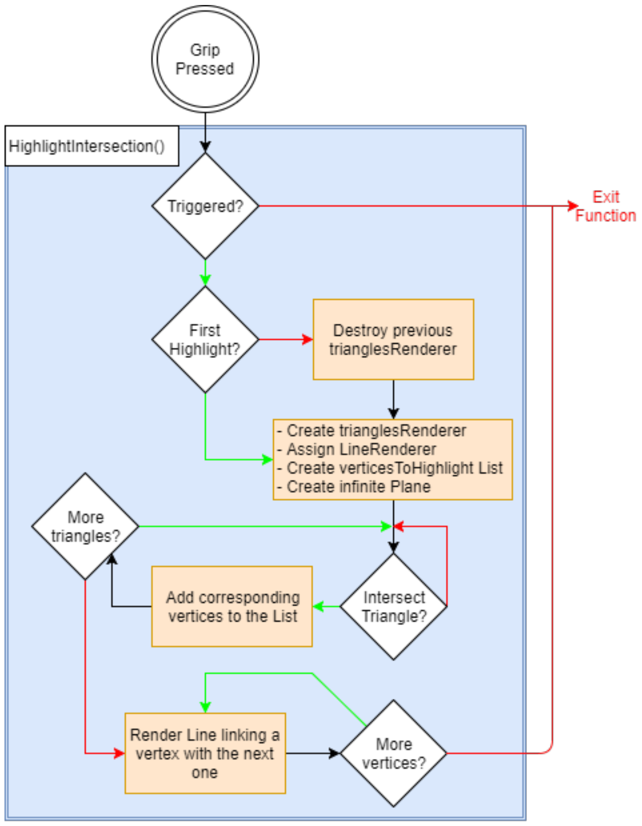
Grip Pressed
HighlightIntersection()
Triggered?
Exit Function
First Highlight?
Destroy previous trianglesRenderer
- Create trianglesRenderer
- Assign LineRenderer
- Create verticesToHighlight List
- Create infinite Plane
More triangles?
Add corresponding vertices to the List
Intersect Triangle?
Render Line linking a vertex with the next one
More vertices?

Figure 3.4: Highlight Function Workflow Diagram

- **trianglesRenderer** is the name given to the GameObject that will draw lines between the vertices to highlight.
- A **LineRenderer Component** is then attached to the triangleRenderer. It will allow to draw lines between two points in the VE.
- The **verticesToHighlight** is a list that will get every vertex that belongs to a triangle that has been cut by the plane when the user pressed the grip button.
- An infinite plane is created, it is not visible in the VE but is on the same plane as the visible plane the user is grabbing. The visible plane is a GameObject but the infinite plane is an instantiated structure defined by Unity, so it has specific function that will be more helpful than the GameObject plane. For exemple, the

function **GetSide** allows to know if a point is in front or behind a plane and was really usefull in this project.

If the Highlight function has already been used, it destroys the previous trianglesRenderer GameObject and creates a new one.
The first loop of the algorithm runs through the triangles array, checking the position of their vertices compared to the infinite plane using the GetSide function. This function returns 1 if the vertex is on the positive side of the plane, 0 if not. If a vertex is on one side of the plane and the other two are on the other side, it means the plane is cutting through the triangle so all its vertices are added to the verticesToHighlight list. If all vertices of a triangle are on the same side of the plane, the triangle is entirely on one side of the plane and its vertices won't be highlighted.
Once every triangle position has been checked, the lineRenderer Component is provided with the verticesToHighlight list with a loop. This way, each vertex will be linked to its direct neighbours with a pink line (for example, vertex n will be linked with vertex n-1 and n+1 if they exist). The triangles are not entirely drawn because they miss 1 side (the side that links the vertices n and n+2) but the function uses only one lineRenderer Component.

This function was helpful in the integration of the Clip function as it was first used in order to understand how the GetSide function of the plane structure was working. The highlight function made clear that the GetSide function was not the problem because it was working perfectly.

3.2.3 Clip Function

This function is used to cut a mesh in two parts. In Unity, a plane can only be seen by one side (the positive one), if the user looks at a plane from the negative side, he won't see it. This has been used as a visual cue that indicates the part of the mesh that will be kept. In fact only the part of the mesh in front of the visible face of the clipping plane will stay in the VE, the rest is stored into an array for the Undo function, but is removed from the scene.
The Clip function is a bit more complex than the previous one, as the algorithm uses two loops: one through the vertices and another one through the triangles.
The following Workflow diagram will help to understand what the function is doing (Figure 3.5):

Like the Highlight function, it first checks if the clipping plane collider is triggered by another GameObject collider. If not, the Undo function that will be detailed in the next section is called.
If the clipping plane hits another GameObject, it gets the last stored mesh before creating the posVerticesL and posTrianglesL lists that will store the fully positive triangles and their vertices. A fully positive triangle is a triangle which vertices are all in front of the visible side of the clipping plane.
The tricky part of the algorithm was in the handling of the triangles. Indeed, since the elements of the triangles array are the original index of their vertices in the vertices array, the fact of keeping only the positive vertices and store them into another List won't keep the same order. So the elements of the triangles array can't be the index of the positive vertices List. That's why another array is created: the posVertexIndex that keeps track of the change of index of a vertex, and will help updating the positive triangles list.

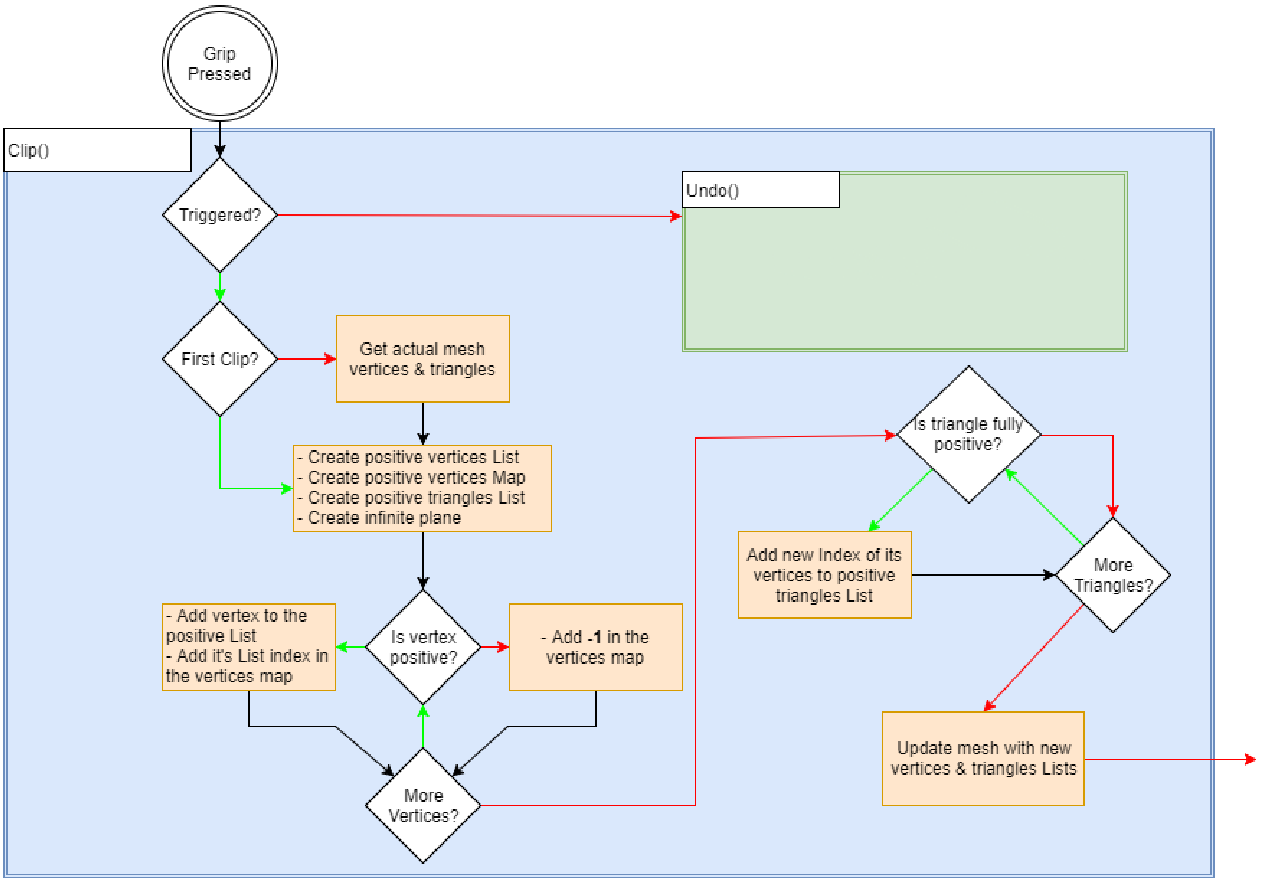
Grip Pressed
Clip()
Triggered?
Undo()
First Clip?
Get actual mesh vertices & triangles
- Create positive vertices List
- Create positive vertices Map
- Create positive triangles List
- Create infinite plane
Is triangle fully positive?
Add new Index of its vertices to positive triangles List
More Triangles?
- Add vertex to the positive List
- Add it's List index in the vertices map
Is vertex positive?
- Add -1 in the vertices map
Update mesh with new vertices & triangles Lists
More Vertices?

Figure 3.5: Clip Function Workflow Diagram

The first loop runs through the Vector3 vertices array. If a vertex is positive (if it is in front of the visible side of the clipping plane), it is stored into the posVerticesL list and its new index is stored into the posVertexIndex array. If a vertex is negative, -1 is written in the posVertexIndex at the same index as the original vertices array. The Figure 3.6 sums up the procedure.

