MSc thesis in Geomatics

Visualizing massive computation fluid dynamic outputs in game engines using voxelization and dynamic loading

Constantijn Dinklo

April 2024

A thesis submitted to the Delft University of Technology in partial fulfillment of the requirements for the degree of Master of Science in Geomatics

Constantijn Dinklo: *Visualizing massive computation fluid dynamic outputs in game engines using voxelization and dynamic loading* (2024)
This work is licensed under a Creative Commons Attribution 4.0 International License.
To view a copy of this license, visit http://creativecommons.org/licenses/by/4.0/.

The work in this thesis was carried out in the:

3D geoinformation group
Delft University of Technology

Supervisors: Dr. C. García-Sánchez
Dr. H. Ledoux
Co-reader: I. Pađen

Abstract

Computation Fluid Dynamics (CFD) simulations are used in a diverse set of fields such as aerodynamics, automotive, biomedical engineering and wind impact. With the advancements in technology the scale of these simulations has increased significantly. This has resulted in *massive* CFD simulations — simulations that do not fully fit within RAM. Due to the limitations in RAM, visualizing the results of *massive* CFD simulations has become an issue, mainly on personal computers. Existing solutions necessitate either substantial external servers equipped with sufficient RAM, a costly and inefficient approach that struggles to accommodate increasingly larger CFD simulations, or reliance on manual intervention for data loading and unloading. However, manual intervention introduces potential for error.

This thesis proposes a solution to automatically load/unload data from memory based on real-time demand. To achieve the proposed solution the methodology is split into two main stages: **(a)** pre-processing and **(b)** visualization.

The pre-processing is required to efficiently manage the *massive* amount of data during the visualization step. It involves segmenting the study area into smaller, more manageable regions. Furthermore, it generates multiple level of details for each region such that the desired level of detail can be used as required.

The visualization is performed within game engines — Unity for this thesis. Game engines provide a solid starting platform as they include aspects such as read/write operations, rendering capabilities, and flexible code execution. During visualization, the relevant regions — regions that are within the user's point of view — are automatically loaded into memory. subsequently, regions that leave the user's point of view are automatically loaded out of memory. Utilizing iso-surfaces, volume rendering, and barbs, the loaded data is then presented to the user.

The results showcase that visualizing *massive* CFD results within game engines is possible in real-time. However, through the data transformation performed in the pre-processing step the data has lost some accuracy. Thankfully, an downward trend can be seen in the loss of accuracy as the level of detail increases. The result show that for data used in the thesis the pre-processing takes between 390 seconds (6 minutes and 30 seconds) and 972 seconds (16 minutes and 12 seconds). Furthermore, a data size reduction of up to 84% can be seen after the pre-processing has finished.

Acknowledgements

I would like to express my gratitude to my first supervisor, Dr. Clara García-Sánchez, for her continuous support throughout this graduation project. Even in times which seemed dire she kept providing me with valuable feedback and suggestions to make sure this final product was made a possibility. Special thanks also go to Dr. Hugo Ledoux for his insights into possible solutions for the problems tackled during this thesis. Finally, many thanks to Ivan Pađen for his comments.

My family played a vital roll in allowing me to finish my graduation project. They provided me with the support and encourage me to keep going, no matter how though the situation got. As for my friends, the amount of enthusiasm and joy they provided me cannot be matched by anything. The length of this thesis could not explain how important my family and friends were during this entire project.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

1.1 Motivation

Computational Fluid Dynamics (CFD) is a branch of fluid mechanics that deals with the numerical analysis and simulation of fluid flows Anderson and Wendt [1995]. It uses computational methods, algorithms, and numerical techniques to solve the governing equations of fluid motion, which are typically partial differential equations that describe the behavior of fluids Anderson and Wendt [1995].

CFD finds application across diverse industries, including aerodynamics, automotive, Heating, Ventilation, and Air Conditioning (HVAC), and biomedical engineering. Specifically, CFD is increasingly used to study the impact wind has within urban areas for addressing aspects such as air pollution Canepa [2004] Chu et al. [2005] Hanna et al. [2006] Gromke et al. [2008], natural ventilation of buildings Jiang and Chen [2002] Evola and Popov [2006] Chen [2009], convective heat transfer Defraeye and Carmeliet [2010] Defraeye et al. [2010], wind erosion and wind energy Mochida et al. [2007] Milashuk and Crane [2011]. To study the impact the wind has on urban areas often requires simulating large areas Blocken et al. [2012]. However, it is still desirable to know how the wind behaves within small regions, such as between buildings of the study area. In order to incorporate this into the simulation we often need to discretize the domain area into small regions, so-called cells in the CFD field. The combination of a large study area and small cells results in an overall large number of cells for the simulation, leading to results with very large data size (from GB to TB).

The scale of a CFD simulation is determined primarily by the number of cells used in the simulation. The number of cells is in turn directly correlated to the ratio of the size of the study area to the size of each cell. Notably, a small study area can still qualify as large-scale if the cell sizes are small, and conversely, a large study area with considerably larger cell sizes might be considered as a small scale CFD simulation (see Figure 1.1).

The advancement in hardware and computational capabilities has facilitated CFD analyses at larger scales Muñoz-Esparza et al. [2021]. Despite the advancements in processing larger CFD simulations, visualizing large CFD simulations results still poses a challenge. While it is acceptable for simulations to run for an extended duration, the visualization process demands real-time interaction, enabling users to navigate freely throughout the environment and view results from diverse perspectives.

Several software options specialized in CFD visualization, such as Autodesk CFD Verma and Samar [2018], Tecplot 360 Tecplot [2023], ANSYS ANSYS [2023], and Paraview Ahrens et al. [2005], are available. However, the majority of these solutions necessitate payment plans Verma and Samar [2018] Tecplot [2023] ANSYS [2023], which might be feasible for commercial entities but can be restrictive for start-ups and researchers with budgetary constraints. Free alternatives, like Paraview, are widely used due to their accessibility. While these free tools effectively visualize standard-sized CFD simulations, the visualization of *massive* CFD

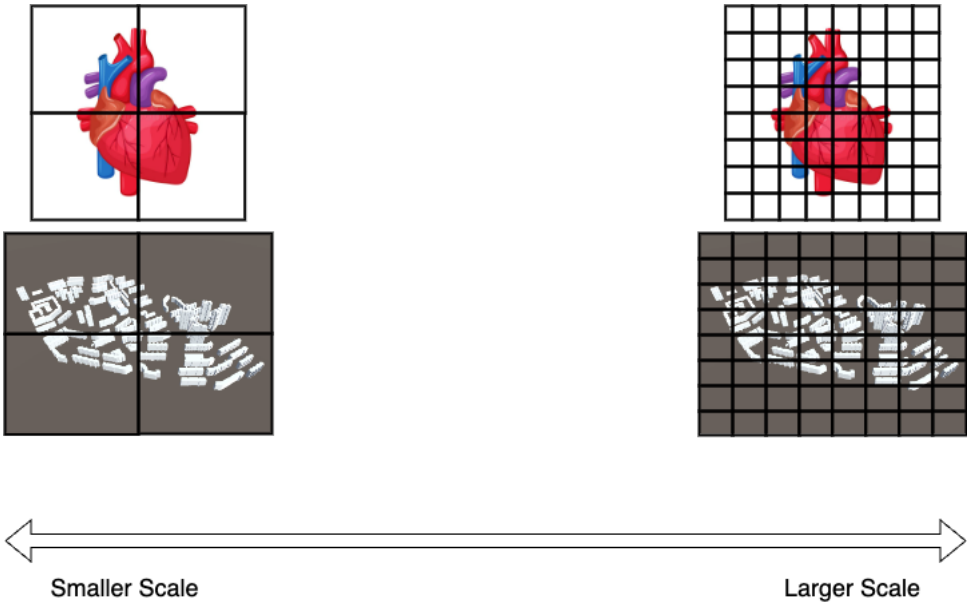


Figure 1.1: Sketch to explain the size of CFD simulation results: the actual area does not determine the scale of the CFD simulation. On the left side both areas are split into 4 cells which is a smaller scale than on the right in which both areas have been split into 16 cells.

simulations presents a distinct challenge. The concept of a "massive" CFD, in the context of this thesis, refers to any CFD dataset that cannot be accommodated entirely within a system's Random Access Memory (RAM). Since there is no access to payed CFD software it is not clear whether these are capable of visualizing results of *massive* CFD simulations.

Due to limitations in RAM, the standalone version of Paraview encounters difficulties visualizing entire massive CFD results. While Paraview does offer options to handle massive CFD simulations, these methods rely on external servers and parallelization Ahrens et al. [2005]. However, setting up and maintaining such servers can be costly. Furthermore, the external servers must still possess RAM capacities larger than the storage size of the CFD results Ahrens et al. [2005], presenting inefficiencies when scaling to even larger CFD simulations. Hence, it's crucial that any proposed solution in this thesis functions efficiently on personal computers without requiring external assistance.

At a fundamental level, this thesis proposes a method to visualize massive CFD simulations by dividing the entire dataset into smaller chunks. These chunks can then be loaded in and out of memory as needed. This approach aligns with one of the potential methods for visualizing massive CFD simulations in the standalone version of Paraview. However, in Paraview this method requires manual loading of each chunk into memory when visualizing. Subsequently if there is not enough RAM available, the manual unloading of chunks is required. This process can be a cumbersome and error-prone. If not managed meticulously, this manual intervention may lead to memory overload and system crashes. A more effective solution would automate the process of loading and unloading chunks in and out of memory when required.

