3D Augmented Reality Course - Final Project A.1 2020 Visualization Challenge: Physical Properties for Mechanical Models

Matteo Moro, Gianmarco Zonta

I. Introduction

Project task: The goal of the project is to develop an application that displays physical properties of a 3D model. Each team is provided with the needed source material, which contains (encoded) data on the model geometries, flowlines positions and surface pressure values. The visualization app should be built inside the Unity3D game engine.

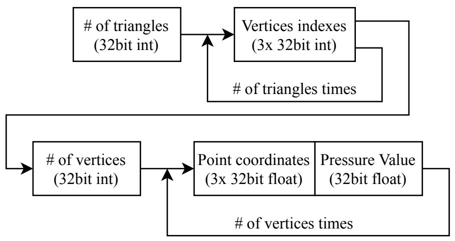
Our approach: The application presented in this report is desktop-based. This choice has been made to obtain a higher display accuracy of the involved geometries, considering the engineering field in which it should be deployed. The task has been performed using some external tools: Python, for the initial data processing, and a combination of *Meshlab* and *Blender*, both 3D manipulation software used for mesh simplification; Among the given files, *mesh.bin*, *flowlines.bin* have been used for the project, interpreted using the *Readme.txt* structure definition.

Report structure: The presented sections describe in detail the project steps, divided in a model processing, flowlines and pressure points visualization and a description of the scene composition. The final *results* section provides an analysis of the overall system performance and some example images.

II. Model Processing

The initial model processing phase is composed by the following steps:

Data extraction: The *mesh.bin* file has been interpreted using a Python script, the aim was extracting the full model description: vertices, faces and normals. This would constitute the basis over which the input data is displayed, so having a detailed, yet efficient, representation was crucial. The source encodes the mesh in a custom binary structure defined in the *Readme.txt* file, summarized in the scheme in figure 1.



# of triangles (32bit int)
Vertices indexes (3x 32bit int)
# of triangles times
# of vertices (32bit int)
Point coordinates (3x 32bit float)
Pressure Value (32bit float)
# of vertices times

Fig. 1: Model binary file structure

After the extraction, the second part of the script builds up a Polygon File Format (PLY) 3D object file, containing 8.9M vertices and 13.3M faces, following the specifications of the standard.

Mesh simplification: Using the tools provided by *Meshlab*, the portions of the model that are not displayed in the final application, mostly the internal components, are removed. This operation is performed using the Ambient Occlusion filter, which allow to assign a quality metric to each vertex, based on the amount of light (coming from an outside moving source) that would hit that component. All the vertices below a specified quality threshold (0.001) has been deleted, alongside the adjacent faces. After the deletion of duplicated faces, vertices, isolated connected components (up to 10K faces), the model has 2.5M vertices, 4.8M faces and retains unvaried all the external visible mesh. At this stage the face normals orientation is checked as wrong and recomputed. The overall model positioning is corrected with a 90° rotation in the x axis.

Model fine tuning and further decimation: The final processing is performed within *Blender*, using advanced visualization and correction aids. This step was mainly focused on the correction of model errors operating at single vertex level. In the same context also the decimation of the mesh in areas of the surface that does not need high level of detail is performed (*Decimate Geometry*, *Degenerate Dissolve* commands), namely the lower part of the car body, the flat surfaces on the top, sides and rear. The final model (presented in figure 2) has 396K vertices and 688K faces and has been exported as a Unity3D-recognized Wavefront OBJ file.

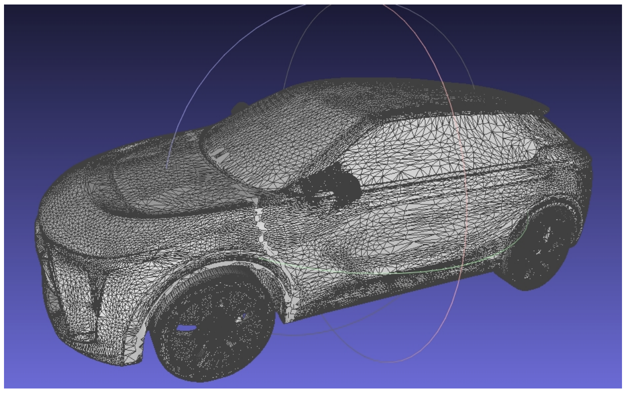


Fig. 2: Car model after processing - mesh

III. Flowlines: Processing and Visualization

Data refactoring: The data concerning the flowlines simulation is stored in the *flowlines.bin* file, following the file structure presented in figure 3.

