# Visualization Challenge: physical properties for mechanical models

3D Augmented Reality Project Report

Giovanni Gallinaro

A.Y. 2019/2020

## 1 Introduction

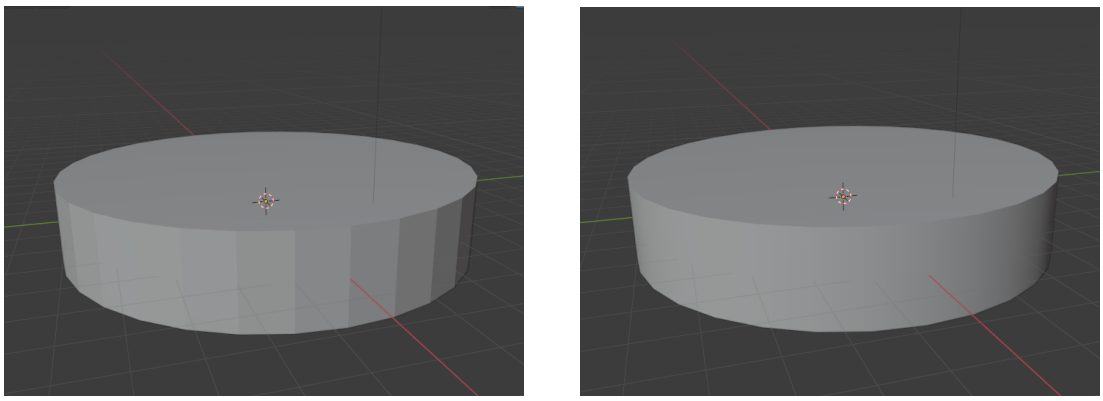
The goal of this project is to create an Unity 3D application which is capable to display the 3D model of a car together with its flow lines. In particular, the aim is to help the final user of the application to visualize its physical properties, such as pressure and aerodynamics, in an efficient and interactive way. For this purpose, the provided dataset from Altair has been used: in the following chapters of the report I will discuss more details on how the supplied files have been used to provide a proper visualization of the 3D data. However, firstly I am going to discuss some details about the implementation of the environment.

## 2 Environment

In order to provide a good looking and an user-friendly application to the final user, an environment has been built besides the interesting data of the car. The idea is to render the setting of the application as a showroom in which the car is displayed at the center.

**Room** To do this, the room, which is circular, has been built using Blender: firstly, a cylinder object has been created and scaled to shape it as a room. Then, in order to visualize the interior of the 3D object (i.e. change the orientation of the faces), the normals have been flipped. This step was necessary since the .obj file imported in Unity, otherwise, would have shown just the exterior walls of the cylinder.

Finally, the angles of the mesh have been smoothed using the Shape Smooth tool from the Object tab of the program and by refining the shape with the Auto Smooth tool, which has been set to 45 degrees. Figure 1 shows the shape before and after the smoothing was applied. Then, the final 3D model has been exported as an .obj file in order to import it in the Unity Scene.



(a) (b)

Figure 1: The cylinder object rendered in Blender before (a) and after (b) smoothing.

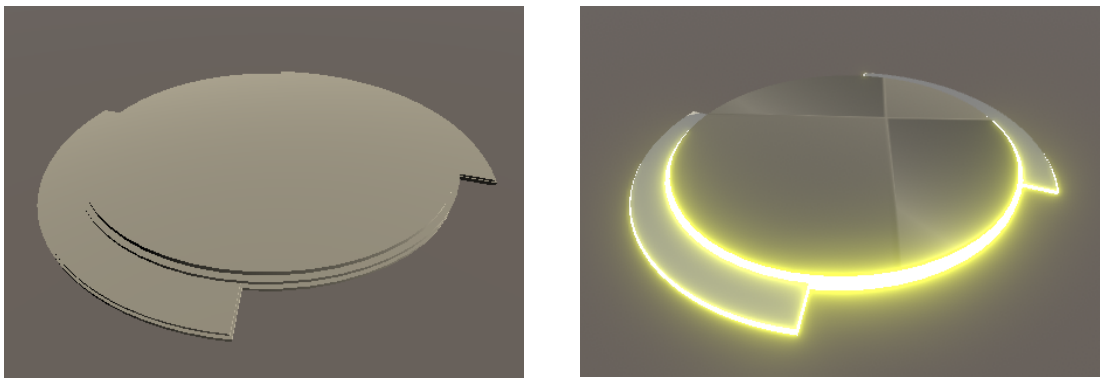
In Unity, the mesh of the room has been coloured by associating a Default material to the object and by setting its colour to white. Furthermore, the normal map of the material has been changed and improved in order for the walls to be reflective with respect the incoming light.