In the context of visualizing *massive* CFD simulations, only the data within the user's Point of View (PoV) needs to be loaded into memory (Figure 1.2). To determine which chunks need

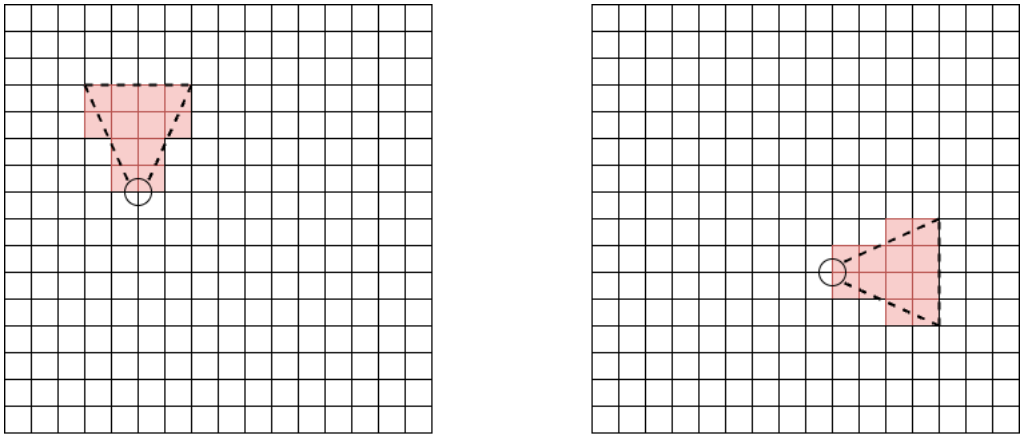


Figure 1.2: For different points of view, different chunks need to be loaded into memory. Areas shown in red should be loaded into memory.

to be loaded in, every chunk needs to check whether it is with in the user's PoV. However, as mentioned earlier, the user should be able to freely move around the environment. This can alter which chunks fall within the user's PoV.

Unfortunately, this method of moving around the environment and automatically loading/unloading data is not well supported by current CFD visualization tools. However, Game Engines do provide an environment which is built to support movement and automation of loading/unloading data. Game Engines are specialized pieces of software designed to develop games within them. They provide the building blocks from which almost any game can be created. Furthermore, games are diverse, which means that Game Engines are capable of providing buildings blocks for a wide variety of applications, not necessitating a strict restriction to developing games.

Game Engines provide a good starting platform from which to build the required application. They provide an environment in which data can be placed, read/write operations to perform the necessary loading/unloading of data, provide the ability to render data in user defined manner (through shaders), and provide flexible code execution through user defined scripts. For this thesis, the Game Engine of choice is Unity. It is used extensively within the game development world and is well documented.

1.2 Research Question

Based on the motivation behind this project and the capabilities of game engines outlined within the motivation, the main research question that this thesis will look to answer is as follows:

Can massive Computational Fluid Dynamic results be visualized using Game Engines in real-time while presenting valuable information?

To facilitate the integration of CFD data into Unity, it must be pre-processed into a format which Unity can handle. Given the substantial scale of the data, the time demand becomes a crucial consideration. Consequently, a key inquiry addressed in this thesis is:

What is the time requirement for pre-processing the CFD data?

Furthermore, this modification of the data will result in both a reduction in the overall data size and the accuracy of the data. This is a trade-off designed to allow for the visualization of such *massive* CFDs. Therefore some further questions will be answered, including:

How much reduction in memory does the modifications of the data provide?

What is the error introduced into the data by the modifications?

1.3 Research Scope

This thesis deals strictly with the visualization of a CFD result. It does not examine how CFD simulations are setup or calculated. It is assumed that the CFD result is complete and error free. This thesis encompasses the entire process from CFD result data to data reduction, data processing, and ultimately, visualization.

CFD outputs exhibit diverse data structures based on the software used for their generation and the applied calculation parameters. While the methodology described in this thesis is generic, the prototype developed specifically targets CFD results generated through OpenFOAM. It is anticipated that other CFD outputs may require adjustments in their data structure to align with the proposed solution. The requirement that this thesis imposes on the CFD data is that every data value can be assigned to some point within the study area.

Furthermore, all results and discussion presented within this thesis are framed within the context of Unity. Alternative Game Engines and other software capable of visualizing CFD results are not discussed within this thesis, with the exception of a brief overview of Unreal Engine. Any comparisons drawn to other CFD output visualizations are made with reference to Paraview. Since this thesis does not delve into all possible tools available it makes no claim as to whether these tools are the best for the solution.

The focus of this thesis is to assert the feasibility of visualizing *massive* CFD results within Unity using dynamic data loading techniques. There are countless visualization techniques to visualize data, within this thesis only iso-surfaces, volume rendering, and barbs will be discussed.

1.4 Thesis Overview

The contents of this thesis is described in four main chapters. Chapter 2 discusses visualization and data managment techniques related to this thesis. It discusses how CFD results are currently visualized. This includes a short insight into how current *massive* CFD results are handled. Furthermore, it discusses how level of detail and dynamic loading are currently used to handle large amounts of data.

Chapter 3 outlines the methodology this thesis proposes to visualize *massive* CFD results. Both the pre-processing steps and the actual visualization process are discussed. Furthermore, the implementation of the visualization is presented as it is strongly linked to the methodology.

Chapter 4 then discusses the results of both the pre-processing and visualization methods propossed in Chapter 3.

Chapter 5 starts off by answering the proposed research questions. This is followed with a discussion of these results. Afterwards, the main contributions made by this thesis are presented. Finally, the limitations and the recommended future work is outlined.

2 Related Work

This chapter discusses related work on which this thesis is build. It specifically explores three primary subjects. Firstly, it examines the existing methodologies for visualizing CFD simulations, including massive simulations. Secondly, the concept of level of detail is discussed. Finally, it explores dynamic loading in the context of this thesis.

2.1 CFD visualizations

CFD visualization, also known as post-processing operations, refers to the process of presenting the outcomes derived from CFD computations in a manner comprehensible to human perception. The output of a CFD simulation comprises a sequence of numerical values, elaborated upon in Chapter 3.1. While each individual numerical datum within the data can be understood, comprehending their collective implication poses a challenge. Consequently, to gain practical understanding of the results, these numerical outputs are transformed into visual representations in either 2D or 3D formats.

In 2D representation, visual outputs can be bar or line charts, typically depicting numerical attributes inherent in the data, such as the distribution of density values across the data through the use of a histogram. On the other hand, 3D visualizations are employed to project geometries and volumes into a three-dimensional spatial context. An example showcasing a 3D visualization within Paraview is illustrated in Figure 2.1 Ahrens et al. [2005]. This method serves to enhance the understanding of complex numerical data by translating it into visual renderings that are easier to analyze.

In their paper Blocken et al. [2012], the authors employ visual CFD post-processing techniques to assess pedestrian discomfort caused by wind conditions. Initially, they conduct a wind simulation covering the TU Eindhoven campus. Subsequently, they generate a visual representations of the velocity distribution over the campus in Figure 2.2, derived from the outcomes of the CFD simulation. These results enable the classification of data into distinct quality classes, which, in turn, facilitate the characterization of the discomfort experienced by pedestrians due to prevailing wind conditions.

As mentioned in chapter 1, there are many tools available for CFD visualization. Each software package possesses capabilities and constraints which differ from software to software. However, due to limitations in scope, this thesis will focus on Paraview as a foundational benchmark.

Paraview serves as an open-source program, allowing unrestricted utilization without the necessity of licenses, unlike most of the other software packages. Notably, it is compatible with major operating systems, ensuring compatibility across computing platforms Ahrens et al. [2005]. Additionally, Paraview stands as a widely adopted software tool for visualizing CFD outputs, owing to its comprehensive documentation and extensive usage within the

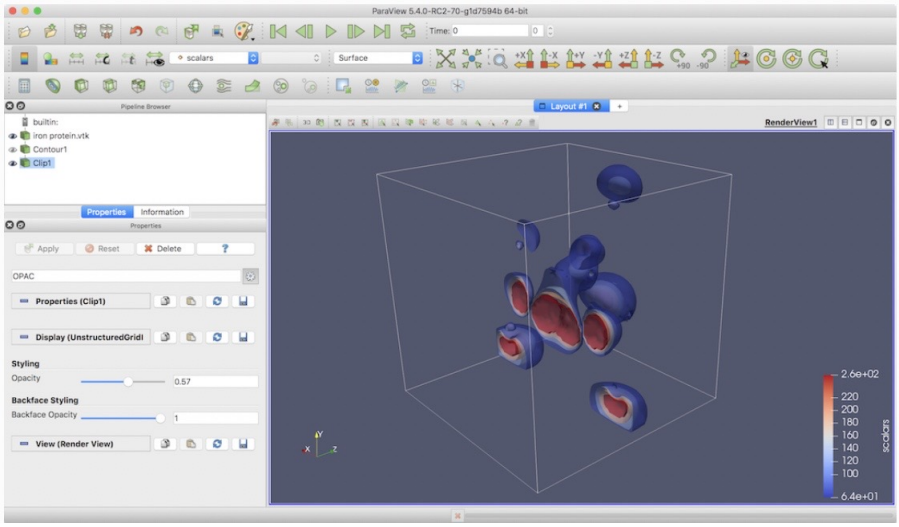


Figure 2.1: Example of 3D visualization. Ahrens et al. [2005]

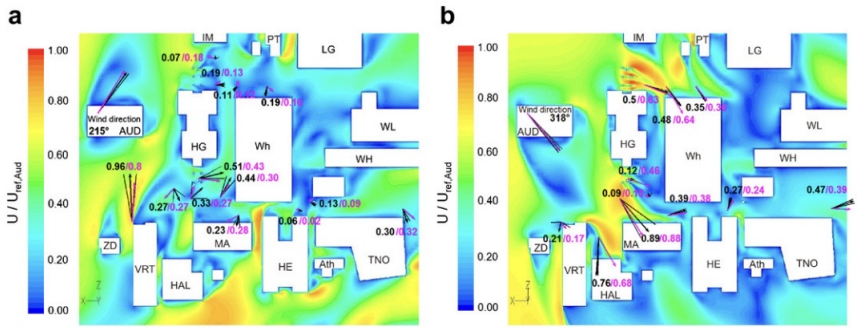


Figure 2.2: Eindhoven Campus CFD visualization Blocken et al. [2012]

field. These attributes make it an ideal baseline for exploring CFD visualization methodologies within the context of this thesis.

2.1.1 Massive CFD visualization

Papers such as Muñoz-Esparza et al. [2021] have produced very large CFD output datasets. Visualizing these results is complex. In order to visualize some of the results in the paper, they take a slice of the data at a specific height and visualize the data as a 2D overlay. This 2D visualization can be seen in Figure 2.3 which show the results of the simulation performed over Dallas. In the same paper there is a single instance of visualizing the results in 3D (Figure 2.4). Besides mentioning that is was developed using VAPOR Clyne et al. [2007] there is no further indication of memory requirement or process required to generate the presented visualization.