Once every vertex position has been checked, the algorithm runs through a loop on every triangle. For each triangle, it doesn't check the vertices of the orignal array, but the values of the posVertexIndex instead, in order to map correctly the positive triangles list with the positive vertices list. A fully positive triangle means all of its vertices are positive, so by

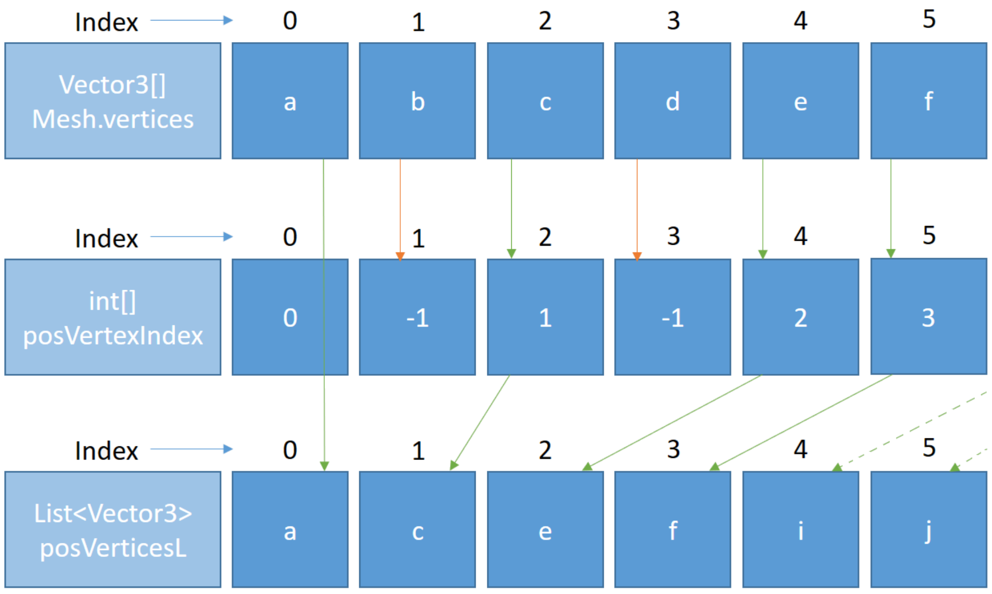

Figure 3.6: Building of the positive vertices List and its map to the orignal vertices array

looking at the posVertexIndex array with the elements of the fully positive triangles, it will indicate the new index of the triangle which must be the element of the new positive triangles list. The Figure 3.7 sums up the procedure.

Once every triangle has been checked, the two lists can be transformed into arrays with the **ToArray** function and the mesh original triangles and vertices arrays can be overwritten with the positive ones.

Because it is working with triangles and vertices only, this function is leaving a ragged edge where vertices were negative. This is an issue when working with low resolution meshes as the triangles can clearly be seen, but the higher the resolution is, the more regular the edge is. Since CFD is dealing with huge datasets, the resolution of the meshes is usually very high so the edge seems clean.

However, the main drawback of this algorithm is that it does not deal with textures, so when the vertices and triangles are overwritten, the textures gets messed up. The **Vector2**

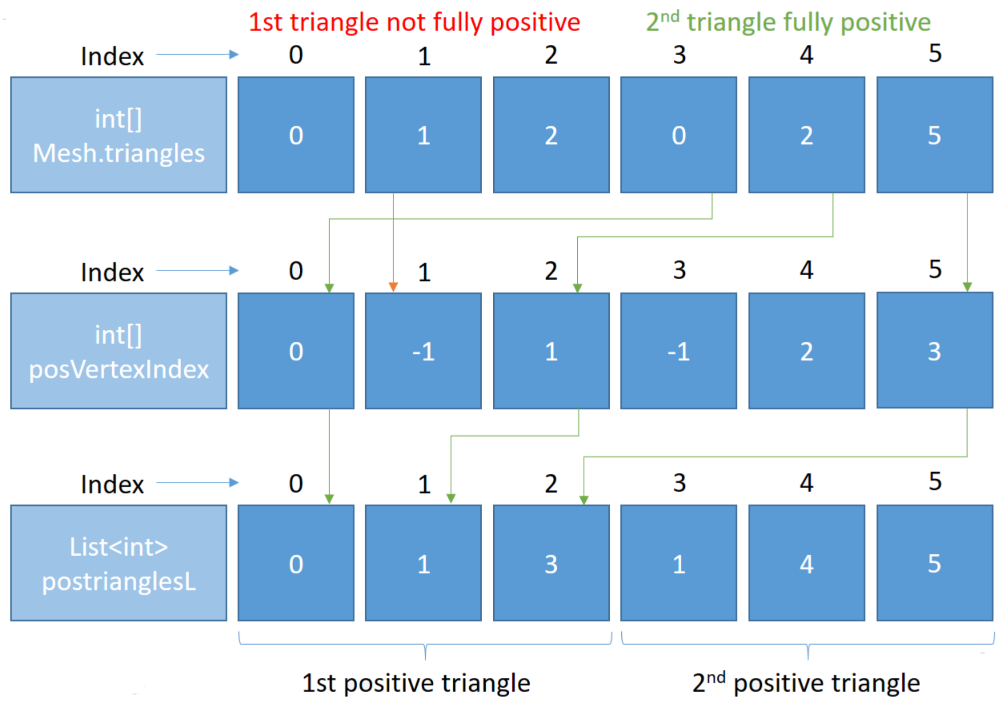

Figure 3.7: Building of the positive triangles List with the help of the posVertexIndex map

Mesh.uv is the base texture coordinate of the Mesh; it has been tried to store them like the vertices, but it didn't change anything, meaning that the problem might not only be due to the textures. The shaders are one of the most complex parameter to deal with when doing mesh transformations, so no research have been done on this subject yet.

The pre-ParaUnity version of this function used to clone the GameObject to clip, in order to end up having to separate meshes: One being the positive part of the original mesh, the other one being its negative part. With the ParaUnity version of the application, having multiple meshes implies having multiple GameObject. Given the high resolution of the meshes imported from Paraview, it ended up being a time consuming process and it has been decided to stay as faithfull to the Paraview version of the function as possible.

3.2.4 Undo Function

The idea behind this function was to create a way of getting to previous results of the clipping function by storing every successive posVerticesL and posTrianglesL in 2D memory Lists.

At each call to the Clip function, the actual vertices and triangles arrays values are stored into the next empty element of the 2D vertices and triangles lists.

When the Undo function is called by pressing the grip button while the clipping plane is not triggered (Figure 3.8), the mesh triangles and vertices are overwritten with the second last element of the 2D lists and the very last element of those two lists is suppressed.

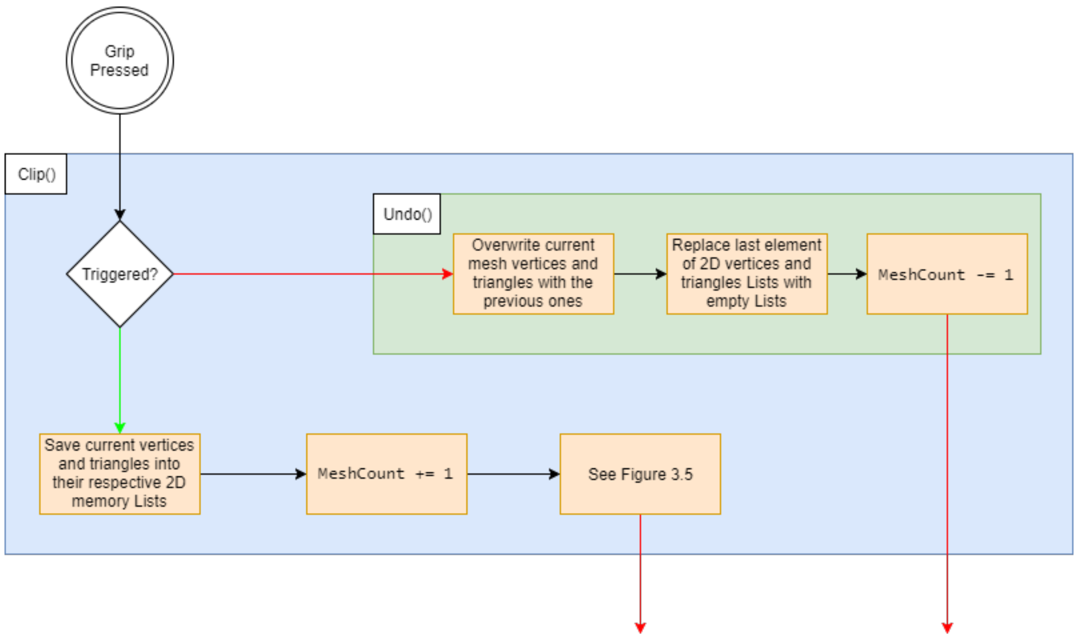
Grip Pressed
Clip()
Triggered?
Undo()
Overwrite current mesh vertices and triangles with the previous ones
Replace last element of 2D vertices and triangles Lists with empty Lists
MeshCount -= 1
Save current vertices and triangles into their respective 2D memory Lists
MeshCount += 1
See Figure 3.5

Figure 3.8: Undo Function Workflow Diagram

However, the same problem as the Clip function is occuring here, so the texture is not correctly set back to previous versions.

Chapter 4

Results

The functions have been tested on CFD data provided by Dr.Antonis F. Antoniadis. The mesh that can be seen on Figures 4.1 and 4.2 represents the air flow around a cylinder.