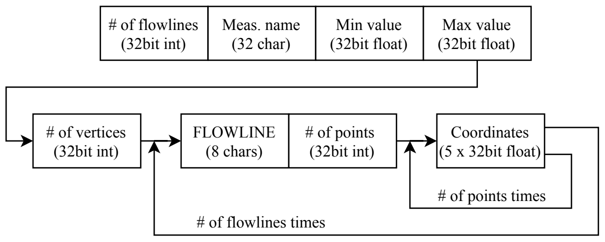


Fig. 3: Flowline binary file structure

A Python script has been designed to extract the data and export individual flowlines as binary files, saved in the *Flowlines* folder. The found 603 flowlines hold a variable number of data points (up to 125) each being a tuple of coordinates containing, in this order, the 3D (x, y, z) position, a time descriptor and current flow velocity.

Flowlines management: The flowline binary data is imported in the Unity3D project. The high level flowlines management is scripted in C# and assigned to an empty GameObject placed in the scene. The purposes of this script and the main manipulation options are:

- **Instantiation:** every flowline object is added to the scene by creating a children GameObject with the corrent flowline ID. This operation is performed automatically at the start of the simulation.
- **Playback control:** the flowline animation can be played or stopped.
- **Decimation control:** it is possible to subsample the displayed flowlines, by updating the available field in the *Inspector* tab. The result is effective after a visualization system restart (a full stop/start cycle).

Flowline description: The single flowline is stored as a Unity Prefab among the available project assets. A C# script defines the flowline evolution in time, reading the positional data from the binary file corresponding to its ID. The general implementation idea is to consider each flowline as a single GameObject (a flowpoint), moving in time, following an established path. The objective is obtaining a controlled flow speed, but at the same time allowing an adaptive behaviour of the stream density depending on the hardware capabilities. Each target position is selected by iterating the coordinates tuple within the *FixedUpdate* method. Special attention has been placed to the flowline timing: the simulation interval between target positions is a directly proportional to the time difference between given measurements. At the same time the fluid motion is achieved using the *MoveTowards* function (always pointing at the latest updated target position) inside the frame-rate-dependant *Update* call. When a known coordinate is reached, the color of the object is updated according to the current flow velocity, following a red-blue colormap.

The visualization is cyclic, flowlines restarts when all reach the end of their motion.

Flowpoint definition: The Flowpoint is the GameObject asset that is instantiated and moved by each flowline script. Our choice for the flowpoint has been a particle generator, customized with the following parameters:

- The generating element is a point-like sphere, with a sufficiently high emission rate (100 particles per second) to obtain a continous flow, in the best hardware conditions.
- The particle starting velocity is zero, this fixes it in the same spot in which it was generated.
- The particle lifetime is a design parameter and defines the length of the displayed flowline. For this project, it has been set to 1.
- The particle size gradually decreases to zero after its generation.

Each particle is rendered as a 2D image (billboarding) to lower its computational requirements.

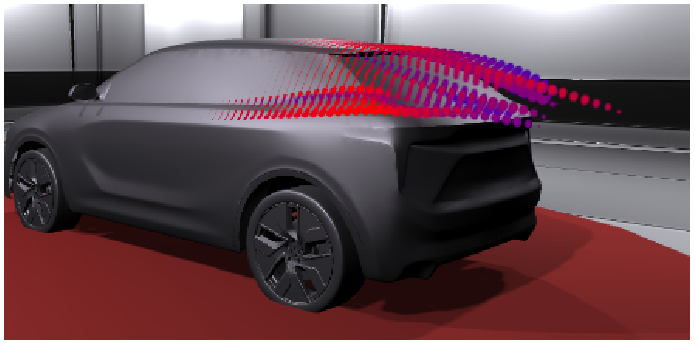


Fig. 4: Flowlines example

IV. Pressure Points: Processing and Visualization

Data refactoring: The data concerning the placement, orientation and values of the pressure points are stored in the *out_coord_norm_pressure.bin* file and its structure is presented here below:

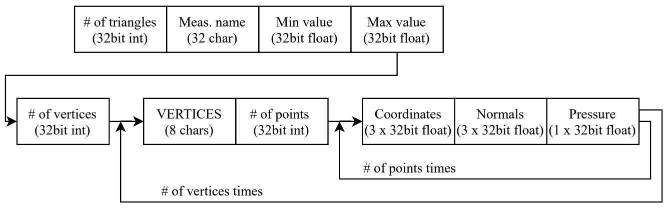


Fig. 5: Mesh binary file structure

To achieve this result we used a Python script to manipulate the data. In it we took advantage of a library called *plyfile* that made the extraction of all the needed components much simpler. As follows, it is presented the process that led to our final results:

- **Model approximation**: since the complete PLY model is made by over 8M vertex and so by 8M pressure points we decided to made an approximated model in order to achieve a more consistent, yet less accurate, representation of the pressure values on our model. Firstly, we loaded the full model on *Meshlab* and by performing a *Simplification: Clustering decimation* (with *world unit* and the *perc on* parameters to 0.05 and 1.015 respectively) we ended up with a model made of almost 19K vertex. Even if the approximation seems drastic (we have only the 0.2% of the points compared with the full model) it has to be considered that the pressure values of groups of points in specific areas of the vehicle vary from the fourth decimal to the six decimal. Since the variation is so little, the approximation let us achieve very good results and also a pleasant presentation of the final model on *Unity*. Lastly, we rotated the model by 90 degrees clockwise and than exported the new PLY object.

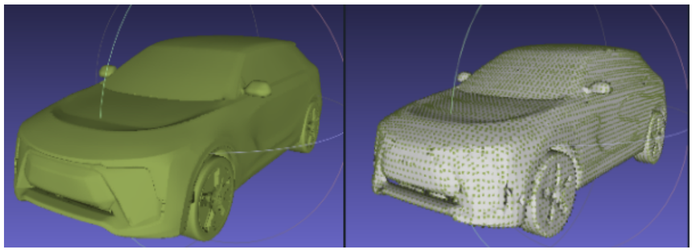


Fig. 6

(a) Left: complete model with 8M vertices

(b) Right: reduced model with 19K vertices

- **Extraction of the points**: After this preliminary phase we created a Python script in which we imported both the full detailed and the reduced models. Since the reduced model has less vertices and consequently different faces, we had to find a way to make a correspondence between the models in order to have a good approximation. In order to do so we created a function that takes in input a point (that it's made by the x, y, z coordinates) of the reduced model and the matrix of points of the complete model. The objective is to find the closest point, in terms of squared distance, belonging to the full model, starting from the reduced one. The results of this approximation are really good since the points differ at a very low decimal. Also the script is performing well considering that it takes more or less 50 minutes to an hour to compute $19K * 8M$ operations. Found this subset of points of our complete model we saved all the coordinates, normals and pressure values in a bin file named *out_coord_norm_pressure.bin*.
- **Unity visualization**: In order to load and work on the data acquired in *Unity* we created a C# script. Firstly, we chose a metric to represent the pressure points. Initially we opted for a static sphere GameObject without shadows and with a white sprite as material to keep everything as simple as possible in order to achieve a decent amount of FPS in game mode. But, even though the GameObject was simplified to its core, rendering almost 19K 3D objects resulted in 30 fps max using a *Nvidia GTX 1660 Super* as graphic card. Since our aim is to have good performance on decent hardware and an overall smooth experience we tried a second approach. This time we moved from a GameObject to a static particle system. This latter gives us the possibility of controlling the behaviour of every single particle both the position, the color and the speed. To control the particles behaviour we developed these functions:
  - **Start**: It's the function in which we load the binary file and create the positions, normals and normalized pressures lists. Also we initialize the particle system using the *InitializedIfNeeded* function.
  - **Update**: Control the user interaction while in game mode. If we press the *p* key we can show/hide the particle system related the pressure points.
  - **LateUpdate**: It's the core method. Here we loop through the particles array and set the position, speed and color of each particle. The position is made by the coordinates (x, y, z) found in the *Start* function, the speed is set to zero for all the particles because we want the system to be static. The color of a particle is controlled by the *Lerp* function that, based on the value of the normalized pressure, linearly interpolates between the two color chosen: light blue and red to represent the less and most extreme pressure values. After the loop ends we set the particle on the particle system.
  - **InitializedIfNeeded**: Function that receives the length of parameter array and initialize both the particle system, the particle system renderer and the array of particles. Also, it includes some useful predefined settings such as the duration, the lifetime of the particle system.
  - **NormalizePressureValues**: auxiliary function used once to find the minimum and maximum values of the pressure array, since running it at every iteration of the code would have been computational demanding, we opted to hardcode those values.