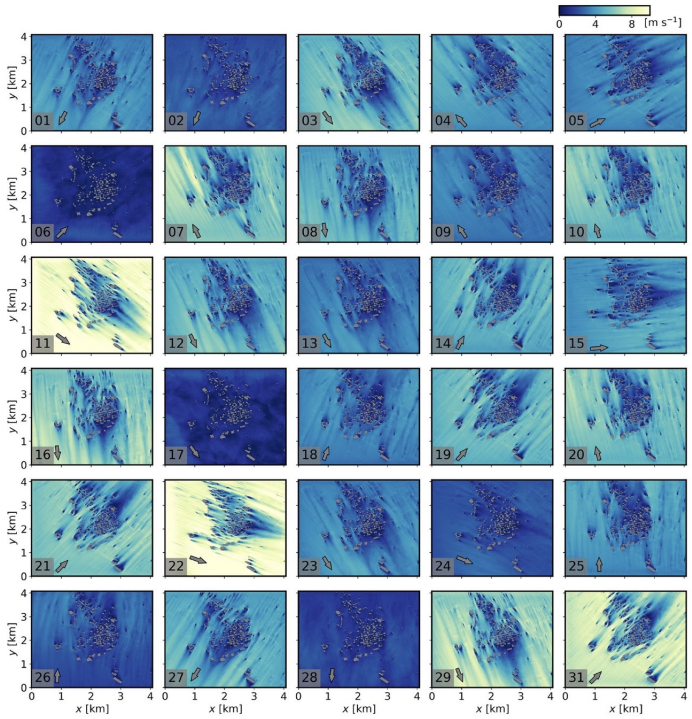


Figure 2.3: Horizontal cross sections of 30 min averaged wind speed, U, at a height of 26 m above ground level for 30 days in January 2018 at 12:00 local time. The day of the month is indicated within the lower right gray box. Gray arrows indicate the large-scale wind direction from the lateral boundary conditions provided by Weather Research and Forecasting 3 km mesoscale simulations. Muñoz-Esparza et al. [2021]

Paraview also mentions the visualization of large scale CFD results. Figure 2.5 depicts a visualization of a CFD simulation performed in PHASTA with 1.3 billion elements. To visualize this result a total of 256 thousand Message Passing Interface (MPI) processors are used. Unfortunately they make no mention of how much time is required to generate this visualization. Neither do they make any mention on how much memory the data requires.

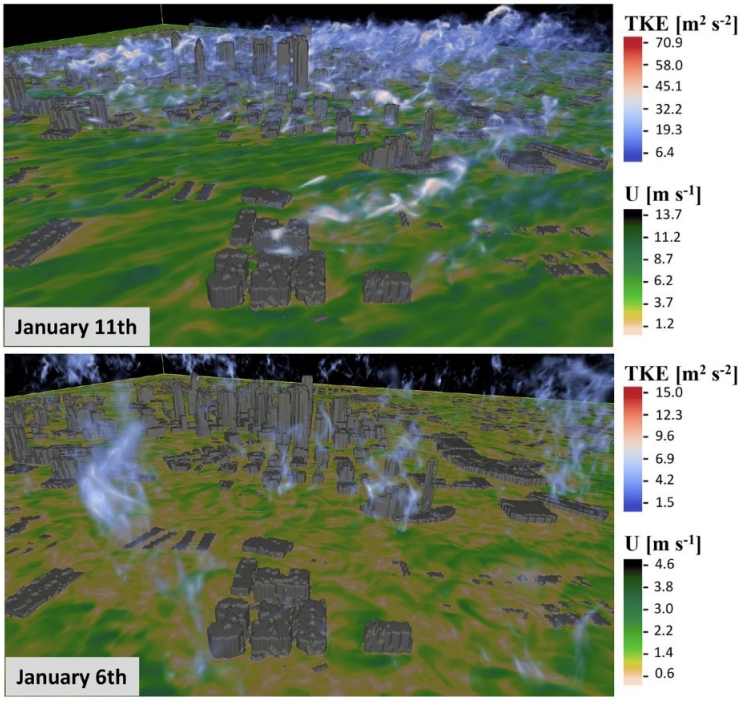


Figure 2.4: Volume rendering visualization of Dallas. Muñoz-Esparza et al. [2021]

Sparse mentions of other large scale CFD visualization are present in other papers. However, they too fail to accurately mention how these are created. Furthermore, as an anecdotal example, while watching tutorial videos on how to use VAPOR Clyne et al. [2007], a CFD visualization tool, there is frequent mention of being careful to not visualize large CFD simulations at the highest accuracy for concerns of memory overload and system crashes.

2.1.2 CFD visualization in Unity

Unity Engine is a game engine produced by Unity Technologies. While originally designed for game development is has since expanded to be used in a wide variety of industries including Architecture and Construction, Automotive, Energy, etc.

Unity has been used previously to visualize CFD outputs. In their paper "CFD post-processing in Unity3D" Berger and Cristie [2015], Matthias Berger and Verina Cristie explain how they took a CFD output and visualized the results in Unity3D. Unity3D is the same as Unity, simply specifying that it is being used in 3D as there is also a 2D option. One of their main goals is to provide an interface that is easier for non-expert users to use to visualize a CFD output. They list the following benefits for using Unity to post-processing a CFD:

1. Does not require a license to execute
2. Platform independent
3. Optimized for visual performance

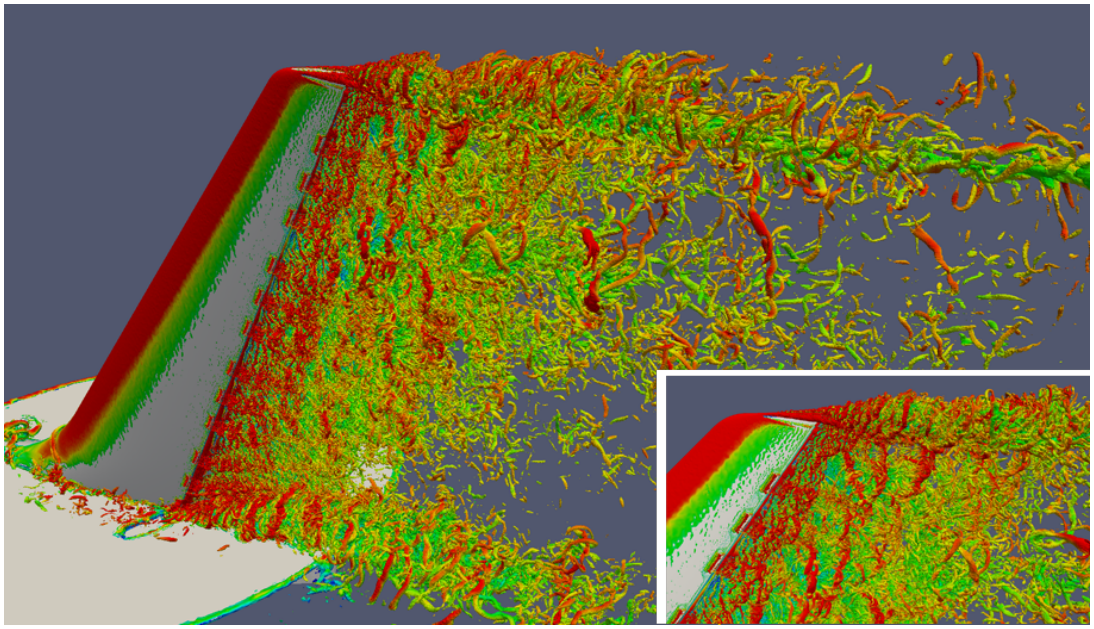


Figure 2.5: Visualization of simulation with 1.3 billion elements. Provided to Paraview by Michel Rasquin. Ahrens et al. [2005]

4. Does not require expert knowledge and skills to run

All of the first three advantages are shared with Paraview. The fourth advantage is of course subjective and should be taken with some wariness.

They also list 2 disadvantages for using Unity to visualize CFD results:

1. The post-processing step has to be implemented from scratch
2. The results in Unity3D are far less accurate than established post-processing software

In short, these disadvantages indicate that Unity is not a dedicated CFD post-processing software. The need to implement the post-processing from the ground up entails developing every facet of the pipeline, from loading to filtering to visualization. This also includes creating data structures which can support the CFD data. While Unity provides some tools to handle the process, such as file reading, these tools are not designed to directly visualize a CFD output. Therefore, this thesis will explain and discuss the steps taken to accomplish this process. The second drawback, pertaining to less accurate visualization, will be discussed in more detail in the results chapter of this thesis. Although Matthias and Verina mention this as a disadvantage the extent of the reduction in accuracy is not explicitly quantified in their paper.

Matthias and Verina's study area encompasses approximately 14 million cells. In their approach to visualize the CFD output, they initially partition the study area into a regular grid (*kxlxm*), such that each cell is defined by dimensions (*space length / k* x *space width / l* x *space height / m*). This segmentation is undertaken with the objective of facilitating efficient access to desired information. They emphasize the need to carefully balance the sizes of the cells. Excessively large cells would compromise visualization accuracy, while overly

small cells could lead to substantial computer memory consumption. Achieving an optimal compromise between these considerations results in an effective visualization.

2.1.3 VTK in Unity

Since Paraview uses the Visualization Toolkit (VTK) to visualize their results it might be logical to also use that library in Unity. However, when adding VTK to Unity through the Unity Asset Store there were compilation errors. Through research during this thesis it became clear that multiple people had similar issues that weren't resolved. Because of this issue, the possibility of using VTK within Unity was dropped. There is an alternative paid version of VTK for Unity, however, since this thesis does not have any budget this option was also discarded. While VTK could be a potential advantage in the visualization process, Unity does already have well optimized built-in functionality to display geometries.

2.2 Level of Detail

Level of Detail (LoD) refers to the amount of detail an entity should be assigned in some state. The definition and application of LoD vary depending on the specific context. In the context of this thesis, LoD is employed to dictate the level of precision required for the CFD visualization. LoD, fundamentally, serves as a strategic compromise between precision and efficiency in the visualization process. It is noteworthy that a higher LoD demands increased storage space.

LoD finds application in a variety of fields, such as Computer Graphics, Geographic Information Systems (GIS), Architecture, 3D printing, Games and Data Visualization. In Computer Graphics a higher LoD might mean that the number of points and faces used to represent an object increase. As can be seen in Figure 2.6 the higher the number of triangles, the more detail is provided in the monkey's face. Of course this comes with the trade off that the higher resolution face requires more data and takes longer to render.