4.1 Highlight Function

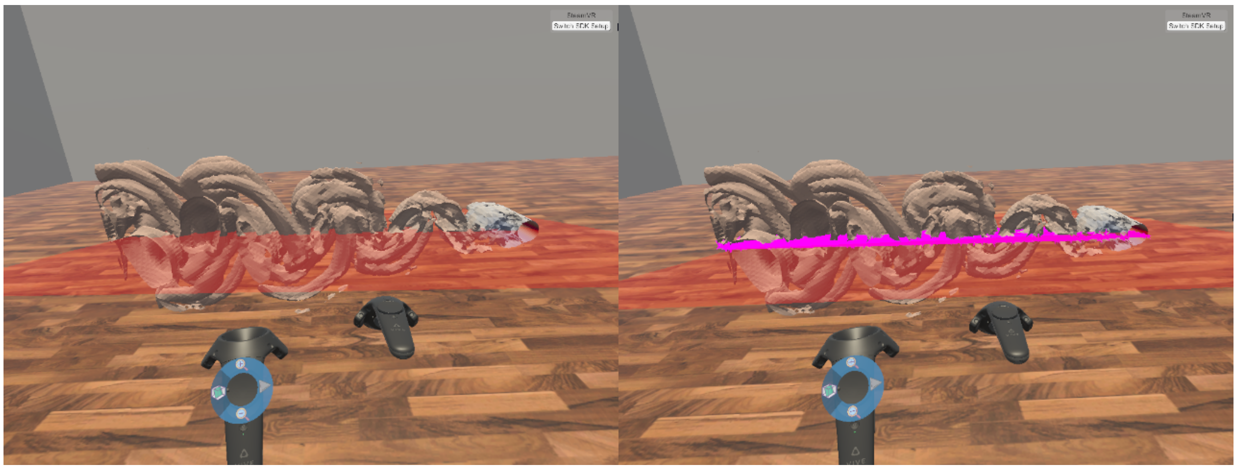

Figure 4.1: Highlight Function before and after pressing the Grip Button

On the Figure 4.1 the left controller shows a menu with the left part selected. This menu is part of the work of Lorenzo D'Eri. Within the application, selecting the left part of the menu makes a red semi-transparent plane appears. The plane is linked to the right controller, so the Transform Component of the plane is the same as the right controller one.

By pressing the grip button, a pink plane is created within the mesh. In fact it is not a plane, but a lot of pink lines linking every vertex of the mesh that were part of a triangle that has been cut by the red plane. This exemple shows the high resolution of the meshes that the application can work with.

The mesh is not convex, it means that there exist at least two points within the mesh that cannot be linked by a straight line without this line going out of the mesh. For short, if a mesh has a hole in it, or if it has a narrow inner angle, then it is convex. This is why the pink lines that are only supposed to link vertices (i.e. that should only follow the mesh) are sometimes going out of the mesh in order to link two vertices.

4.2 Clip Function

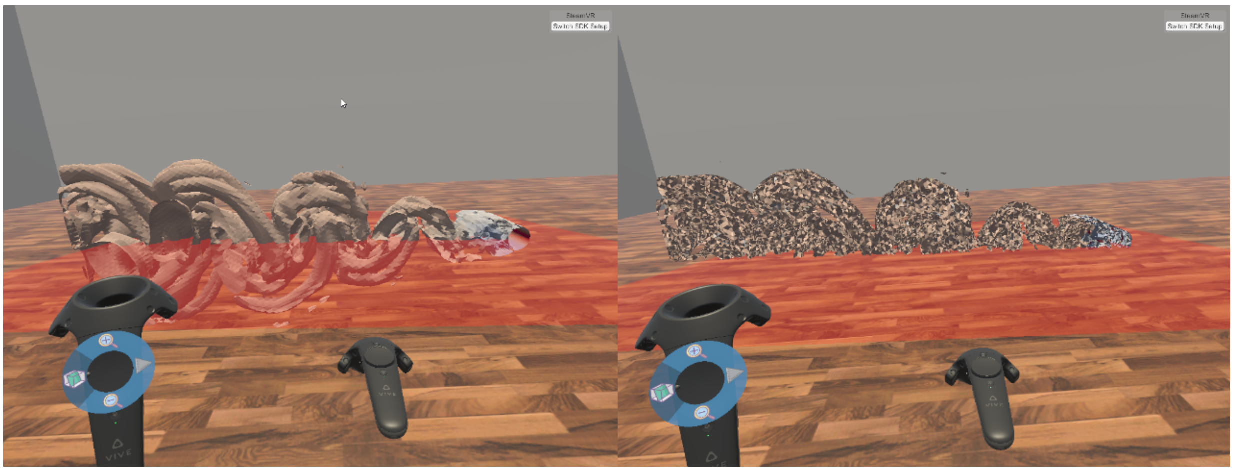

Figure 4.2: Clip Function before and after pressing the Grip Button

On the Figure 4.2 the left controller shows a menu with the left part selected. This menu is part of the work of Lorenzo D'Eri. Within the application, selecting the left part of the menu makes a red semi-transparent plane appears. The plane is linked to the right controller, so the Transform Component of the plane is the same as the right controller one.

By pressing the grip button, the part of the mesh that is behind the plane is removed. As stated earlier, the plane is visible because the user is watching it from its positive side. If the user was placed behind the plane, he would not be able to see the plane. This is helpful in order to know where is the positive side of the plane so the part of the mesh that will be removed is known.

The clipped mesh is not exactly the same as the original one. Indeed, while the vertices and the triangles are exactly the same, the textures are not. No research has been done on this topic since dealing with the textures and shaders is harder and would have required a lot of time.

Chapter 5

Conclusions

5.1 Objectives

The goal of this project was to create a VR interface for CFD data visualization. Given that there is a lot of possible ways to implement an interface, this report has focused on the implementation of direct interaction with the mesh instead of some kind of HUD. Paraview is a scientific data visualization software that has only few VR capabilities but the possibility to work with plug-ins. The ParaUnity plug-in has been used to transfer CFD data to a Unity application. Unity is a powerful and well documented game engine for video-game and simulation development. A Unity application has been created in order to visualize the CFD data from Paraview through ParaUnity plug-in. Several forms of interaction have been developed in this application, everyone of them uses a semi-transparent plane as a spacial cue:

- A **Highlight function** has been implemented for testing purposes, linking the vertices belonging to triangles that have been cut by the plane.
- The **Paraview Clip function** has been implemented, that keeps only the part of the

mesh that is in front of the visible side of the plane (its positive side). Although the textures & shaders issue is still to be fixed, the vertices and triangles are at the right place.

- The **Undo function** implemented within the clip function has been developed in order to simplify the interaction with the mesh. It stores the consecutive vertices and triangles of each clip function in order to be able to cancel changes. After several tests, the function still suffers from some bugs.

This project was the opportunity to work with a quite new technology that redefines the human-machine interaction, to try re-creating Paraview functions in a new Unity application and even improving them with the well known Undo feature. Though the results are not perfect in term of visualization, it has been a great opportunity to learn how to handle meshes in Unity.

5.2 Future Work

There is a lot of ways to improve the application, but before talking about what can be added, it is necessary to fix what is still not working perfectly, and to improve what could become useful:

- *Rework the Undo function:* This implies changing the way of storing the vertices and triangles. It is possible to change the Clip function in order to keep the GameObject that are being cut and store them into a List before creating a clone of it, disabling the original GameObject and then compute the changes on its clone. The Undo function would then only look for the second last stored GameObject in order to enable it again before deleting the last GameObject.

- *Improve the Highlight function:* Executing the Highlight function while the semi-transparent plane is triggered by the GameObject would improve the visual cue since the mesh would always be highlighted in order to ease the visualization of what the Clip function should cut.

- *Implement the Slice function:* some research has been done on the implementation of this function, but no full implementation of it is ready. The general idea is following the principle of the Highlight function that stores the vertices belonging to a triangle that has been cut by the plane. Then it would be necessary to store the triangles in a same way as the Clip function, by creating a map array from the old vertices index to the new one in order to construct a corresponding new triangles list.

The biggest problem of this project was the textures & shaders, and this would be the first thing to look at, after the few points listed above. By understanding the mechanics of these two parameters, the visualization would become much clearer.
Paraview is a huge application with a lot of different functions made to easily explore scientific data visualizations, and thus constitutes an impressive source of ideas for future implementations of the Unity application.

Appendix A

How to successfully build and install ParaUnity

What follows has been written by Lorenzo D'Eri, who has worked on the improvement of an already existing plug-in.

A.1 Building:

A.1.1 Prerequisites

- CMake 3.8.1
- Visual Studio 2015 x64 Community Edition

A.1.2 Compile Qt 4.8.6

Note: if you have trouble with the compilation, have a look here or here

The files in /Qt 4.8.6 are a patched version of Qt (see previous links; I used python's patch.py and the .diff file in that folder) that lets you compile it with Visual Studio 2015 x64. In order to build it do the following:

- Move the content of Qt 4.8.6 in C:\Qt\4.8.6.
- Open the VS2015 x64 Native Tools Command Prompt from Start
- cd C:\Qt\4.8.6
- ./configure.exe -make nmake -platform win32-msvc2015 -prefix c:\Qt\4.8.6 -opensource -confirm-license -nomake examples -nomake tests -nomake demos -debug-and-release
- nmake
- nmake install
- Add C:\Qt\4.8.6\bin to Path

A.1.3 Compile Paraview 5.2.0

I decided to use version 5.2.0 because it was the latest that supported Qt4 by default. It could probably work with the latest as well, but I wouldn't count on that. In order to build it do the following:

- Open CMake with sources in \ParaView-v5.2.0 and build in \ParaView-v5.2.0\build
- Configure with Visual Studio 14 2015 Win64 as a generator
- Check that PARAVIEW_QT_VERSION is 4 and that QT_QMAKE_EXECUTABLE points to C:/Qt/4.8.6/bin/qmake.exe. Configure again.
- Generate
- Open with VS2015

- Build

A.1.4 Compile ParaUnity Plugin

This is a plugin for ParaView that comes from here. It is developed for Qt 4.8 and ParaView 5, and that is the main reason why we are using those older versions instead of Qt5 and ParaView 5.4.
Note: I tried to make it work with ParaView 5.4, the plugin builds and loads, but no data is passed from ParaView to Unity, and debugging is probably not worth it.
In order to build ParaUnity do the following:

- Open a terminal in \ParaUnity\Unity3DPlugin
- mkdir build
- cd build
- cmake -G "Visual Studio 14 2015 Win64" -DParaView_DIR="<PARAVIEW_DIR>\build" .. (<PARAVIEW_DIR>is most likely ../../../../ParaView-5.2.0)
- Open \ParaUnity\ParaView\Unity3DPlugin\build\Project.sln in Visual Studio
- Right click on the project Unity3D, go to C/C++ >Additional Include Directories and add C:\Qt\4.8.6\include\QtNetwork
- Build
- You now have some files (most importantly a Unity3D.dll file) in /build/Debug. Remember their location.