**Platform** A virtual platform, above which the car will be settled, has been placed at the center of the room. The 3D model has been downloaded from cgtrader.com and it consists of three parts: one is the actual support for the car model, one is a support placed on both sides of the central platform, while the last one follows the outlines of the previously mentioned models. Figure 2a shows the model as it was downloaded (without rendering through materials).

**Illumination** Five directional light sources are present in the scene: four are located every 90 degrees near the sides of the room, and they point at the walls in order to increase reflectance. The fifth one is placed at the center of the room and points down, towards where the car will be placed.

Moreover, the last item of the platform model mentioned above has been rendered as an additional light source. To do this, the Post-Processing packet has been downloaded from the Package Manager provided by Unity. This packet provides useful tools to render a Game Object in Unity as an emitting light model through the following steps: firstly, a Post Processing Layer has been added to the Main Camera object. Then, an empty Game Object has been created and a Post Processing Volume component has been added to it. Inside this component, a Bloom effect has been selected: by changing both the *weight* and *intensity* parameters, the amount of light emitted from the platform can be tuned. Finally, the colour of the platform light source material has been set to yellow, setting the intensity to 7.19, and the *emission* box has been ticked.

The final result, shown in Figure 2b and compared to the first look of the model, has a much better looking appearance than before.



(a) (b)

Figure 2: The platform object before (a) and after (b) the addition of the materials and the Post Processing Volume.

# 3 Dataset visualization

In this chapter I will discuss the techniques that have been used to display both the car and the flow lines in the scene. Both are visualized as point clouds and have been extracted by the raw data provided in the dataset.

**Flow lines** The data belonging to the flow lines represent some properties regarding the aerodynamics of the car. The original raw data is contained in the file *flowlines.bin*, which has been converted into a *flows_ata.xyz* file through the provided MATLAB script. This file contains all the 3D coordinates of the points associated to the flow lines, along with its default RGB colour. All the data is separated by commas, and each line corresponds to a different point.

The display of the pointcloud has been managed through the provided script `PointCloudManager.cs`, which has been improved in order to provide a more dynamic and fancy visualization of the data. Firstly, the script loads the data and stores the coordinates into the `Vector3[] Points` array and the RGB values in the `Color[] colors` array. In particular, the colours are assigned according to the designed input choice, which can be between: an arbitrary colour, the default RGB values or a gradient based on the height values (to do this, also the minimum and maximum height values have been computed).

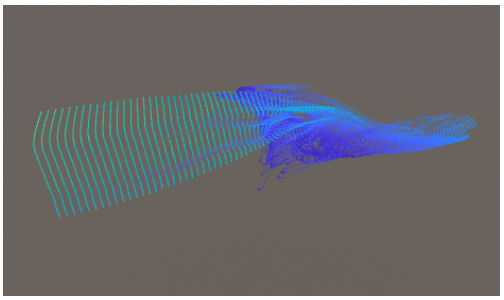


Figure 3: Rendering of the flows.

Furthermore, to allow the script to generate the points in an animated way, the points have been subdivided into an arbitrary number of groups, each one containing $\frac{total\#ofpoints}{\#ofgroups}$ points. The script has been associated to an empty Game Object in Unity, and it creates a flows_ata.xyz child which, in turn, creates one children for each group of points through the *loadXYZ(string dPath)* co-routine. This co-routine gets started from the *loadPointCloud()* function, which, in turn, gets called from *loadScene()* in the *Start()* function. When the data gets loaded, the boolean variable `loaded` is set to true. Once the variable is updated (i.e. all the data is available), some operations can be performed in *Update()*, which gets called once per frame during the execution of the Unity application.

The animation is handled by the *playFrame()* function: if the boolean variable `play` is true, for each frame the script loads a group of points, until all the data is visualized in the scene. By changing the number of groups, it is possible to tune the speed of the animation. Also, by pressing Space on the keyboard during the execution of the application, it is possible to re-play the animated sequence which creates the flow lines: in this case, the *playFrame()* function destroys all the meshes associated to the group of points and creates new ones with the same procedure.

Finally, the created mesh is translated is order to be aligned with the car model. A final look of the data visualization is shown in Figure 3.