Figure 2.6: Different Level of Details. MystiveDev [2021]

For a relatable everyday example, consider Google Maps. Google Maps partitions the world into discrete tiles Goo [2023]. Based on the zoom level there are a different number of tiles which cover the entire surface, show cased in Figure 2.7 Goo [2012]. At the top zoom level the entire surface of the earth is covered with a singular tile. This tile can only display minimal detail as it needs to encapsulate the entire world. As the LoD increases (zooming in) the number of tiles increases thereby decreasing the area a single tile needs to display. This results in tiles being capable of providing a more precise and detailed view of the area they cover. As can be seen in Figure 2.8, a tile a zoom level 17 has much more detail than a tile at zoom level 7 TomTom [2020].

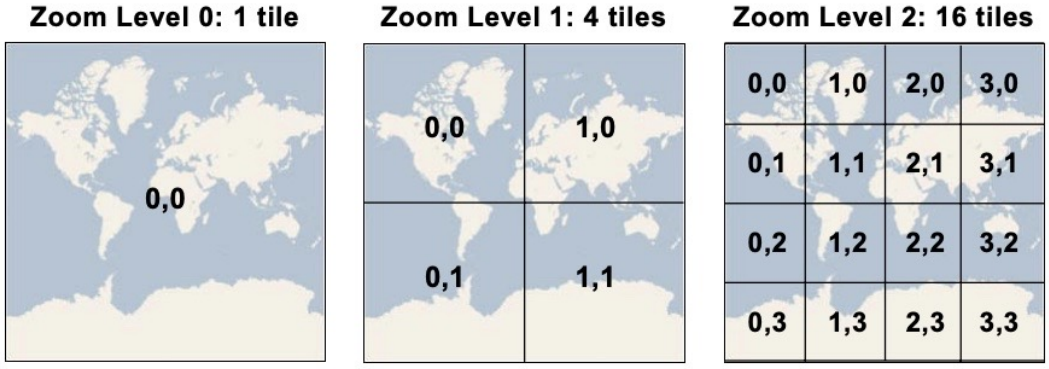


Figure 2.7: Partitioning of tiles for the first 3 Google Map zoom levels Goo [2012]

2.3 Dynamic Loading

At a fundamental level, dynamic loading is a principle that revolves around the dynamic loading of software components at run-time rather than at compile time.

In their paper Friston et al. [2017] the authors dynamically load remote geometries into Unity. The geometries are stored on a remote 3D Repo server. They designed their own 3DRepo4Unity library which is implemented in .NET. This library dynamically loads in the geometries from the 3D Repo. As part of the geometry gets loaded it is displayed in Unity. As seen in Figure 2.9 not the entire geometry needs to be loaded for parts to be shown.

Another example of dynamic loading comes from the paper "Locality Aware Dynamic Load Management for Massively Multiplayer Games" Chen et al. [2005] where parts of the world are only loaded as needed. In Massive Multiple Games the worlds in which players play are very large. This means that not the entire world can be loaded at once. They solve this issue through partitioning the world space into multiple regions (Figure 2.10). Each region can then be loaded for a player when they are in or close to that region. Figure 2.10 shows players in purple with a gray area of interest around them. Only regions and items that (partially) fall within the region of interest are loaded for that player.

To build on the example mentioned previous with Google Maps. Google Maps only needs to load a tile if it satisfies the following conditions; the tile belongs to the current zoom level and the tile is within the area the user is currently looking at Goo [2012]. All other tiles do not currently need to be loaded into memory, saving valuable space. When the user moves

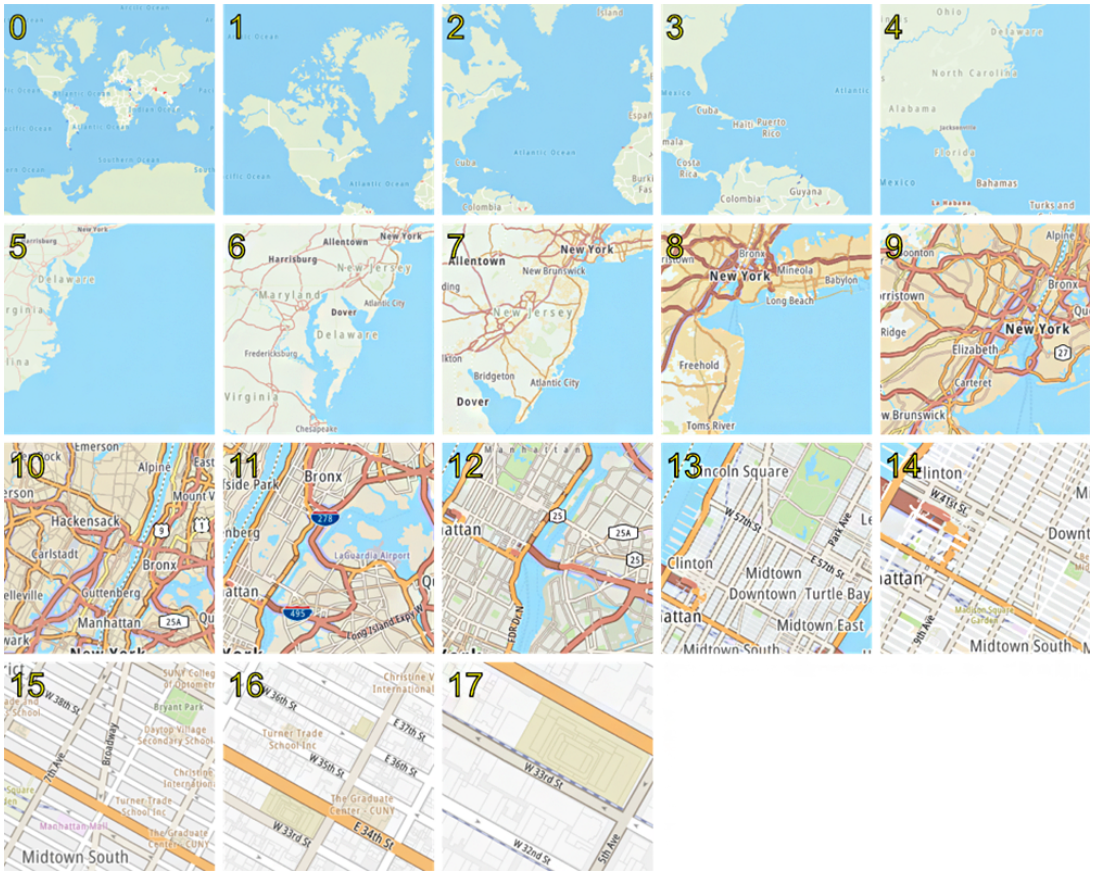


Figure 2.8: The 17 different zoom levels on Google Maps TomTom [2020]

around the map, Google Maps recalculates which tiles are required and loads the desired tiles. While this approach enhances the loading time, it is also necessitated by the fact that (most likely) not all tiles can fit into memory. To keep all tiles within memory would require 55.1 GB of memory Goo [2012].

2.4 Visualization Techniques

This section provides a brief explanation of the visualization techniques used in this thesis. For this thesis, visualization techniques are methods used during visualization of data to gain a better understand of that data.

2.4.1 Iso-Surfaces

Iso-surfaces are surfaces in three-dimensional space that represent a constant value of a particular quantity or property. These surfaces are used extensively in scientific visualization, particularly in fields such as fluid dynamics, medical imaging, and molecular modeling. For

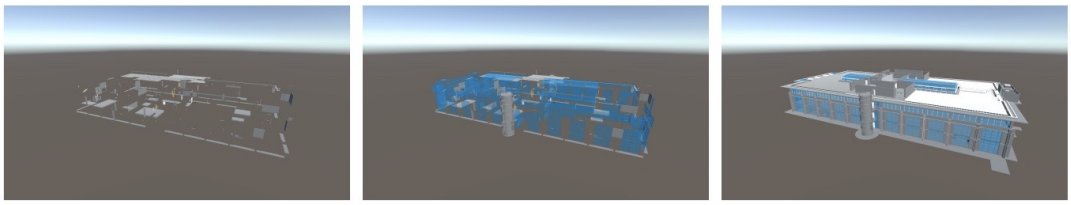


Figure 2.9: Different parts of the geometry are shown when that part is loaded. Friston et al. [2017]

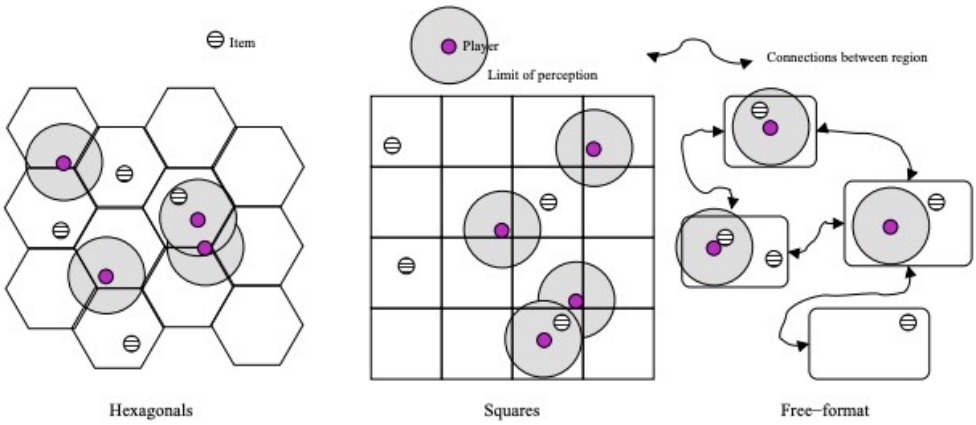


Figure 2.10: Different possible ways to partition the world space into smaller regions. Chen et al. [2005]

example, in fluid dynamics, iso-surfaces could represent boundaries of constant velocity, pressure, or temperature within a fluid flow simulation (see Figure 2.11.