A.2 Running with a scene

A.2.1 Prerequisites

- A prepared Unity scene with the appropriate GameObjects. Mine is found here.
- Unity 5.6.1f1

A.2.2 Exporting the Unity Scene

This is necessary, and has to be repeated every time there's a change in the Unity scene, as the plugin right now is only able to load a specific exe in the correct folder.
Once the scene is set up, from the Unity IDE do the following:

- File >Build Settings
- Make sure only the scene you are interested in is checked
- Set Target Platform as Windows and Architecture as x86_64.
- Hit build
- Choose the same location as the Unity3D.dll (something like \ParaUnity\Unity3DPlugin\build\Debug)
- Save the file as unity_player.exe. **The filename is important as right now it's hardcoded in the plugin!**

A.2.3 Loading the Plugin into Paraview

This needs to be repeated every time paraview is rebuilt. Otherwise it should be persistent.

- Open ParaView 5.2.0 (from paraview.exe in the \build\bin\Debug folder, or from Visual Studio)

- Go to Tools >Manage Plugins, click Load New and locate Unity3D.dll.
- Open the Dropdown from Unity3D and select Auto Load.

A.2.4 Running the plugin

If everything is set up correctly, the following should give results:

- Load any file in ParaView (e.g. a simple sphere)
- Click the unity button with the P
- You should see your unity scene with the ParaView object in the middle.