**Car** Concerning the 3D model of the car, the *mesh_simple.bin* has been converted into the *mesh_simple.xyz* file through some little modifications on the provided MATLAB script. The *mesh_simple.xyz* file contains the coordinates of all the points of the car model, the normals and the associated pressure value. The dataset gets loaded into the Unity application through the `CarPointCloudManager.cs`, similarly to the approach used for the flow lines. Also in this case, the points gets divided into groups: this time, however, not because the visualization is dynamic, but because mesh objects in Unity can have up to 65535 vertices (the dataset has 486030 points).

The points are displayed and coloured by assigning an RGB value for each pressure data through a gradient (as for the height of the flow lines, also in this case the minimum and maximum pressure values are computed beforehand). Figure 4 shows the default visualization of the car in the scene.

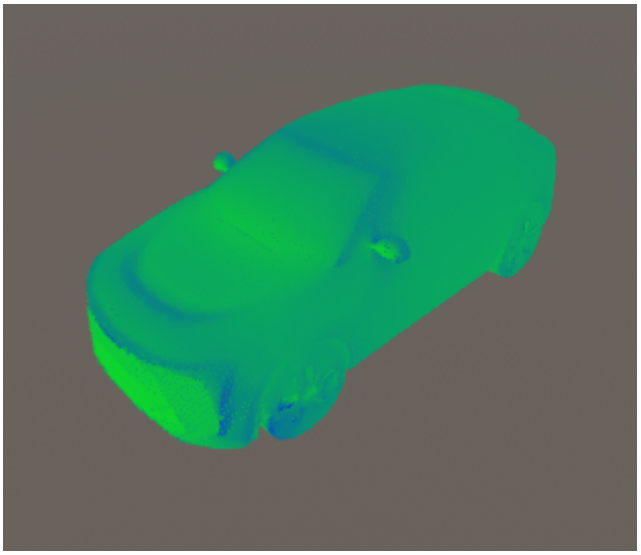


Figure 4: Rendering of the car.

# 4 Interface

After the setting of the 3D scene and the visualization of all the elements, some interactive elements have been added in order for the user to inspect the physical properties. In particular, three UI elements have been implemented: a slider, which allows the user to select the minimum value of pressure to show in the application, a mouseover control, used to allow the user to highlight different areas of the car, and a dropdown menu, which allows to select among different possible colors for the flow lines.

**Slider** Unity allows to create a slider object which gets associated, as a child, to a Canvas object. The Canvas Render Mode option has been set to `Screen Space - Overlay`. In the inspector tab of the Slider, it is possible to associate it to one or more functions from the scripts, in order to change a determined value. In this case, the function *selectPoints(float minPressure)* from the `CarPointCloudManager.cs` has been linked, where the variable *minPressure* is set to be controlled from the slider.

Whenever the user interacts with the slider, only the points with pressure greater of the controlled minimum pressure are shown with the original colour (these points are considered to be "active"), otherwise they are rendered in gray, as shown in Figure 5. This is done by iterating each mesh and by re-colouring each point if its corresponding pressure value is lower than the reference pressure. The temporary colours of the points, which depend on the current slider value, are stored in a array of `Colors` named `tempColors`, so that, when the slider is dragged back, the original colours of the mesh can be recovered. For each mesh, the colours are assigned by setting `mesh.colors` equal to `tempColors`. Also, whenever the reference `minPressure` value changes, the average pressure value among the active points (i.e. those that are greater than `minPressure`) and the percentage of active points, with respect to the total number of vertices, are computed. These parameters are also shown in the bottom left corner of the Canvas through a Text object (Figure 5). The function *displayAvg()* is called in *Update()*.

Finally, a script has been associated to the slider in order to avoid the interference with the interaction of the user with the Main Camera (details are going to be explained better later in this report).