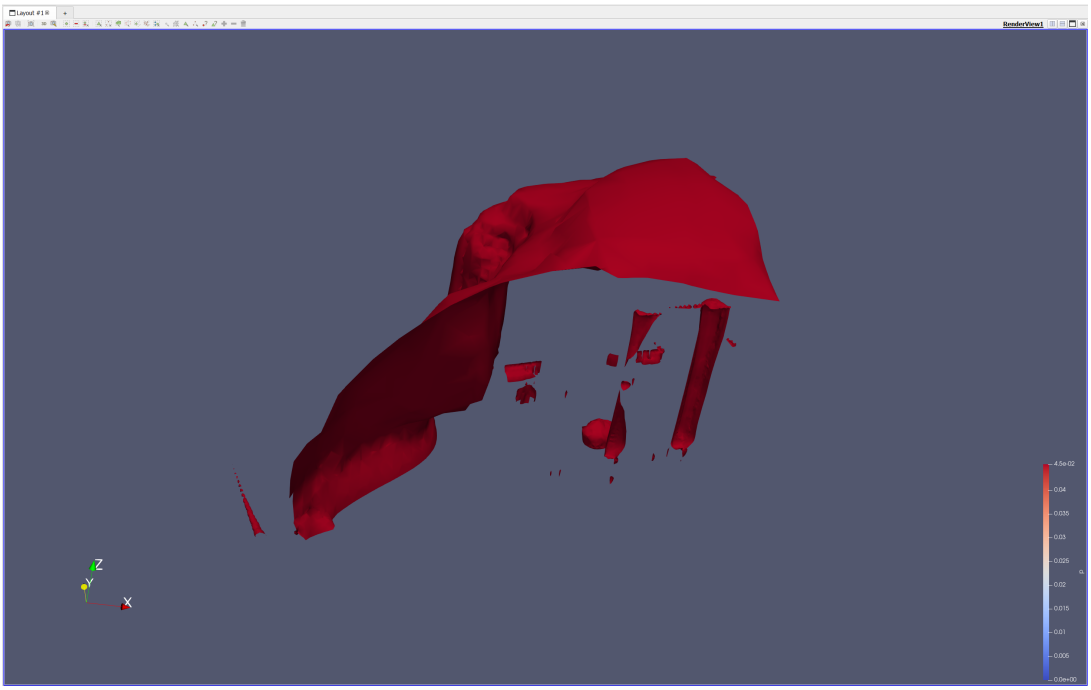


Figure 2.11: Representation of an iso-surface in paraview. This displays the surface on which the pressure value is 0.04

2.4.2 Volume Rendering

Volume rendering is a technique used in computer graphics and imaging to generate a 2D image from a 3D volume dataset Westover [1990]. Volume rendering allows for the visualization of complex internal structures and properties within 3D datasets that may not be readily apparent from traditional 2D projections. As illustrated in Figure 2.12, the employment of volume rendering techniques unveils certain bone structures which would not have been possible through surface rendering alone.

2.4.3 Barbs

In computer graphics, ”barbs” usually refers to markers or indicators placed within the environment to denote direction or magnitude. They are commonly used in visualization to represent vector fields, such as fluid flow, magnetic fields, or wind patterns. Figure 2.13 depicts the use of barbs to indicate wind velocity direction.

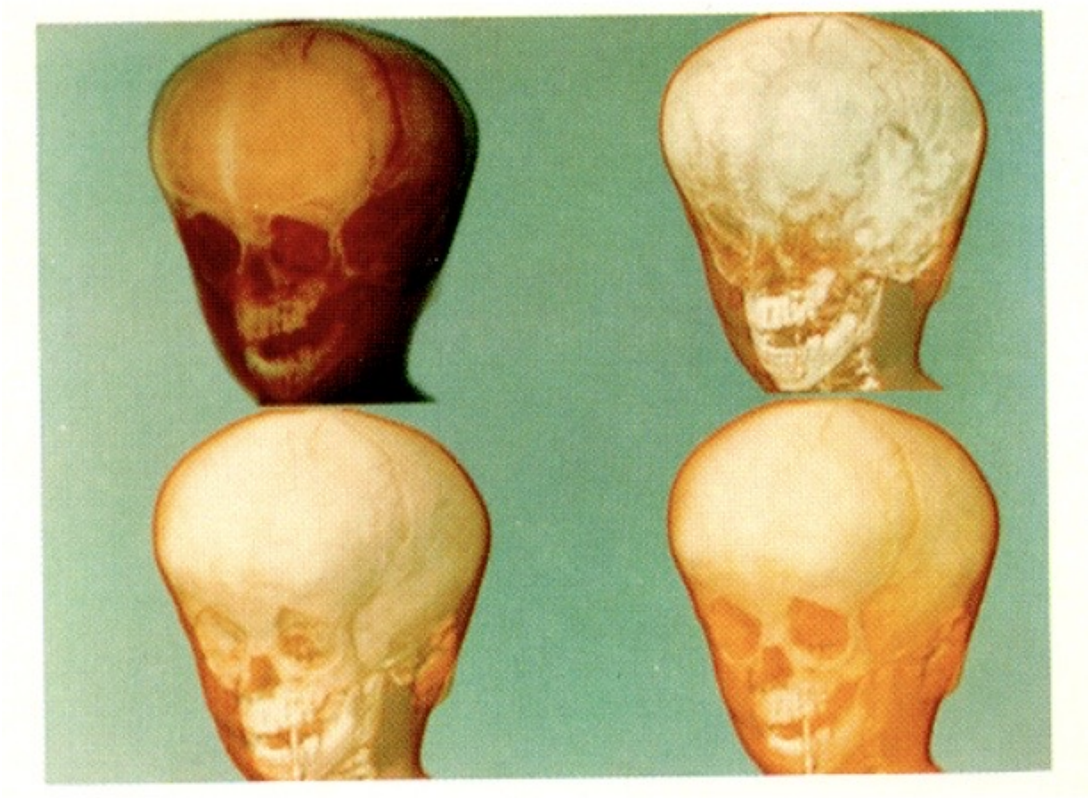


Figure 2.12: Volume Rendering of human head. Drebin et al. [1988]

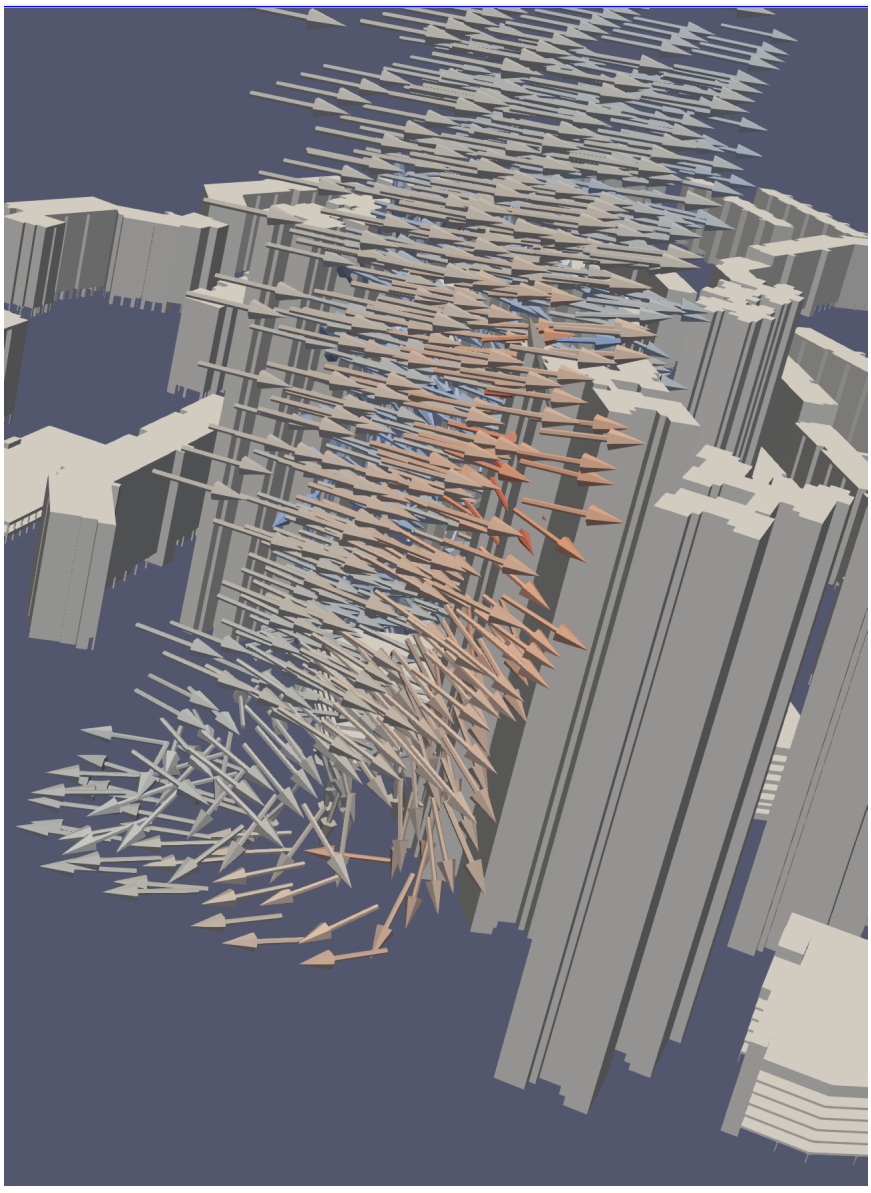


Figure 2.13: Representation of barbs in paraview. The arrows indicate the direction in which the wind is moving.

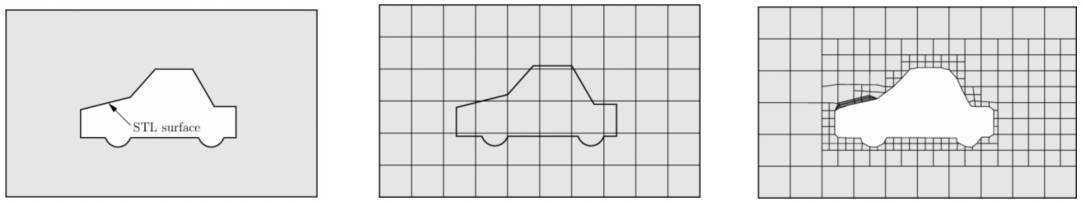
3 Methodology and Implementation

The content of this chapter is the analysis of the methodology developed to address the research questions of this thesis. The methodology is divided into two main steps: (a) Pre-processing the data into manageable regions, (b) Visualizing the regions in Unity. However, first the structure of a CFD output will be discussed to gain an understanding the input data for the pre-processing.

While all the steps are performed on 3D data sets, many of the steps have exemplary images depicted in 2D for simplicity.