Appendix B

Source Code pre-ParaUnity

B.1 slicingPlaneBehaviour2.cs

```
  using System.Collections;
using System.Collections.Generic;
using UnityEngine;

/**
 * Script attached to the slicing plane GameObject. It
    ↪ controls his behavior
 *
 * @author Remi Nabonne
 **/
public class SlicingPlaneBehaviour2 : MonoBehaviour
{
    private Mesh slicingPlaneMesh;          // MeshFilter
       ↪ Component of the slicing plane (i.e. plane used for
       ↪ the clip function in order to cut a mesh)
    private GameObject gameObjectToSlice;   // Declaration of
       ↪  a GameObject that will be initialized in the
       ↪ OnTriggerEnter(Collider collider) function in order
       ↪ to store the GameObject to cut
    private GameObject trianglesRenderer;   // Declaration of
       ↪  a GameObject on which a LineRenderer will be
       ↪ attached to in order to Highlight the vertices of
       ↪ cut triangles in the HighlightIntersection function

    // Use this for initialization
    void Awake()
    {
        Globals.slicingPlane = gameObject;
        //gameObject.SetActive(false);
        gameObject.SetActive(true);
    }

    // Update is called once per frame
    void Update()
    {

    }

    /**
     * @fn private void OnTriggerEnter(Collider collider)
     *
     * Function that defines actions made whenever the
        ↪ slicingPlane Collider Component is triggered by
        ↪ another GameObject Collider Component
     * If it is a controller a message will appear in the
        ↪ debug section (testing purpose)
     * If it is a GameObject it will store it into the
        ↪ gameObjectToSlice variable
     **/
    private void OnTriggerEnter(Collider collider)
    {
        if (collider.gameObject.name == "Controller (left)")
        {
            Debug.Log("Left Controller triggered slicingPlane
                ↪ ");
        }
        else if (collider.gameObject.name == "Controller (
            ↪ right)")
        {
            Debug.Log("Right Controller triggered
                ↪ slicingPlane");
        }

        // Because no Paraview data can be loaded on this
            ↪ version of the application, it has been tested
            ↪ on a Unity mesh renamed "Cube"
        else if (collider.gameObject.name == "Cube")
        {
            Debug.Log("SlicingPlane triggered the Cube");
            // Initialization of the GameObject to cut when
               ↪ its collider is triggering the collider of
               ↪ the slicingPlane
            gameObjectToSlice = collider.gameObject;
        }
    }

    /**
     * @fn private void OnTriggerExit(Collider colldier)
     *
     * Function that defines actions made whenever a
        ↪ GameObject collider is leaving the collider of the
        ↪ slicingPlane
     * If it is a controller a message will appear in the
        ↪ debug section (testing purpose)
     * If it is a GameObject it will reset the
        ↪ gameObjectToSlice variable to "null"
     **/
    private void OnTriggerExit(Collider collider)
    {
        if (collider.gameObject.name == "Controller (left)")
        {
            Debug.Log("Left Controller trigger exited from
               ↪ slicingPlane");
        }
        else if (collider.gameObject.name == "Controller (
           ↪ right)")
        {
            Debug.Log("Right Controller trigger exited from
               ↪ slicingPlane");
        }
        else if (collider.gameObject.name == "Cube")
        {
            Debug.Log("Slicing Plane Trigger Exited from cube
               ↪ ");
            // Reseting the GameObject to cut in order not to
               ↪  cut it when pressing the grip button while
               ↪ the plane is not in contact with it
            gameObjectToSlice = null;
        }
    }

    /**
     * @fn public void highlightIntersection()
     *
     * Function that draws lines between specific vertices of
        ↪  the gameObjectToSlice mesh
     * Attach a LineRenderer Component to the
        ↪ trianglesRenderer GameObject. This Component draws
        ↪ lines between points
     * Stores vertices belonging to a triangle that has been
        ↪ cut by the slicingPlane into the List of Vector3
        ↪ verticesToHighlight
     * Draws lines between each consecutive vertex of the
        ↪ verticesToHighlight List
     **/
    public void highlightIntersection()
    {
        // Checks if the GameObject to highlight is triggered
           ↪  by the slicingPlane
        if (gameObjectToSlice != null)
        {
           // Checks if a trianglesRenderer GameObject
              ↪ already exists. If yes, destroys it
            if (trianglesRenderer != null)
                Destroy(trianglesRenderer);

            // Create a new trianglesRenderer GameObject and
               ↪ attach a LineRenderer Component to it
            trianglesRenderer = new GameObject("
               ↪ trianglesRenderer");
            trianglesRenderer.AddComponent<LineRenderer>();
            LineRenderer lineRenderer = trianglesRenderer.
               ↪ GetComponent<LineRenderer>();

            // Disable the shadows made by the drawn lines
               ↪ and sets the line width
            lineRenderer.shadowCastingMode = UnityEngine.
               ↪ Rendering.ShadowCastingMode.Off;
        lineRenderer.receiveShadows = false;
        lineRenderer.widthMultiplier = 0.01f;

        // Gets the MeshFilter Component of the
           ↪ gameObjectToSlice, its vertices and
           ↪ triangles
        Mesh meshToSlice = gameObjectToSlice.GetComponent
           ↪ <MeshFilter>().mesh;
        Vector3[] verticesToSlice = meshToSlice.vertices;
        int[] trianglesToSlice = meshToSlice.triangles;

        // Initialize the Vector3 List that will store
           ↪ the vertices between wich a line is to be
           ↪ drawn
        List<Vector3> verticesToHighlight = new List<
           ↪ Vector3>();

        // Gets the MeshFilter Component of the
           ↪ slicingPlane and its vertices in order to
           ↪ create an infinite invisible Plane with the
           ↪ help of the Unity Plane structure
        Mesh slicingPlaneMesh = gameObject.GetComponent<
           ↪ MeshFilter>().sharedMesh;
        Vector3[] slicingPlaneVertices = slicingPlaneMesh
           ↪ .vertices;
```

```
// A Unity Plane is created with 3 points on the
   ↪ same plane as the plane to be created
var p1 = gameObject.transform.TransformPoint(
   ↪ slicingPlaneVertices[110]);
var p2 = gameObject.transform.TransformPoint(
   ↪ slicingPlaneVertices[65]);
var p3 = gameObject.transform.TransformPoint(
   ↪ slicingPlaneVertices[0]);
var myPlane = new Plane(p1, p2, p3);

/*
lineRenderer.positionCount = 3;
lineRenderer.SetPosition(0, gameObject.transform.
   ↪ TransformPoint(slicingPlaneVertices[110]));
lineRenderer.SetPosition(1, gameObject.transform.
   ↪ TransformPoint(slicingPlaneVertices[65]));
lineRenderer.SetPosition(2, gameObject.transform.
   ↪ TransformPoint(slicingPlaneVertices[0]));
*/

//int pos = 0, neg = 0;
// Loops the triangles of the gameObjectToSlice
   ↪ and checks if the plane is cutting them (i.e
   ↪  if their vertices are not all in front of
   ↪ the same side, the slicingPlane is cutting
   ↪ them)
```

```
            for (int i = 0; i < trianglesToSlice.Length-2; i
               ↪ += 3)
            {
                // Transforms the vertices global coordinates
                   ↪  to local coordinates (local to the
                   ↪ slicingPlane)
                var tmpTriangle = gameObjectToSlice.transform
                   ↪ .TransformPoint(verticesToSlice[
                   ↪ trianglesToSlice[i]]);
                var tmpTriangle1 = gameObjectToSlice.
                   ↪ transform.TransformPoint(verticesToSlice
                   ↪ [trianglesToSlice[i + 1]]);
                var tmpTriangle2 = gameObjectToSlice.
                   ↪ transform.TransformPoint(verticesToSlice
                   ↪ [trianglesToSlice[i + 2]]);

                /*
                if (myPlane.GetSide(tmpTriangle) && myPlane.
                   ↪ GetSide(tmpTriangle1) && myPlane.GetSide
                   ↪ (tmpTriangle2))
                {
                    pos += 3;
                }
                if (!myPlane.GetSide(tmpTriangle) && !myPlane
                   ↪ .GetSide(tmpTriangle1) && !myPlane.
                   ↪ GetSide(tmpTriangle2))
                {
```

```
                neg += 3;
            }
            */

            // Checks if the vertices of the triangle are
               ↪  in front of the same side of the
               ↪ slicingPlane.
            // If not, stores them into the
               ↪ verticesToHighlight Vector3 List
            if ((myPlane.GetSide(tmpTriangle) || myPlane.
               ↪ GetSide(tmpTriangle1) || myPlane.GetSide
               ↪ (tmpTriangle2)) && !(myPlane.GetSide(
               ↪ tmpTriangle) && myPlane.GetSide(
               ↪ tmpTriangle1) && myPlane.GetSide(
               ↪ tmpTriangle2)))
            {
                verticesToHighlight.Add(verticesToSlice[
                   ↪ trianglesToSlice[i]]);
                verticesToHighlight.Add(verticesToSlice[
                   ↪ trianglesToSlice[i + 1]]);
                verticesToHighlight.Add(verticesToSlice[
                   ↪ trianglesToSlice[i + 2]]);
            }
        }

        Debug.Log("verticesToHighligh.Count: " +
           ↪ verticesToHighlight.Count);
```

```
            // Indicates the lineRenderer Component how many
               ↪ points are asked to be linked
            lineRenderer.positionCount = verticesToHighlight.
               ↪ Count;
            // Loops through the verticesToHighlight Vector3
               ↪ List to draw a line between each consecutive
               ↪  element of the list (n-1 with n and n with
               ↪ n+1 ...)
            for(int j = 0; j < verticesToHighlight.Count; j
               ↪ ++)
            {
                lineRenderer.SetPosition(j, gameObjectToSlice
                   ↪ .transform.TransformPoint(
                   ↪ verticesToHighlight[j]));
            }
        }
    }

    /**
     * @fn public void Clip()
     *
     * Function that cuts the part of a mesh in two distinct
        ↪ meshes: a positive part (every triangle that is in
        ↪ front of the positive side of the slicingPlane),
        ↪ and a negative part.
     * Stores the vertices of the mesh to clip that are in
        ↪ front of the positive side of the slicingPlane in
        ↪ the Vector3 List posVerticesL, and those who are
        ↪ behind in the Vector3 List negVerticesL
     * Maps the new index of the stored vertices compared to
        ↪ their original index with the int array
        ↪ posVertexIndex and negVertexIndex
     * Stores the triangles of the mesh to clip that are
        ↪ fully in front of the slicingPlane in the int List
        ↪ posTrianglesL and update the index of their
        ↪ vertices with the posVertexIndex int array
     * Stores the triangles of the mesh to clip that are
        ↪ fully behind the the slicingPlane in the int List
        ↪ negTrianglesL and update the index of their
        ↪ vertices with the negVertexIndex int array
     * Updates the mesh vertices and triangles with the
        ↪ posVerticesL Vector3 List and the posTrianglesL int
        ↪  List
     * Updates the mesh clone vertices and triangles with the
        ↪  negVerticesL Vector3 List and the negTriangles int
        ↪  List
     **/
    public void Clip()
    {
        Debug.Log("Clip Function called");
        // Checks if the GameObject to Clip is triggered by
           ↪ the slicingPlane
        if (gameObjectToSlice != null)
        {
            Debug.Log("gameObjectToSlice.name = " +
               ↪ gameObjectToSlice.name);
            Debug.Log("Slicing Plane is within the trigger
               ↪ zone of object to slice");

            // Gets the MeshFilter Component of the
               ↪ gameObjectToSlice, its vertices and
               ↪ triangles
            Mesh meshToSlice = gameObjectToSlice.GetComponent
               ↪ <MeshFilter>().mesh;
            Vector3[] verticesToSlice = meshToSlice.vertices;
            int[] trianglesToSlice = meshToSlice.triangles;

            // Clones the meshToSlice
            Transform clone = clone = ((Transform)Instantiate
               ↪ (gameObjectToSlice.transform, transform.
               ↪ position + new Vector3(0, 0.25f, 0),
               ↪ gameObjectToSlice.transform.rotation));
            Mesh slicedMesh = clone.GetComponent<MeshFilter
               ↪ >().mesh;
            Vector3[] slicedVertices = slicedMesh.vertices;
            int[] slicedTriangles = slicedMesh.triangles;
```

```
// Initialize the positive part of the mesh
   ↪ vertices and triangles
// Initialize the positive vertices index map int
List<Vector3> posVerticesL = new List<Vector3>();
int[] posVertexIndex = new int[verticesToSlice.
   ↪ Length];
List<int> posTrianglesL = new List<int>();

// Initialize the negative part of the mesh
   ↪ vertices and triangles
// Initialize the negative vertices index map
List<Vector3> negVerticesL = new List<Vector3>();
int[] negVertexIndex = new int[verticesToSlice.
   ↪ Length];
List<int> negTrianglesL = new List<int>();

// Gets the slicingPlane MeshFilter Component in
   ↪ order to create an infinite invisible plane
   ↪ with the Unity Plane structure
Mesh slicingPlaneMesh = gameObject.GetComponent<
   ↪ MeshFilter>().sharedMesh;
Vector3[] slicingPlaneVertices = slicingPlaneMesh
   ↪ .vertices;

// Creates the Plane
var p1 = gameObject.transform.TransformPoint(
   ↪ slicingPlaneVertices[110]);
```

```
            var p2 = gameObject.transform.TransformPoint(
               ↪ slicingPlaneVertices[65]);
            var p3 = gameObject.transform.TransformPoint(
               ↪ slicingPlaneVertices[0]);
            var myPlane = new Plane(p1, p2, p3);

            Debug.DrawLine(p1, p2, Color.red);
            Debug.DrawLine(p2, p3, Color.green);
            Debug.DrawLine(p3, p1, Color.blue);


            int i = 0;
            // Loops through the mesh vertices and stores
               ↪ them either in the posVerticesL or the neg
               ↪ one depending on the side of the Plane they
               ↪ are on
            while (i < verticesToSlice.Length)
            {
                // buffer that stores the vertices local
                   ↪ coordinates into the tmpVertices var (
                   ↪ local to the slicingPlane)
                var tmpVertices = gameObjectToSlice.transform
                   ↪ .TransformPoint(verticesToSlice[i]);

                // Checks the side of the plane the vertex is
                   ↪  on
                if (myPlane.GetSide(tmpVertices))
```

```
          {
              // If on the positive side, stores it
                 ↪ into the posVerticesL Vector3 List
                 ↪ and stores its new index into the
                 ↪ posVertexIndex at its old index
              posVertexIndex[i] = posVerticesL.Count;
              posVerticesL.Add(verticesToSlice[i]);

              // This vertex is not on the negVerticesL
                 ↪  Vector3 List, it has no index in it
              negVertexIndex[i] = -1;
          }
          else if(!myPlane.GetSide(tmpVertices))
          {
              // If on the negative side, stores it
                 ↪ into the pnegVerticesL Vector3 List
                 ↪ and stores its new index into the
                 ↪ negVertexIndex at its old index
              negVertexIndex[i] = negVerticesL.Count;
              negVerticesL.Add(verticesToSlice[i]);

              // This vertex is not on the posVerticesL
                 ↪  Vector3 List, it has no index in it
              posVertexIndex[i] = -1;
          }
          i++;
      }
```

```
        // Loops through the triangles and checks if they
           ↪  are fully positive or negative (i.e if all
           ↪ their vertices are on the same side of the
           ↪ slicingPlane)
        for(int j = 0; j < trianglesToSlice.Length-2; j
           ↪ += 3)
        {
            // buffers that store the triangle vertices
               ↪ local coordinates into the tmpTriangle
               ↪ var (local to the slicingPlane)
            var tmpTriangle = gameObjectToSlice.transform
               ↪ .TransformPoint(verticesToSlice[
               ↪ trianglesToSlice[j]]);
            var tmpTriangle1 = gameObjectToSlice.
               ↪ transform.TransformPoint(verticesToSlice
               ↪ [trianglesToSlice[j + 1]]);
            var tmpTriangle2 = gameObjectToSlice.
               ↪ transform.TransformPoint(verticesToSlice
               ↪ [trianglesToSlice[j + 2]]);

            // Checks if all the triangle vertices are in
               ↪  front of the slicingPlane. If yes,
               ↪ stores them into the posTrianglesL int
               ↪ List but update them with the new index
               ↪ of their vertices in the posVerticesL
               ↪ Vector3 List
```

```
if (myPlane.GetSide(tmpTriangle) && myPlane.
   ↪ GetSide(tmpTriangle1) && myPlane.GetSide
   ↪ (tmpTriangle2))
{
    posTrianglesL.Add(posVertexIndex[
       ↪ trianglesToSlice[j]]);
    posTrianglesL.Add(posVertexIndex[
       ↪ trianglesToSlice[j + 1]]);
    posTrianglesL.Add(posVertexIndex[
       ↪ trianglesToSlice[j + 2]]);
}
// Checks if all the triangle vertices are
   ↪ behind the slicingPlane. If yes, stores
   ↪ them into the negTrianglesL int List but
   ↪  update them with the new index of their
   ↪  vertices in the negVerticesL Vector3
   ↪ List
else if (!myPlane.GetSide(tmpTriangle) && !
   ↪ myPlane.GetSide(tmpTriangle1) && !
   ↪ myPlane.GetSide(tmpTriangle2))
{
    negTrianglesL.Add(negVertexIndex[
       ↪ trianglesToSlice[j]]);
    negTrianglesL.Add(negVertexIndex[
       ↪ trianglesToSlice[j + 1]]);
    negTrianglesL.Add(negVertexIndex[
       ↪ trianglesToSlice[j + 2]]);
```

```
                }
            }

            Debug.Log("PosTrianglesCount: " + posTrianglesL.
               ↪ Count);
            Debug.Log("PosVerticesCount: " + posVerticesL.
               ↪ Count);

            Debug.Log("NegTrianglesCount: " + negTrianglesL.
               ↪ Count);
            Debug.Log("NegVerticesCount: " + negVerticesL.
               ↪ Count);

            // Update the meshToSlice with the positive
               ↪ vertices and triangles
            meshToSlice.triangles = posTrianglesL.ToArray();
            meshToSlice.vertices = posVerticesL.ToArray();
            meshToSlice.RecalculateBounds();

            // Update the meshToSlice clone with the negative
               ↪  vertices and triangles
            slicedMesh.triangles = negTrianglesL.ToArray();
            slicedMesh.vertices = negVerticesL.ToArray();
            slicedMesh.RecalculateBounds();
        }
    }
}
```