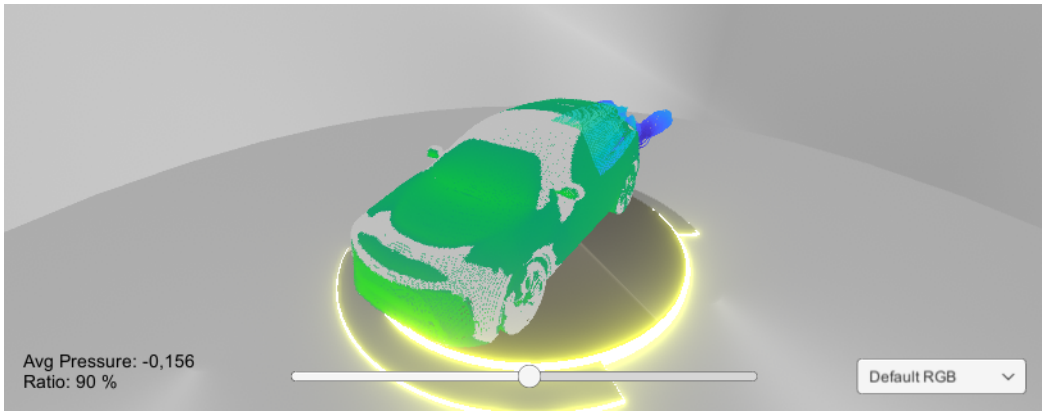



Figure 5: Pointcloud rendered by selecting just a range of points by means of the interactive slider (bottom of the scene). In the bottom left corner of the screen, the average pressure value and the percentage of active points are visualized.

**Mouseover** Another possibility for the user to interact with the data is to hover the mouse over the mesh of the car. By moving the mouse around, different areas of the pointcloud are highlighted and the corresponding average pressure value is shown above it.

This is managed by the *GetMouseInfo()* function, which gets called in *Update()*. It uses the *Camera.ScreenPointToRay(Input.mousePosition)* function to retrieve the position of the cursor on the screen and returns a `Ray` object starting from the camera and going through the mouse position. Then, the ray gets casted by means of the *Physics.Raycast(Vector3 origin, Vector3 direction, float maxDistance)* function in the corresponding direction through infinity and, if the returned value is true, the function checks if the ray hits the car in a certain position. If true, the corresponding point group that has been hit is highlighted, that means, every other group is coloured in gray.

Note that this procedure is possible only when the Game Object has a collider element associated with it: so, when the meshes of the car get instantiated in the script, a *Box Collider* component is also added (Figure 6). At the same time, the average among the highlighted points is computed and it is displayed on screen by creating a new Text object, which position depends on the current position of the cursor.

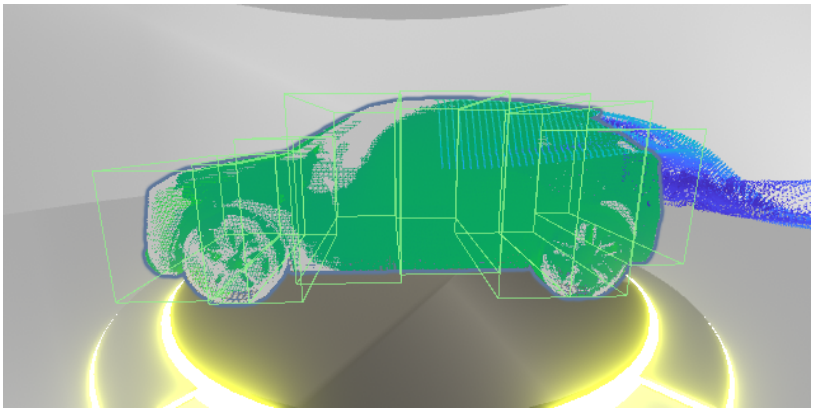


Figure 6: Box collider components are used to detect which of the point groups the user want to interact with.

Whenever the user moves the mouse away from the corresponding point group (to another group or away from the car), the Text object gets destroyed and the points get recoloured according to the new position of the mouse. An example of highlighted rendering of the car is shown in Figure 7.