3.1 Structure of OpenFOAM CFD outputs

A CFD dataset is meant to capture an environment in which the simulation needs to occur. This includes both the area in which the fluid needs to flow and the Computer Aided Design (CAD) geometries containing or affecting the fluid flow (i.e. engine walls or buildings). The CAD geometries (Figure 3.1a) within the simulation are captured through the use of meshes and generally come in the form of STL or OBJ files Sousa et al. [2018]. The rest of the space, the space in which the fluid is located, is discretized using cells (Figure 3.1b and 3.1c) Sousa et al. [2018].



(a) (b) (c)

Figure 3.1: This shows the process of generating a CFD input with snappyHexMesh method. (a) Initial area with CAD geometry. (b) Area divided into cells. (c) Final cell structure which fits around the CAD geometry. Ahrens et al. [2005]

Figure 3.1 shows the basic steps to create a CFD mesh using the snappyHexMesh method. First the CAD geometries (note that it can be more then 1) are placed within the area. Secondly, the area is split into cells. Finally, the cells are modified to fit around the CAD geometries. Since this thesis does not look at generating CFD models or performing the calculation, this thesis will not go into detail about the multitude of methods which can be used to generate these cell structures. The main take away from this is how cells capture the environment around the CAD geometries, not how the cells are generated. Furthermore, this thesis looks specifically at the CFD structure created by OpenFOAM.

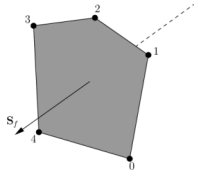


Figure 3.2: A single face in OpenFOAM.

All cells in the CFD are made up of a set of faces Ope [2004]. Faces are made up of an ordered list of points, such that neighbouring points are connected through an edge (see Figure 3.2). The direction of the normal of the face is defined by the right hand rule i.e. looking towards a face, if the numbering of the points follows an anti-clockwise path, the normal vector points towards you Ope [2004].

The list of faces which make up a cell can be in arbitrary order. As stated in the OpenFOAM documentation, each cell needs to have the following properties:

1. **Convex:** Every cell must be convex and its cell centre inside the cell
2. **Closed:** Every cell must be closed, both geometrically and topologically where:
    - geometrical closedness requires that when all face normals are oriented to point outwards of the cell, their sum should equal the zero vector to machine accuracy.
    - topological closedness requires that all the edges in a cell are used by exactly two faces of the cell in question

Furthermore, the set of cells within the data need to be **contiguous**, that is they must completely cover the entire domain and must not overlap each other Ope [2004]. This constitutes the CFD Mesh, the resulting folder structure looks as follows:

```
PolyMesh
├── Points
├── Faces
└── Cells
```

A simulation is performed on a set of quantities of interest (velocity, density, etc). These determine which values the simulation will predict. Each cell in the CFD contains a value for each quantity of interest (figure 3.3).

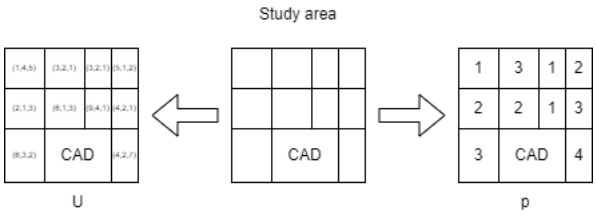


Figure 3.3: Each cell has a value depending on the quantity of interest being looked at.

The value for each quantity of interest is stored in its own file. The number of values within this file is equal to the number of cells in the simulation. Each value in a file is linked to the

corresponding cell in the *cells* file. (e.g. the 12th value in the *k* file is assigned to the 12th cell).

```
├── k
├── U
└── ...
```

The combination of the CFD mesh and the quantities of interest values combine to create the CFD output structure. The corresponding final file structure looks as follows:

```
CFDOutput
├── PolyMesh
│   ├── Points
│   ├── Faces
│   └── Cells
└── Values
    ├── k
    ├── U
    ├── ...
    └── ...
```

The complete CFD output contains more files, however, they are of no interest for this thesis.

3.2 Constructing Multiple LoDs from a CFD output

Given the limitation of loading the complete CFD result into memory due to its size, this thesis proposes a methodology to manage data by dividing the entire study area into discrete regions. Each region will contain a portion of the study area's data, mainly the data within the area this region occupies. Additionally, these regions are designed to facilitate the display of their data at various LoDs. To achieve this, multiple LoD datasets are generated for each region, enabling selective loading of desired LoDs. The proposed methodology follows a structured three-step approach: firstly, transforming the CFD output cells into a point cloud; secondly, segmenting this point cloud into distinct regions; and finally, generating multiple LoDs for each region through a voxelization process.

3.2.1 Point Cloud

The first step entails converting the CFD cells into a point cloud. While OpenFOAM has the capability to supply the center of each cell, this information might not be available within the dataset and hence needs to be derived from the provided data. The transformation of each cell into a point is accomplished by determining its geometric centroid Weisstein [2023]. For a given set of n points the geometric centroid can be computed using the following formula, where m_i is the weight of point i and x_i denotes the position of point i: Weisstein [2023]

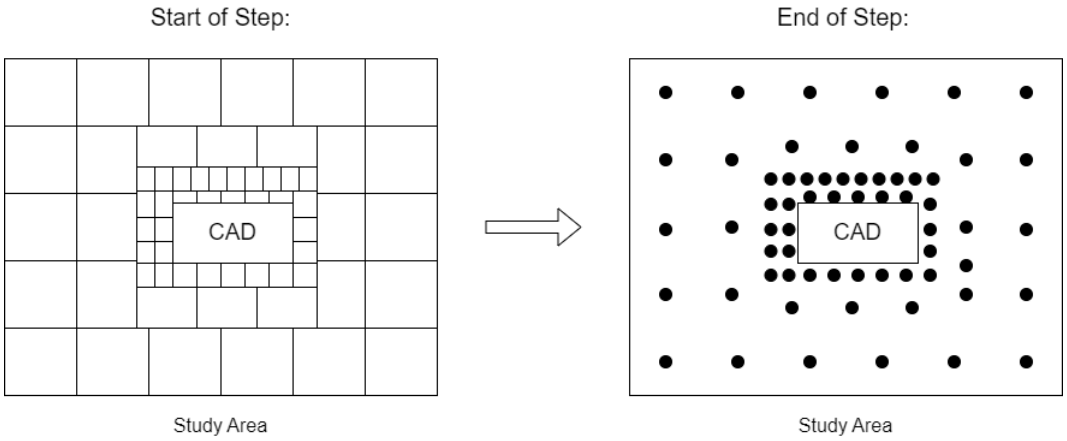


Figure 3.4: Each cell in the CFD result is turned into a point. The final result is a single point cloud within the study area.

$$x = \frac{\sum_{i=1}^{n} m_i x_i}{\sum_{i=0}^{n} m_i} \tag{3.1}$$

In the context of this thesis, each point is of equal value when determining the cell's center. Consequently, each point i is given the same weight w_i. Therefore, the formula can be simplified to the following: Weisstein [2023]

$$x = \frac{\sum_{i=1}^{n} x_i}{n} \tag{3.2}$$

Given that each cell needs to be **convex**, it is guaranteed that the geometric centroid falls within the cell. Furthermore, the quantities of interest associated with each cell c are now attributed with the point i generated from cell c. The resulting data set is a point cloud with a number of points equal to the number of cells in the CFD simulation.

This step provides a reduction in memory requirements. Initially, the data necessitated storage for all individual points, a list of points comprising a face, and another list of faces constituting a cell. However, after the transformation into a point cloud, only a singular point per cell needs to be stored.

3.2.2 Study area discretization

When the entire area is represented as a singular large point cloud, there is no efficient method to swiftly determine which data to load/unload from memory at any specific instance. To identify all the points closest to the user, a calculation of each point's distance

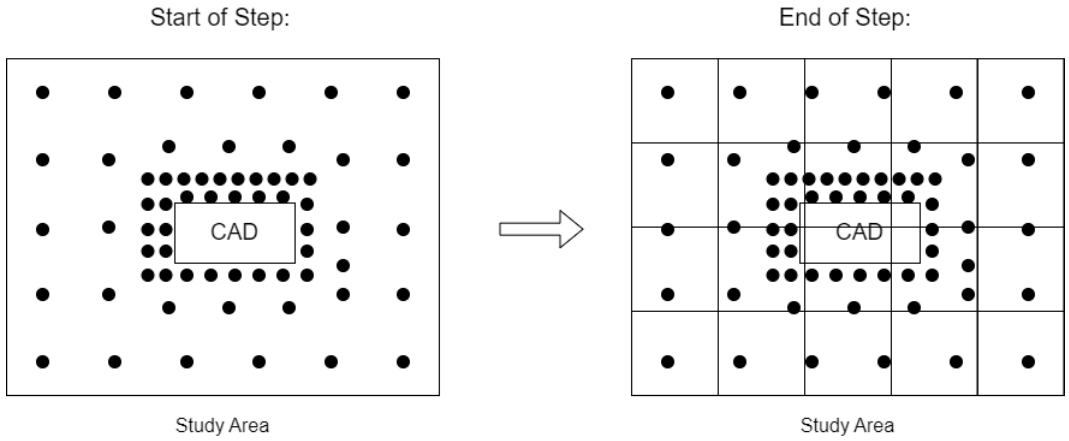


Figure 3.5: The entire study area is split into regions of equal size.

from the user is necessary. To facilitate more manageable data handling the study area is partitioned into discrete regions (see Figure 3.5). Instead of assessing the distance of each point, it becomes feasible to compute the distance to each region. Consequently, data within that region can be efficiently managed based on requirements. To discretize the study area into regions, this thesis proposes dividing the area into cubes. This approach is primarily adopted for two key reasons.

Firstly, the textures derived from the data in future stages must adhere to specific dimensions, mainly set at $2^n \times 2^n \times 2^n$, where 'n' represents any positive integer. This requirement necessitates an equal number of data points in all directions. To maintain proportional consistency in all dimensions, the data points should represent an area which is equal in all dimensions (width, height, depth). If the region is imbalanced in one or two dimensions, the resulting data points in the texture will also exhibit dis-proportionality in those respective dimensions. The rationale behind utilizing cube-shaped regions is more thoroughly elucidated in Section 3.4.2, where the rendering process of the data is explained, highlighting the importance of maintaining proportional consistency in all dimensions.