With this approach the animation in game mode is consistently smoother and the achieved fps on the same machine moved from 30 to 100 average fps.

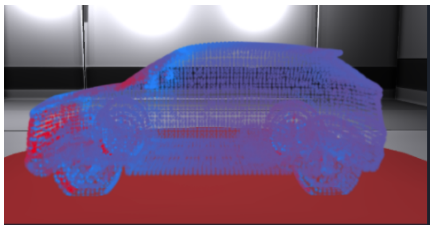


Fig. 8: Pressure points particle system

V. Scene composition

The final scene is a virtualization of a showroom and it is made of the following components:

- **Vehicle**: as we said in section II it's a model simplified without losing the exterior details. It's made of 396K vertices and 688K faces. The original model had 8M vertices and 13M faces.
- **Flowlines**: In section III we explained how we modeled the 603 flowlines as a dynamic particle system using a circular sprite as visual element.
- **Pressure points**: In section IV we modeled the 19K pressure points as a static particle system. The colors are an index of the pressure value of that point.
- **Platform**: it's a simple cylinder GameObject that act as a decorative platform to emulate a car showroom.
- **Futuristic structure**: we built a futuristic showroom like structure using the *3D Free Modular Kit* found on the Unity Assets Store [1]. We didn't use the whole package, so we kept only the visible components.
- **Camera System**: To achieve a showroom appearance we worked on emulating the camera movement that we all know. The user can choose to move the camera around the car in a circular path. Alternatively, the user can choose to press some keys to move the camera to a fixed point.
- **GUI**: A simple, futuristic looking UI was added to simplify the interaction of the user with the simulation.
  - *Camera position*: The user can press the keys from 1 to 5 to switch to a different static camera position
  - *Camera rotation*: By pressing the space bar key the user can toggle on/off the rotation of the camera around the model
  - *Visualizations*: By pressing the P or the F keys the user can enable or disable the pressure points and the flowlines respectively

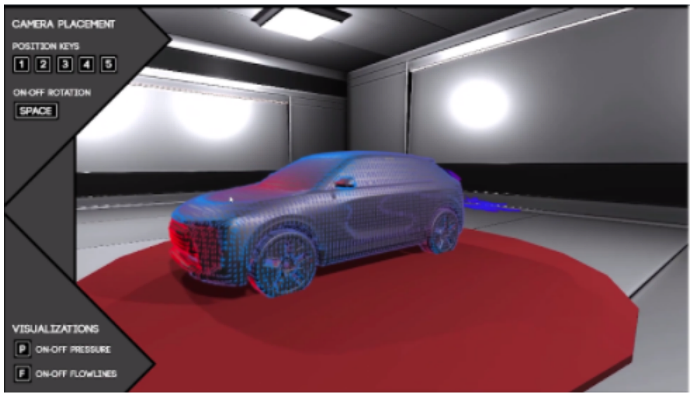


Fig. 9: GUI

[1]https://assetstore.unity.com/packages/3d/environments/3d-free-modular-kit-85732

VI. Concluding Remarks

In this paper we described the main phases of our work. In section II we extracted all the meaningful data from the original binary file provided for this project. Then, in section III we explained how we modeled the flowlines by using a particle system and a GameObject. In section IV we went through the process on how we thought to represent the pressure point with a glance on the overall performance. Finally, we showed how we composed the scene by describing each element that we've inserted to have a pleasing presentation of our solution. We think that our project can be a good starting point for future developments. Some possible further implementation can be: the addition of an animation that can alter the pressure points when the flowlines are above them or adding physics to the model to simulate pressure and see how much we move away from the physical data. We think that this project was exciting because let us work with different languages to accomplish the best results possible for each of our tasks. For data extraction, Python comes really handy because has plenty of libraries to work with raw data. For the 3D modeling we learned to interact with multiple software to have a better comprehension of the model. We have also encountered some difficulties, starting with the data extraction, it was challenging to understand how the data were stored. When we switched to *Unity*. we had to deal with all the console errors that sometimes were unclear and we had to debug manually by rechecking the code line by line. The amount of manipulable parameters and the different variations added to the overall coding difficulties. Thanks to this project, we developed wide scope knowledge about different 3D manipulation software, handling raw datasets and obtaining a pleasant final composition.

A video demo of the project can be found at the link: https://youtu.be/PIZQM981aGA.