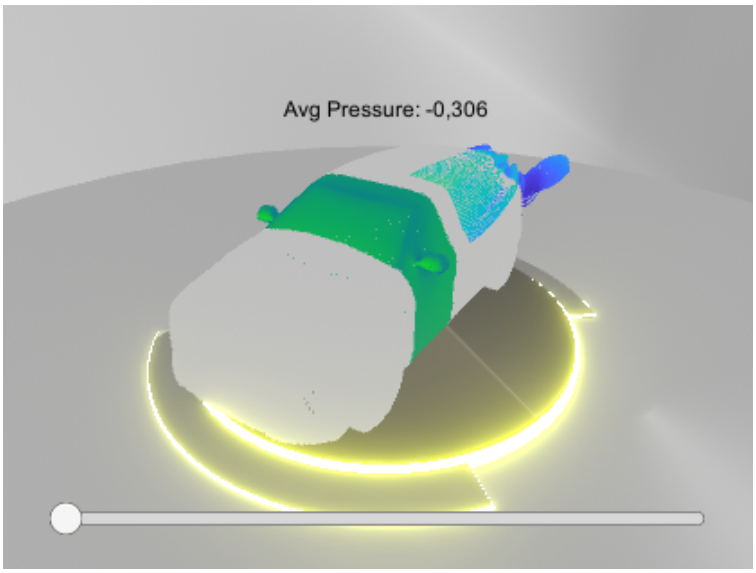
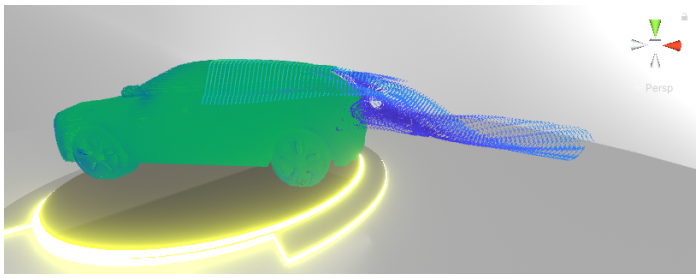



Figure 7: Example of an area selected by the user to visualize the physical properties only within a certain range of coordinates. Above the selected area, a text box indicates the average pressure of the highlighted points.

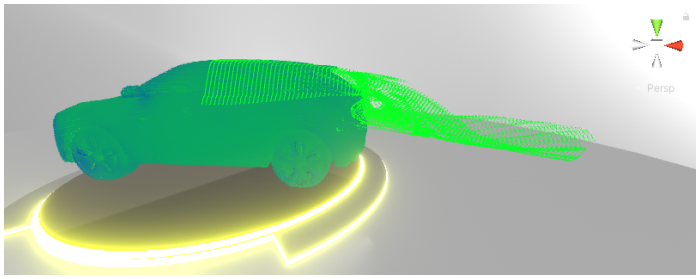
**Dropdown** Similarly to what was done for the Slider, a Dropdown Game Object has been created (Figure 5, bottom right of the screen) and the function *retrieveColor(int colorInd)*, from the `PointCloudManager.cs` script, has been associated to it.

This function takes as argument an index and, depending on its value, decides which colours to apply to the mesh of the flow lines (default RGB colours, and arbitrary input solid colour or colours depending on the height of the points).

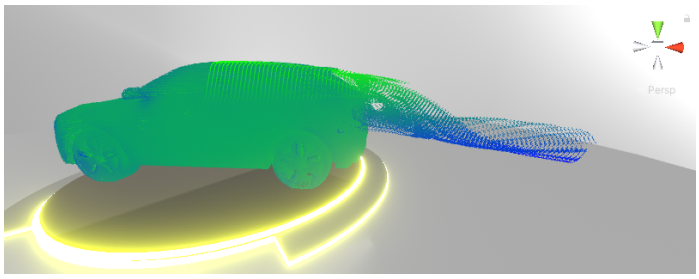
Moreover, it also re-plays the animation of the points with the new colours, chosen by the user, by setting a boolean variable `restart` to true. Figure 8 shows how the flow lines look with the different choices of colour.



(a) default RGB colors (imported from the .xyz file).



(b) selected input color (in this case, green).



(c) points colored by height (gradient).

Figure 8: Flow lines rendered with the different colors. The user can change the appearance of the pointcloud by interacting with the dropdown menu.

# 5 Camera

An important aspect of the application is the possibility for the user to move the camera around the car and check different areas and properties. In particular, in the developed application the user can perform a 360 degrees rotation around the y-axis and a (almost) 90 degrees rotation around the x-axis by holding down and dragging the mouse around the screen. Also, the camera can be moved forward and backward by scrolling with the mouse wheel. All these movements are handled by the `CameraManager.cs` script.

Firstly, the camera has been implemented to point towards the car and an offset has been chosen as distance from the central object. By changing this offset with the scrolling of the mouse, the user can zoom in and out the scene.

In order to perform the rotation, whenever the user holds down the left button of the mouse, the *Input.GetMouseButton(0)* gets called and, if it returns true, the direction of the moving mouse gets estimated. To do this, the current position of the cursor gets computed by means of the *Camera.ScreenToViewportPoint(Input.mousePosition)* function and it gets compared with the previous one. The resulting direction is then multiplied by 180 and the corresponding rotation is obtained. This rotation is given in input to the *Camera.transform.Rotate* function which rotates the camera object. As anticipated before, bounds have been set on the rotation around the x-axis in order to allow the rotation between 0 and 90 degrees.