Secondly, critical consideration is that a region should be loaded into memory based on its proximity to the user, irrespective of the user's viewing direction. Notably, the loading process involves loading the entirety of a region or none of it. As depicted in Figure 3.6, certain regions, although containing substantial portions outside the user's PoV, necessitate loading in their entirety. This results in the loading of undesired data into memory. While cube-shaped regions do not entirely eliminate this issue, they present a compromise that strives to maintain an equitable trade-off as much as possible. By employing cube-shaped regions, this approach aims to mitigate the problem by offering a more balanced compromise between loading unnecessary data and optimizing memory usage, considering various viewing angles and minimizing the amount of extraneous data loaded into memory.

However, due to the dimensions of the study area being fixed, achieving regions that are a perfect cube (with all sides equal) might not be feasible. Consider the scenario illustrated in Figure 3.7, where each region possesses dimensions of 100x110, deviating from a 100x100 perfect cube. This deviation is necessary to cover the entire study area, as adhering strictly to regions of a 100x100 dimension would leave parts of the study area uncovered by the regions. Although it's possible to fill these gaps with regions of alternate dimensions, doing

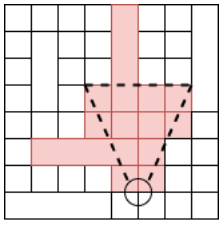


Figure 3.6: Study area split into regions, not all of which are cubes. The dotted area indicates the users Point of View. Red regions need to be loaded into memory.

so adds complexity and is thus deemed undesirable for the sake of simplicity. It is important to note that multiple dimensions could fulfill the requirements (e.g. in Figure 3.7, each region could adopt dimensions of 50x55, creating 24 regions). Deciding which dimension to choose is elaborated on later in this thesis.

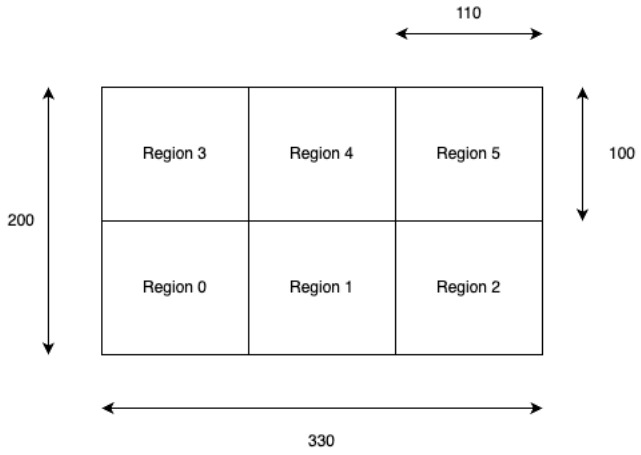


Figure 3.7: One possible way to split the study area into regions

To accomplish creating regions which are as cube-like as possible, a single desired measurement m is provided by the user. m is the desired width, height and depth of all regions. From this measurement the number of times the study area can be subdivided in each direction $d = \{width, height, depth\}$ is determined through the following formula:

$$num\_regions_d = round(\frac{study\_area_d}{m}) \tag{3.3}$$

The result needs to be rounded since the subdivision needs to divide the study area a *whole* number of times (no fractions). Then, to determine the actual width, height and depth of each region the study area is again divided, however this time by the number of times it needs to be subdivided in that direction:

$$region\_size_d = \frac{study\_area_d}{num\_regions_d} \tag{3.4}$$

Using these formulas results in knowing into how many regions the study area will be subdivided and the dimensions of all regions. All regions then have the following dimensions $region\_size_{width}$, $region\_size_{height}$, $region\_size_{depth}$ and the total number of regions can be calculated through $num\_regions_{width} * num\_regions_{height} * num\_regions_{depth}$.

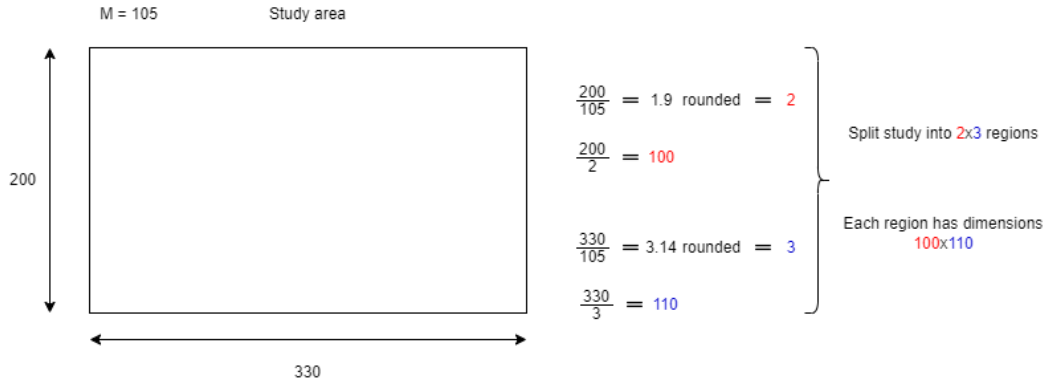


Figure 3.8: This figure shows the calculations performed to get to the number of regions the study area is split into and the dimensions each region gets. This example is for $m = 105$.

This thesis chooses a few different measurements m, mainly 200, 250, 300, 350 and 400, to discuss in Chapter 4. While this thesis will discuss some variance in dimensions, this thesis does not delve into determining the optimal dimensions for a given study, as each dimension set carries its own set of advantages and disadvantages, necessitating a contextual evaluation beyond the scope of this thesis.

After the segmentation of regions, the points within the point cloud are organized into their corresponding designated regions. This process involves iterating through all the points in the point cloud. The algorithm 5.2 is employed to ascertain the specific region to which each point belongs. As a result, each region r encompasses a defined cube-like area within the study region, and all points falling within the boundaries of this area are allocated to the respective region r.

For every region, its position and the points assigned to the region are written to file to be used when generating the LoDs for each region. The position of a region is defined as the location of its bottom left corner. For Figure 3.7, Region 0 would have location $(0,0)$, Region 1 location $(110,0)$, ..., Region 5 location $(220,100)$. Furthermore, a single file *dimensions* indicating the dimensions of each region is saved.

For a study area split into N regions, the structure looks as follows:

```
Regions
├── Region0
│   └── bounds
├── Region1
│   └── bounds
├── ...
├── RegionN
│   └── bounds
└── dimensions
```

3.2.3 Region LoDs

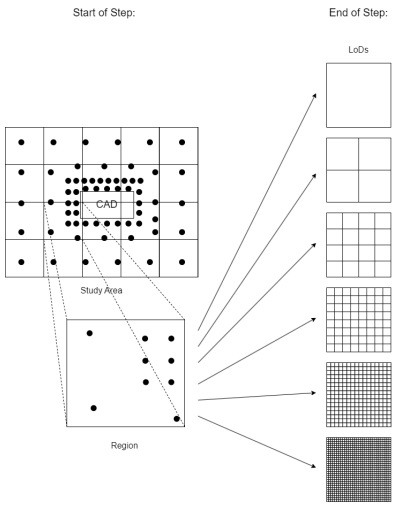


Figure 3.9: Each region gets turned into multiple LoDs.

Following the preceding step, the study area and data have been discretized into a number of distinct regions. However, within each region, the data lacks a structured organization. The absence of such structuring limits the loading process, as either all the data or none of it can be loaded from a given region. By transforming the data into a structured dataset, it becomes feasible to load specific parts of the data without necessitating a comprehensive search through the entire dataset.

To structure the data, the region undergoes a subdivision into equal distinct areas known as voxels. The number of voxels used to subdivide a region is contingent upon the desired LoD. For a designated LoD_d, the region is partitioned into 2^d voxels along the width, height, and depth dimensions. Consequently, an LoD_d is comprises of $2^d \times 2^d \times 2^d = 8^d$ individual voxels. Each voxel can be used to store some data point for that LoD. Figure 3.10 provides a visualization of LoDs ranging from 0 to 5, albeit depicted in a two-dimensional representation for simplicity. For each region a number of LoDs are created.

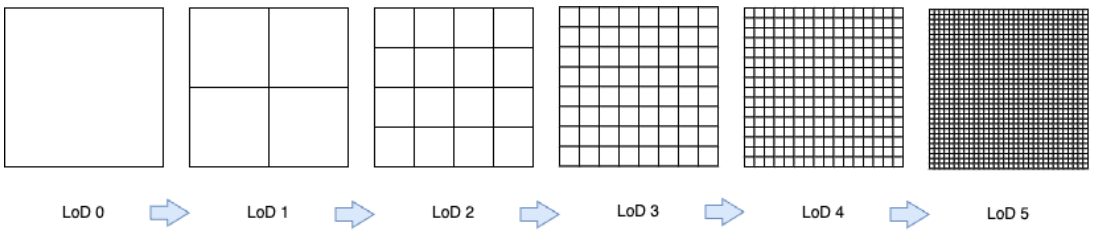


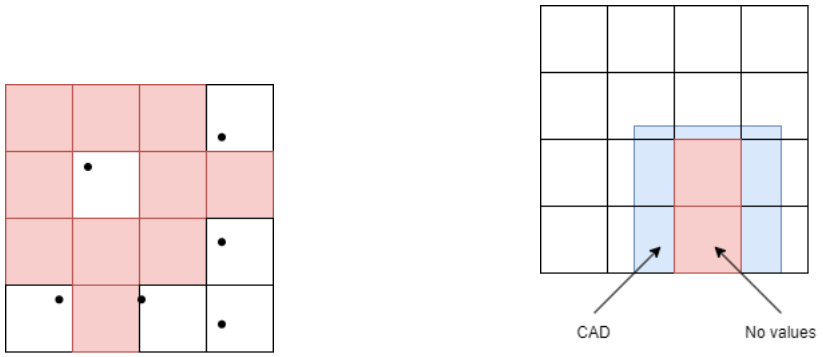
Figure 3.10: Different LoDs

For every quantity of interest q, a value v_q is attributed to each voxel. The value v_q represents the average value of q for all points that fall within the voxel's boundary. In the current methodology, equal weighting is allocated to all points within the voxel to calculate the value v_q. Alternative methodologies that involve assigning different weights to points based

on various parameters such as distance from the center are not explored. This decision is primarily driven by the pursuit of simplicity and a reduced need for extensive calculations.