B.2 WandController.cs

```
  using System.Collections;
using System.Collections.Generic;
using UnityEngine;

/**
 * Script attached to both controllers GameObject (right and
    ↪ left) that describes their behavior
 *
 * @author Remi Nabonne
 **/
public class WandController : MonoBehaviour
{
    // Declares the button of the controllers
    private Valve.VR.EVRButtonId gripButton = Valve.VR.
       ↪ EVRButtonId.k_EButton_Grip;
    private Valve.VR.EVRButtonId triggerButton = Valve.VR.
       ↪ EVRButtonId.k_EButton_SteamVR_Trigger;
    private Valve.VR.EVRButtonId touchPad = Valve.VR.
       ↪ EVRButtonId.k_EButton_SteamVR_Touchpad;

    // Declares the Controller
    private SteamVR_Controller.Device controller { get {
       ↪ return SteamVR_Controller.Input((int)trckdObj.index)
       ↪ ; } }
    private SteamVR_TrackedObject trckdObj;
```

```
// Declares the GameObject that might be grabbed by the
   ↪ controller
private GameObject pickup;

// Declares the slicingPlane that might be grabbed by the
   ↪  controller
private GameObject slicingPlane;

//private SlicingPlaneBehaviour planeBehaviour;

// Declares the script of the slicingPlane used by the
   ↪ highlightIntersection and Clip functions
private SlicingPlaneBehaviour2 planeBehaviour;

// Use this for initialization
void Start()
{
    // Initialize the slicingPlane and get its script
       ↪ Component
    slicingPlane = Globals.slicingPlane;
    //planeBehaviour = slicingPlane.GetComponent<
       ↪ SlicingPlaneBehaviour>();
    planeBehaviour = slicingPlane.GetComponent<
       ↪ SlicingPlaneBehaviour2>();
    trckdObj = GetComponent<SteamVR_TrackedObject>();
}
```

```
    // Update is called once per frame
    void Update()
    {
        if (controller == null)
        {
            Debug.Log("Controller not Initialized");
        }

        // Describes what to do when the grip button is
           ↪ pressed or released
        if (controller.GetPressDown(gripButton) && !
           ↪ slicingPlane.activeSelf)
        {
            //Debug.Log("Enabling the slicing plane");

            // Makes the slicingPlane appear
            slicingPlane.SetActive(true);
        }
        else if (controller.GetPressDown(gripButton) &&
           ↪ slicingPlane.activeSelf)
        {
            //Debug.Log("Disabling the slicing plane");

            // Makes the slicingPlane disapear
            slicingPlane.SetActive(false);
        }
        // Describes what to do when the touchpad is pressed
            ↪ or released
        if(controller.GetPressDown(touchPad) && slicingPlane.
            ↪ activeSelf)
        {
            // Calls the highlightIntersection function of
                ↪ the slicingPlane script Component
            planeBehaviour.highlightIntersection();

            // IN ORDER TO REPLACE THE HIGHLIGHTINTERSECTION
                ↪ FUNCTION WITH THE CLIP FUNCTION, COMMENT THE
                ↪  LINE ABOVE AND UNCOMMENT THE LINE BELOW
                ↪ THIS LINE

            //planeBehaviour.Clip();
            // Calls the Clip function of the slicingPlance
                ↪ script Component
        }

        // Describes what to do when (the trigger button is
            ↪ pressed or released and the controller collider
            ↪ Component is within the collider Component of
            ↪ another GameObject (pickup))
        if (pickup != null && controller.GetPressDown(
            ↪ triggerButton))
        {
            //Debug.Log("Grip Button is down");
            // Grab interaction, the Transform Component of
               ↪ the pickup GameObject herits of the same
               ↪ parameters values as the controller one
            pickup.transform.parent = this.transform;
        }
        if (pickup != null && controller.GetPressUp(
           ↪ triggerButton))
        {
            //Debug.Log("Grip Button is up");

            // Release interaction, the Transform Component
               ↪ of the pickup GameObject will be constant
               ↪ untill the next grab interaction
            pickup.transform.parent = null;
        }
    }

    // Stores the GameObject that is within the Collider
       ↪ Component of the controller
    private void OnTriggerEnter(Collider collider)
    {
        pickup = collider.gameObject;
    }
    // Reset the pickup GameObject as the controller is no
       ↪ longer within its Collider Component
    private void OnTriggerExit(Collider collider)
    {
        pickup = null;
    }
}
```

B.3 Globals.cs

```
  using System.Collections;
using System.Collections.Generic;
using UnityEngine;



/**
 * Class used to access the slicingPlane script Component
    ↪ from the Controllers script in order to call the
    ↪ highlightInteraction and Clip functions
 *
 * @author Remi Nabonne
 **/
public class Globals {

  public static GameObject slicingPlane;
}
```