Also, when the user performs a fast rotation and releases the mouse button, the camera will continue to rotate around the y-axis and will decrease its speed until it slowly reaches 0 (i.e. it stops). This is done by taking into account the last direction value available (last movement of the user) and letting the camera rotate by reducing the speed in half in every frame. This effect will take place only when the speed of the rotation is greater than a certain threshold, otherwise the user would have problems in performing small and precise movements.

So, in summary, when the user rotates the camera slowly to focus its attention to a certain area, the rotation of the camera stops immediately as soon as the user releases the mouse button; on the other hand, when the user wants to perform a big rotation (i.e. greater speed) around the car, like, for example, to change the view side, a residual velocity is also taken into account and the camera continues to rotate a little longer after the user releases the button. Figure 9 shows some examples of camera angles.

The whole rotation system is controlled by a boolean variable `canRotate`. If the variable is false, the system is suspended and the camera is no longer controllable by the user. In particular, *canRotate* can be set to false by the script `Slider.cs`. In fact, this script checks, for each frame, if the mouse is over the slider (and thus the user wants to interact with the UI, not rotate the camera) through the function *EventSystem.current.IsPointerOverGameObject()*. If so, it finds the Main Camera object in the scene and assigns false to the boolean variable of the `CameraManager.cs` script. This procedure prevents the user from simultaneously rotate the camera and move the slider.

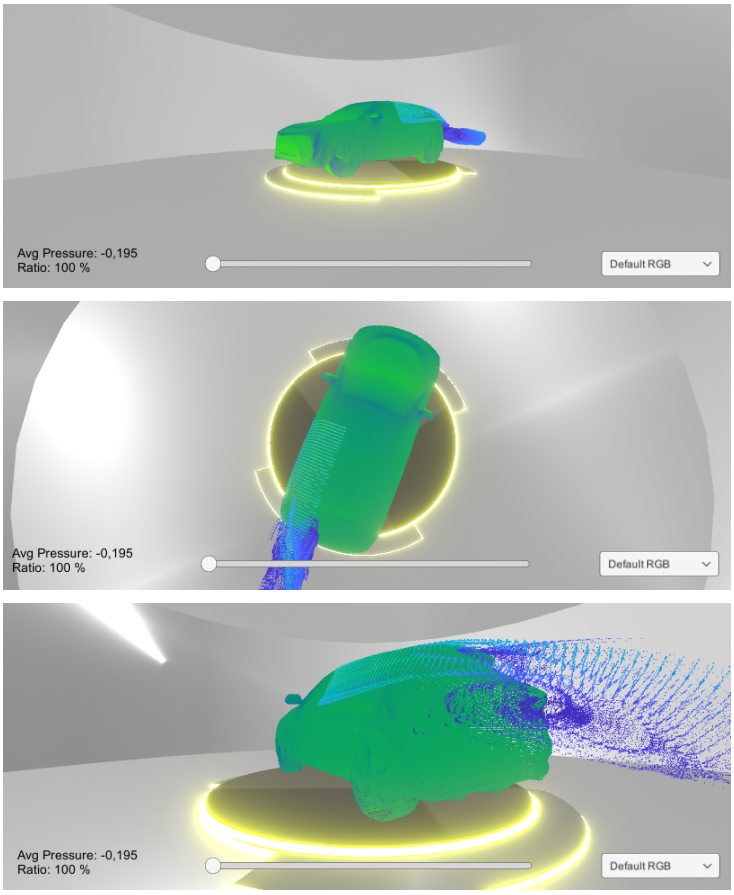



Figure 9: Final look of the application, from different viewpoints.

# 6 Conclusions

This project was very challenging since this was the first time for me developing such a complex Unity application from scratch. However, thanks to the Labs of the course, it was clear from the beginning which were the strategies that I needed to follow in order to carry out certain tasks.

In addition, I also took my time implementing some details that were not necessary for the delivery of the project (e.g. the cylinder from Blender, post processing volumes, ray casting, etc...) but that got me really interested and eager to experiment brand new techniques within the world of 3D applications.

Nevertheless, this challenge leaves open an infinite number of possibilities to further improve the application and add even more interactive elements to the scene.