Appendix C

Source Code post-ParaUnity

C.1 SlicingPlaneBehaviour.cs

```
  using System.Collections;
using System.Collections.Generic;
using UnityEngine;

/**
 * Script attached to the slicing plane GameObject. It
    ↪ controls his behavior
 *
 * @author Remi Nabonne
 **/
public class SlicingPlaneBehaviour : MonoBehaviour {

    private Mesh slicingPlaneMesh;
       ↪                                           //
       ↪ MeshFilter Component of the slicing plane (i.e.
        ↪ plane used for the clip function in order to cut a
        ↪ mesh)
    private GameObject gameObjectToSlice;
        ↪                                   // Declaration of
        ↪ a GameObject that will be initialized in the
        ↪ OnTriggerEnter(Collider collider) function in order
        ↪ to store the GameObject to cut
    private GameObject gOToSRef;
    private List<List<Vector3>> verticesList = new List<List<
        ↪ Vector3>>();   // Declaration of the 2D vertices
        ↪ List that stores every vertex for the Undo function
    private List<List<int>> trianglesList = new List<List<int
        ↪ >>();          // Declaration of the 2D triangles
        ↪ List that stores evey triangle for the Undo function
    private int meshCount;
        ↪                                                 //
        ↪ Counter of mesh stored into the vertices & triangles
        ↪  2D lists
    private List<GameObject> trianglesRenderers = new List<
        ↪ GameObject>();   // Declaration of the GameObject
        ↪ used in the highlightIntersection function. The
        ↪ LineRenderer Component will be attached to itv

    void Awake () {
        Globals.slicingPlane = this;
        gameObject.SetActive(false);
        meshCount = 0;
    }

    /**
     *  @fn public void Show (ControllerBehaviour controller)
     *
     *  Function called whenever the clip icon in the radial
        ↪ menu is selected.
     *  It prints the red semi-transparent plane to the
        ↪ screen and makes its Transform Component equal to
        ↪ the Controller one (The plane will move exactly
        ↪ like the controller, as if the user had it in its
        ↪ hands)
     **/
    public void Show (ControllerBehaviour controller)
    {
        transform.position = controller.transform.position;
        transform.rotation = controller.transform.rotation;
        transform.parent = controller.transform;
        transform.localPosition += new Vector3(0, 0, 1.05f);
        gameObject.SetActive(true);
    }

    /**
     *  @fn public void Hide (ControllerBehaviour controller)
     *
     *  Function called whenever the clip icon in the radial
        ↪ menu is unselected.
     *  It desactivate the semi-transparent plane
     **/
    public void Hide ()
    {
        gameObject.SetActive(false);
        transform.parent = null;
    }

    /**
     * @fn public bool IsShowing()
     *
     * Function used to get the slicingPlane activity status
        ↪ (visible = active / non visible = non active)
     **/
    public bool IsShowing()
    {
        return gameObject.activeSelf;
    }

    /**
     * @fn private void OnTriggerEnter(Collider collider)
     *
     * Function that defines actions made whenever the
        ↪ slicingPlane Collider Component is triggered by
        ↪ another GameObject Collider Component
     * If it is a GameObject it will store it into the
        ↪ gameObjectToSlice variable
     **/
    private void OnTriggerEnter(Collider collider)
    {
        if (collider.gameObject == Globals.paraviewObj)
        {
            //  Initialization of the GameObject to cut when
               ↪ its collider is triggering the collider of
               ↪ the slicingPlane
            gameObjectToSlice = collider.gameObject;
        }
    }

    /**
     * @fn private void OnTriggerExit(Collider colldier)
     *
     * Function that defines actions made whenever a
        ↪ GameObject collider is leaving the collider of the
        ↪ slicingPlane
     * If it is a GameObject it will reset the
        ↪ gameObjectToSlice variable to "null"
     **/
    private void OnTriggerExit(Collider collider)
    {
        if (collider.gameObject == Globals.paraviewObj)
        {
            // Reseting the GameObject to cut in order not to
               ↪  cut it when pressing the grip button while
            ↪ the plane is not in contact with it
          gameObjectToSlice = null;
      }
  }

  /**
   * @fn public void highlightIntersection()
   *
   * Function that draws lines between specific vertices of
      ↪  the gameObjectToSlice mesh
   * Attach a LineRenderer Component to the
      ↪ trianglesRenderer GameObject. This Component draws
      ↪ lines between points
   * Stores vertices belonging to a triangle that has been
      ↪ cut by the slicingPlane into the List of Vector3
      ↪ verticesToHighlight
   * Draws lines between each consecutive vertex of the
      ↪ verticesToHighlight List
   **/
  public void HighlightIntersection()
  {
      // Checks if the GameObject to highlight is triggered
         ↪  by the slicingPlane
      if (gameObjectToSlice != null)
      {
          // Checks if a trianglesRenderer GameObject
            ↪ already exists. If yes, destroys it
            if (trianglesRenderers.Count > 0)
            {
                foreach (GameObject tr in trianglesRenderers)
                {
                    Destroy(tr);
                }

                trianglesRenderers.Clear();
            }



            foreach(MeshFilter m in gameObjectToSlice.
               ↪ GetComponentsInChildren<MeshFilter>())
            {
                // Create a new trianglesRenderer GameObject
                   ↪ and attach a LineRenderer Component to
                   ↪ it
                GameObject trianglesRenderer = new GameObject
                   ↪ ("trianglesRenderer");
                trianglesRenderer.AddComponent<LineRenderer
                   ↪ >();
                trianglesRenderers.Add(trianglesRenderer);
                LineRenderer lineRenderer = trianglesRenderer
                   ↪ .GetComponent<LineRenderer>();

                // Disable the shadows made by the drawn
                   ↪ lines and sets the line width
```

```
lineRenderer.shadowCastingMode = UnityEngine.
    ↪ Rendering.ShadowCastingMode.Off;
lineRenderer.receiveShadows = false;
lineRenderer.widthMultiplier = 0.01f;

// Gets the MeshFilter Component of the
    ↪ gameObjectToSlice, its vertices and
    ↪ triangles
Mesh meshToSlice = m.mesh; //
    ↪ gameObjectToSlice.GetComponent<
    ↪ MeshFilter>().mesh;
Vector3[] verticesToSlice = meshToSlice.
    ↪ vertices;
int[] trianglesToSlice = meshToSlice.
    ↪ triangles;

// Initialize the Vector3 List that will
    ↪ store the vertices between wich a line
    ↪ is to be drawn
List<Vector3> verticesToHighlight = new List<
    ↪ Vector3>();

// Gets the MeshFilter Component of the
    ↪ slicingPlane and its vertices in order
    ↪ to create an infinite invisible Plane
    ↪ with the help of the Unity Plane
    ↪ structure
```

```
Mesh slicingPlaneMesh = gameObject.
    ↪ GetComponent<MeshFilter>().sharedMesh;
Vector3[] slicingPlaneVertices =
    ↪ slicingPlaneMesh.vertices;

// A Unity Plane is created with 3 points on
    ↪ the same plane as the plane to be
    ↪ created
var p1 = gameObject.transform.TransformPoint(
    ↪ slicingPlaneVertices[110]);
var p2 = gameObject.transform.TransformPoint(
    ↪ slicingPlaneVertices[65]);
var p3 = gameObject.transform.TransformPoint(
    ↪ slicingPlaneVertices[0]);
var myPlane = new Plane(p1, p2, p3);

/*
lineRenderer.positionCount = 3;
lineRenderer.SetPosition(0, gameObject.
    ↪ transform.TransformPoint(
    ↪ slicingPlaneVertices[110]));
lineRenderer.SetPosition(1, gameObject.
    ↪ transform.TransformPoint(
    ↪ slicingPlaneVertices[65]));
lineRenderer.SetPosition(2, gameObject.
    ↪ transform.TransformPoint(
    ↪ slicingPlaneVertices[0]));
```

```
*/

//int pos = 0, neg = 0;

// Loops the triangles of the
   ↪ gameObjectToSlice and checks if the
   ↪ plane is cutting them (i.e if their
   ↪ vertices are not all in front of the
   ↪ same side, the slicingPlane is cutting
   ↪ them)
for (int i = 0; i < trianglesToSlice.Length -
   ↪  2; i += 3)
{
    // Transforms the vertices global
       ↪ coordinates to local coordinates (
       ↪ local to the slicingPlane)
    var tmpTriangle = gameObjectToSlice.
       ↪ transform.TransformPoint(
       ↪ verticesToSlice[trianglesToSlice[i
       ↪ ]]);
    var tmpTriangle1 = gameObjectToSlice.
       ↪ transform.TransformPoint(
       ↪ verticesToSlice[trianglesToSlice[i +
       ↪  1]]);
    var tmpTriangle2 = gameObjectToSlice.
       ↪ transform.TransformPoint(
       ↪ verticesToSlice[trianglesToSlice[i +
    ↪  2]]);

if (myPlane.GetSide(tmpTriangle) &&
    ↪ myPlane.GetSide(tmpTriangle1) &&
    ↪ myPlane.GetSide(tmpTriangle2))
{
    pos += 3;
}
if (!myPlane.GetSide(tmpTriangle) && !
    ↪ myPlane.GetSide(tmpTriangle1) && !
    ↪ myPlane.GetSide(tmpTriangle2))
{
    neg += 3;
}

// Checks if the vertices of the triangle
    ↪  are in front of the same side of
    ↪ the slicingPlane.
// If not, stores them into the
    ↪ verticesToHighlight Vector3 List
if ((myPlane.GetSide(tmpTriangle) ||
    ↪ myPlane.GetSide(tmpTriangle1) ||
    ↪ myPlane.GetSide(tmpTriangle2)) && !(
    ↪ myPlane.GetSide(tmpTriangle) &&
    ↪ myPlane.GetSide(tmpTriangle1) &&
    ↪ myPlane.GetSide(tmpTriangle2)))
{
```

```
                verticesToHighlight.Add(
                    ↪ verticesToSlice[trianglesToSlice
                    ↪ [i]]);
                verticesToHighlight.Add(
                    ↪ verticesToSlice[trianglesToSlice
                    ↪ [i + 1]]);
                verticesToHighlight.Add(
                    ↪ verticesToSlice[trianglesToSlice
                    ↪ [i + 2]]);
            }
        }

        Debug.Log("verticesToHighligh.Count: " +
            ↪ verticesToHighlight.Count);

        // Indicates the lineRenderer Component how
            ↪ many points are asked to be linked
        lineRenderer.positionCount =
            ↪ verticesToHighlight.Count;
        // Loops through the verticesToHighlight
            ↪ Vector3 List to draw a line between each
            ↪  consecutive element of the list (n-1
            ↪ with n and n with n+1 ...)
        for (int j = 0; j < verticesToHighlight.Count
            ↪ ; j++)
        {
```

```
                    lineRenderer.SetPosition(j,
                       ↪ gameObjectToSlice.transform.
                       ↪ TransformPoint(verticesToHighlight[j
                       ↪ ]));
                }
            }
        }
    }

    /**
     * @fn public void Slice()
     *
     * Function that slices a mesh and keeps a thin layer of
        ↪ the mesh that is touching the slicingPlane
     * NOT YET FINISHED, the below description is the general
        ↪  idea of how to implement it
     * Stores vertices belonging to a triangle that has been
        ↪ cut by the slicingPlane into the List of Vector3
        ↪ slicedVertices
     * Maps the new index of the stored vertices compared to
        ↪ their original index with the int array
        ↪ slicedVertexIndex
     * Stores the triangles of the mesh to slice that are cut
        ↪  by the slicingPlane into the slicedTriangles int
        ↪ List and update the index of their vertices with
        ↪ the slicedVertexIndex int array
     * Updates the mesh vertices and triangles with the
        ↪ slicedVertices Vector3 List and the slicedTriangles
        ↪  int List
     **/
    public void Slice()
    {
        if (gameObjectToSlice != null)
        {
            Debug.Log("Slicing Object");

            for (int meshIndex = 0; meshIndex <
               ↪ gameObjectToSlice.GetComponentsInChildren<
               ↪ MeshFilter>().Length; meshIndex++)
            {
                gOToSRef = gameObjectToSlice;
                Mesh meshToSlice = gameObjectToSlice.
                   ↪ GetComponentsInChildren<MeshFilter>()[
                   ↪ meshIndex].mesh;
                Vector3[] verticesToSlice = meshToSlice.
                   ↪ vertices;
                int[] trianglesToSlice = meshToSlice.
                   ↪ triangles;

                verticesList.Add(new List<Vector3>());
                trianglesList.Add(new List<int>());
                for (int vertexIndex = 0; vertexIndex <
                   ↪ verticesToSlice.Length; vertexIndex++)
        verticesList[0].Add(verticesToSlice[
            ↪ vertexIndex]);
    for (int triangleIndex = 0; triangleIndex <
        ↪ trianglesToSlice.Length; triangleIndex
        ↪ ++)
        trianglesList[0].Add(trianglesToSlice[
            ↪ triangleIndex]);
    meshCount ++;

    List<Vector3> slicedVertices = new List<
        ↪ Vector3>();
    int[] slicedVertexIndex = new int[
        ↪ verticesToSlice.Length];
    List<int> slicedTrianglesL = new List<int>();

    Mesh slicingPlaneMesh = gameObject.
        ↪ GetComponent<MeshFilter>().sharedMesh;
    Vector3[] slicingPlaneVertices =
        ↪ slicingPlaneMesh.vertices;

    var p1 = gameObject.transform.TransformPoint(
        ↪ slicingPlaneVertices[110]);
    var p2 = gameObject.transform.TransformPoint(
        ↪ slicingPlaneVertices[65]);
    var p3 = gameObject.transform.TransformPoint(
        ↪ slicingPlaneVertices[0]);
    var myPlane = new Plane(p1, p2, p3);
```

```
Debug.DrawLine(p1, p2, Color.red);
Debug.DrawLine(p2, p3, Color.green);
Debug.DrawLine(p3, p1, Color.blue);

for (int i = 0; i < trianglesToSlice.Length -
   ↪ 2; i += 3)
{
    var tmpTriangle = gameObjectToSlice.
       ↪ transform.TransformPoint(
       ↪ verticesToSlice[trianglesToSlice[i
       ↪ ]]);
    var tmpTriangle1 = gameObjectToSlice.
       ↪ transform.TransformPoint(
       ↪ verticesToSlice[trianglesToSlice[i +
       ↪  1]]);
    var tmpTriangle2 = gameObjectToSlice.
       ↪ transform.TransformPoint(
       ↪ verticesToSlice[trianglesToSlice[i +
       ↪  2]]);

    if ((myPlane.GetSide(tmpTriangle) ||
       ↪ myPlane.GetSide(tmpTriangle1) ||
       ↪ myPlane.GetSide(tmpTriangle2)) && !(
       ↪ myPlane.GetSide(tmpTriangle) &&
       ↪ myPlane.GetSide(tmpTriangle1) &&
       ↪ myPlane.GetSide(tmpTriangle2)))
```

```
                    {
                        slicedVertices.Add(verticesToSlice[
                           ↪ trianglesToSlice[i]]);
                        slicedVertices.Add(verticesToSlice[
                           ↪ trianglesToSlice[i + 1]]);
                        slicedVertices.Add(verticesToSlice[
                           ↪ trianglesToSlice[i + 2]]);

                        slicedVertexIndex
                    }
                }

                for (int i = 0; i < )
            }
        }
    }

    /**
     * @fn public void Clip()
     *
     * Function that cuts a mesh and keeps the part that is
        ↪ in front of a red semi-transparent plane.
     * Stores the vertices of the mesh to clip that are in
        ↪ front of the positive side of the slicingPlane in
        ↪ the Vector3 List posVerticesL
     * Maps the new index of the stored vertices compared to
        ↪ their original index with the int array
        ↪ posVertexIndex
     * Stores the triangles of the mesh to clip that are
        ↪ fully in front of the slicingPlane in the int List
        ↪ posTrianglesL and update the index of their
        ↪ vertices with the posVertexIndex int array
     * Updates the mesh vertices and triangles with the
        ↪ posVerticesL Vector3 List and the posTrianglesL int
        ↪  List
     **/
    public void Clip()
    {
        // Checks if the GameObject to Clip is triggered by
           ↪ the slicingPlane
        // If not, then the Undo algorithm will replace the
           ↪ actual mesh by the last version
        if (gameObjectToSlice != null)
        {
            Debug.Log("Clipping Object");

            for (int meshIndex = 0; meshIndex <
               ↪ gameObjectToSlice.GetComponentsInChildren<
               ↪ MeshFilter>().Length; meshIndex++)
            {
                gOToSRef = gameObjectToSlice;
                // Gets the MeshFilter Component of the
                   ↪ gameObjectToSlice, its vertices and
                   ↪ triangles
```

```
Mesh meshToSlice = gameObjectToSlice.
   ↪ GetComponentsInChildren<MeshFilter>()[
   ↪ meshIndex].mesh;
Vector3[] verticesToSlice = meshToSlice.
   ↪ vertices;
int[] trianglesToSlice = meshToSlice.
   ↪ triangles;

// Stores the actual Mesh vertices into the 2
   ↪ D arrays for the Undo function
verticesList.Add(new List<Vector3>());
trianglesList.Add(new List<int>());
for (int vertexIndex = 0; vertexIndex <
   ↪ verticesToSlice.Length; vertexIndex++)
    verticesList[0].Add(verticesToSlice[
       ↪ vertexIndex]);
for (int triangleIndex = 0; triangleIndex <
   ↪ trianglesToSlice.Length; triangleIndex
   ↪ ++)
    trianglesList[0].Add(trianglesToSlice[
       ↪ triangleIndex]);
meshCount++;

// Initialize the positive part of the mesh
   ↪ vertices and triangles
// Initialize the positive vertices index map
   ↪  int
```

```
List<Vector3> posVerticesL = new List<Vector3
    ↪ >();
int[] posVertexIndex = new int[
    ↪ verticesToSlice.Length];
List<int> posTrianglesL = new List<int>();

// Gets the slicingPlane MeshFilter Component
    ↪  in order to create an infinite
    ↪ invisible plane with the Unity Plane
    ↪ structure
Mesh slicingPlaneMesh = gameObject.
    ↪ GetComponent<MeshFilter>().sharedMesh;
Vector3[] slicingPlaneVertices =
    ↪ slicingPlaneMesh.vertices;

// Creates the Plane
var p1 = gameObject.transform.TransformPoint(
    ↪ slicingPlaneVertices[110]);
var p2 = gameObject.transform.TransformPoint(
    ↪ slicingPlaneVertices[65]);
var p3 = gameObject.transform.TransformPoint(
    ↪ slicingPlaneVertices[0]);
var myPlane = new Plane(p1, p2, p3);

Debug.DrawLine(p1, p2, Color.red);
Debug.DrawLine(p2, p3, Color.green);
Debug.DrawLine(p3, p1, Color.blue);
```

```
        // Loops through the mesh vertices and stores
            ↪  them either in the posVerticesL or the
            ↪ neg one depending on the side of the
            ↪ Plane they are on
        int i = 0;
        while (i < verticesToSlice.Length)
        {
            // buffer that stores the vertices local
                ↪ coordinates into the tmpVertices var
                ↪  (local to the slicingPlane)
            var tmpVertices = gameObjectToSlice.
                ↪ transform.TransformPoint(
                ↪ verticesToSlice[i]);

            // Checks the side of the plane the
                ↪ vertex is on
            if (myPlane.GetSide(tmpVertices))
            {
                // If on the positive side, stores it
                    ↪  into the posVerticesL Vector3
                    ↪ List and stores its new index
                    ↪ into the posVertexIndex at its
                    ↪ old index
                posVertexIndex[i] = posVerticesL.
                    ↪ Count;
                posVerticesL.Add(verticesToSlice[i]);
        }
        else if (!myPlane.GetSide(tmpVertices))
        {
            // This vertex is not on the
               ↪ posVerticesL Vector3 List, it
               ↪ has no index in it
            posVertexIndex[i] = -1;
        }
        i++;
    }

    // Loops through the triangles and checks if
       ↪ they are fully positive (i.e if all
       ↪ their vertices are on the positive side
       ↪ of the slicingPlane)
    for (int j = 0; j < trianglesToSlice.Length -
       ↪  2; j += 3)
    {
        // buffers that store the triangle
           ↪ vertices local coordinates into the
           ↪ tmpTriangle var (local to the
           ↪ slicingPlane)
        var tmpTriangle = gameObjectToSlice.
           ↪ transform.TransformPoint(
           ↪ verticesToSlice[trianglesToSlice[j
           ↪ ]]);
```

```
                var tmpTriangle1 = gameObjectToSlice.
                   ↪ transform.TransformPoint(
                   ↪ verticesToSlice[trianglesToSlice[j +
                   ↪  1]]);
                var tmpTriangle2 = gameObjectToSlice.
                   ↪ transform.TransformPoint(
                   ↪ verticesToSlice[trianglesToSlice[j +
                   ↪  2]]);

                // Checks if all the triangle vertices
                   ↪ are in front of the slicingPlane. If
                   ↪  yes, stores them into the
                   ↪ posTrianglesL int List but update
                   ↪ them with the new index of their
                   ↪ vertices in the posVerticesL Vector3
                   ↪  List
                if (myPlane.GetSide(tmpTriangle) &&
                   ↪ myPlane.GetSide(tmpTriangle1) &&
                   ↪ myPlane.GetSide(tmpTriangle2))
                {
                    posTrianglesL.Add(posVertexIndex[
                       ↪ trianglesToSlice[j]]);
                    posTrianglesL.Add(posVertexIndex[
                       ↪ trianglesToSlice[j + 1]]);
                    posTrianglesL.Add(posVertexIndex[
                       ↪ trianglesToSlice[j + 2]]);
                }
                }

                Debug.Log("triNBR: " + trianglesList[0].Count
                    ↪ );
                Debug.Log("vertNBR: " + verticesList[0].Count
                    ↪ );

                // Update the meshToSlice with the positive
                    ↪ vertices and triangles
                meshToSlice.triangles = posTrianglesL.ToArray
                    ↪ ();
                meshToSlice.vertices = posVerticesL.ToArray()
                    ↪ ;
                meshToSlice.RecalculateBounds();
            }
        }

        // The GameObject to Clip is not triggered by the
            ↪ slicingPlane, so its mesh is replaced by the
            ↪ previous version if there is any
        else if(gameObjectToSlice == null && meshCount > 1)
        {
            for (int meshIndex = 0; meshIndex < gOToSRef.
                ↪ GetComponentsInChildren<MeshFilter>().Length
                ↪ ; meshIndex++)
            {
                Debug.Log("Undo");
                Debug.Log("triNBR: " + trianglesList[0].Count
                   ↪ );
                Debug.Log("vertNBR: " + verticesList[0].Count
                   ↪ );
                Mesh meshToShow = gOToSRef.
                   ↪ GetComponentsInChildren<MeshFilter>()[
                   ↪ meshIndex].mesh;
                meshToShow.vertices = verticesList[meshCount
                   ↪ - 2].ToArray();
                meshToShow.triangles = trianglesList[
                   ↪ meshCount - 2].ToArray();
                //meshToShow.vertices = verticesList[0].
                   ↪ ToArray();
                //meshToShow.triangles = trianglesList[0].
                   ↪ ToArray();
                meshToShow.RecalculateBounds();
            }
        }
    }
}
```

References

[1] Steve Bryson, ”Virtual reality in scientific visualization”, *Communications of the Association for Computing Machinery*, vol.39 Issue 5, p.62-71, May 1996.

[2] Eugenia M. Kolasinsky, ”Simulator Sickness in Virtual Environments”, *Army research inst for the behavioral and social sciences alexandria VA*, 1995.

[3] Mark R. Mine, ”Virtual Environment Interaction Techniques”, *UNC Chapel Hill computer science technical report TR95-018*, p.507248-2, 1995.

[4] Robinett, Warren, and Richard Holloway, ”Implementation of flying, scaling and grabbing in virtual worlds”, *Symposium on Interactive 3D graphics, ACM*, p.189-192, June 1992.

[5] Jacoby, Richard H., and Stephen R. Ellis, ”Using virtual menus in a virtual environment.”, *Symposium on Electronic Imaging: Science and Technology*, p.39-48, 1992.

[6] Ohno, Nobuaki, and Akira Kageyama, ”Scientific visualization of geophysical simulation data by the CAVE VR system with volume rendering”, *Physics of the Earth and Planetary Interiors*, vol.163, no.1, p.305-311, 2007.

[7] Cruz-Neira, Carlina, Daniel J. Sandin, and Thomas A. DeFanti, ”Surround-screen projection-based virtual reality: the design and implementation of the CAVE”, *20th*

annual conference on Computer graphics and interactive techniques, ACM, p.135-142, 1993.

[8] Paraview website: http://www.paraview.org/fluid-dynamics/

[9] Kitware website: https://www.kitware.com/platforms/vtk

[10] HTC Vive website: https://www.vive.com/

[11] Unity3D website: https://unity3d.com/

[12] Unity3D Documentation: https://docs.unity3d.com/

[13] Leslie Lamport, *LaTeX: a document preparation system*. Addison Wesley, Massachusetts, 2nd edition, 1994.