The adoption of a structured data format for storing data values eliminates the necessity to explicitly define the location of each data point. Instead, for a specified LoD_d, determining the location of each data value only necessitates knowledge of the bottom-left coordinate and the dimensions of the region (see algorithm 5.1).

The scenario can occur where no points fall within a voxel's boundaries. There are two scenarios where a voxel might contain no points. Firstly, a voxel might happen to be positioned outside the center of any cell (see Figure 3.11a). As the LoD increases, the voxels within the region become smaller. This can result in voxels smaller then the cells, particularly in the outer regions of the simulation where cell sizes are larger. Consequently, no cell center falls within the voxel resulting in no points being inside the voxel. Secondly, a prism might fall within a CAD geometry (see Figure 3.11b). Since no CFD cells reside within CAD geometries, there won't be any points falling within the voxel in this situation. Presently, an efficient method to discern between these cases is unavailable (further elaborated in Chapter 5).



(a) (b)

Figure 3.11: Visual representation of the two scenarios in which voxels have no value. (a) spares set of data values in the region. (b) cells inside CAD geometries.

However, it is imperative that every voxel possesses a value. Therefore, if a voxel v_d withing LoD_d is devoid of a value, it references voxels from a lower LoD (LoD_{d-1}). By definition, the voxel in LoD_d is encompassed within one of the voxels in LoD_{d-1}. Consequently, the value of the corresponding voxel from LoD_{d-1} is assigned to the no-value voxel v_d. This necessitates the systematic creation of LoDs from low to high, as higher LoDs occasionally depend on values from lower LoDs to ensure every voxel receives a value.

For each quantity of interest the results for each LoD are written to file to be used later in the program during run-time. For a quantity of interest k with M LoDs the file structure would simply be:

```
k
├─ 0
├─ 1
├─ 2
├─ ...
└─ M
```

3.2.4 Resulting File Structure

For a data set containing N regions and M LoDs the folder structure looks as follows:

```
Regions
├── Region0
│   ├── k
│   │   ├── 0
│   │   ├── 1
│   │   ├── 2
│   │   ├── ...
│   │   └── M
│   ├── U
│   ├── ...
│   ├── p
│   └── bounds
├── Region1
├── ...
├── RegionN
└── dimensions
```

Every *region* has its own subfolders, 1 for each quantity of interest (k, U and P in the example above). Each of these folders contains a file with a number which indicates the level of the tree that file represents. In addition each *region* contains a file with the bounds, which contain the regions x, y, z position of its bottom-left corner. Furthermore, a dimensions file is provided which contains the dimensions of each region.

3.3 Introduction to Unity

This section provides a brief overview of how Unity works to gain the necessary understanding for this thesis.

Unity works with *game objects* that can be placed into the world space. These are the simplest entities and only contain a location, rotation and scale; together called a transform. This entity does nothing except exist in a space in the world. From there, components can be attached to *game objects* to determine what a *game object* does. Often components are user written scripts. For example, a script can be coded to alter the y rotation of the *game object*, making it look as if the object is rotating around the y-axis.

All Unity scripts placed onto *game objects* go through a life cycle. Since the entire life cycle is very complex, only the relevant parts for this thesis are presented here (Figure 3.12). For a full overview please go to the following link: https://docs.unity3d.com/Manual/ExecutionOrder.html

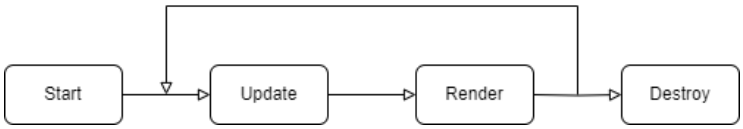


Figure 3.12: The life cycle for Unity Script

The first step in a scripts life cycle is the start function. This function is used to initialize

values. The next function is the update function, here the values of the *game object* can be updated. For the example above, the y rotation of the *game object* should be updated in this function. In the Render part of the life cycle the *game object* is rendered to the screen (if required). The life-cycle keeps looping over the Update and Render functions until the *game object* gets destroyed. When a *game object* gets destroyed it calls its Destroy function which should be used to free up memory and other actions needed when a *game object* is removed.

Furthermore, *game objects* can be activated or deactivated. Deactivated *game objects* are not destroyed, but do not continue their life-cycle, they are also not rendered. *game objects* can also contain children *game objects*. This is not the same as attaching a component to a *game object*. Children of a *game object* that is deactivated are also automatically deactivated.

3.3.1 Build-in Components

Unity also has build in components which can be added to *game objects* . These components are programmed by Unity to accomplish a specific task. These are provided as many of them are frequently used in many games.

Camera

One example of a build-in component is a camera component. Attaching the camera component to a *game object* makes that *game object* function as a camera which the user can use to view the scene while the program is running.

While the camera component provides the ability to view the scene by default it only shows the scene from a single view point. However, for visualizing CFD results it is desired to be able to view the scene from any view point. This requires the addition of a script component which defines how the camera can be moved through out the scene. Notice that both the Camera component and the script component are added to the same *game object* such that the script component allows the user to move the *game object* through the game world and the Camera component makes the *game object* act like a camera.

LODGroup

An LODGroup is a build-in Unity Component meant to handle swapping between different LoDs automatically. Each LoD within an LODGroup is given a % at which it becomes visible. This % refers to the height ratio between the GameObject's screen space height to the total screen height. i.e. If the transition to LoD1 is set for 50%, LoD1 will be rendered when the GameObject takes up 50% of the screens height.

3.3.2 Rendering Meshes

This section provides a brief overview of how a generic mesh is rendered in Unity. Furthermore, it explain what shaders are and how they are used during rendering. The actual meshes and shaders used to visualize the CFD results are discussed in Chapter 3.4.

In Unity, the visualization of a mesh necessitates the attachment of two components to a *gameobject*: (1) *mesh filter* and (2) *mesh renderer*. The *mesh filter* component serves the purpose of managing and storing the mesh's structural elements, e.g. its points and faces. The *mesh renderer* component is responsible for the actual rendering of the mesh structure onto the

screen. This rendering process is achieved by employing shaders, which are specialized programs executed on the Graphics Processing Unit (GPU).

A shader operates through two primary steps: the vertex shader and the fragment shader. The vertex shader executes a specific operation on each vertex of the mesh it is applied to. As an example, in figure 3.13a the vertex shader would run on v1, v2, v3 and v4. The fragment shader runs on all fragments (explained later) that cover the mesh (see Figure 3.13b). Importantly, the execution of shader code occurs in parallel. This means that as the shader code runs on a vertex or fragment it has no awareness of the other vertices and fragments.

(a) (b)

Figure 3.13: (a) A Mesh face. (b) The fragments that overlap with the mesh face.

Each vertex shader receives as input a vertex and all associated data linked to it, such as normals, color, and UV channels. The output of the vertex shader is a user-defined data structure denoted as $frag$, which can be anything. This structure typically includes the transformed position of the input vertex, its normals, and potentially some UV channels. Frequently, the primary function of the vertex shader is to handle the transformation of vertex positions from world space to screen space.

The fragment shader runs on every fragment that overlaps the mesh. A fragment is essentially a pixel on the screen and is created by Unity's internal code. In figure 3.13b each of the regions is a fragment. As input the fragment shader takes an instance of the $frag$ data structure produced by the vertex shader. As output the fragment shader produces a RGBA color (see Figure 3.14). Notice that frequently there are more fragments than vertices. For each fragment a $frag$ data structure is constructed by linearly interpolating from the $frag$ data structures produces by the vertex shaders. Within the fragment shader the $frag$ input along with user written logic can be used to determine the resulting color value. The resulting color value will be drawn to the screen for that pixel (fragment).

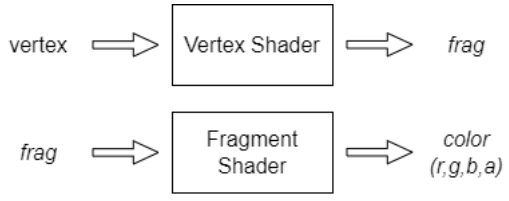
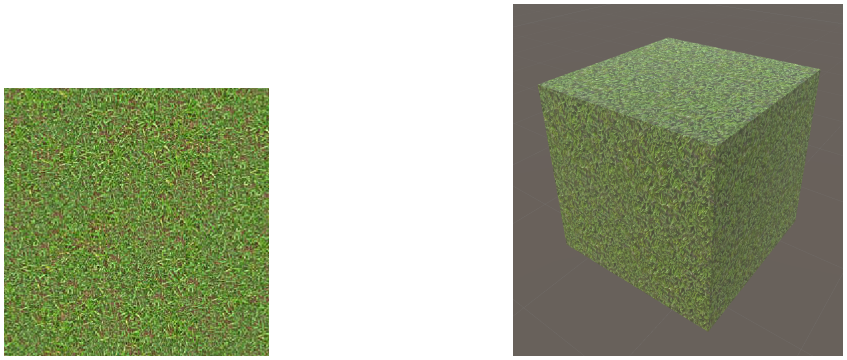


Figure 3.14: The input and output of the vertex and fragment shaders.

It can be useful to provide extra information to the shader besides the mesh structure. This is accomplished through the user of materials. Materials are data containers which make their

data accessable for shaders. This can be something like a color. By creating two materials, one with a yellow color and one with red color, the same shader can be used with different material input to display both a yellow and red object.

A common data input of a material is a texture. Textures are matrixes of data which can be used by the shader. There is no clear definition of what a texture needs to do since at its core it is a bunch of numbers which can be used as liked by the developer. Most commonly textures are used to provide details on the surface for meshes. As seen in Figure 3.15, when applying a texture to a cube the cube looks like it is covered by grass.



(a) (b)

Figure 3.15: (a) A 256 $\times$ 256 texture that looks like grass. (b) The results of applying the grass texture to a cube.

Figure 3.16 show cases how the numerical values of a texture are stored behind the scenes. This is a 2 $\times$ 2 2D texture being applied to a mesh and the result. Notice how the shape of the texture gets stretched to match the shape of the mesh.

Texture can also come in the form of a 3D texture. Figure 3.17 show cases how data is stored in a 3D texture. Textures should always have dimensions of powers of 2 (e.g. 4x4x4, 256x256x256). However, it is noteworthy to mention that the memory requirement of a 3D texture increases very quickly as its dimensions increase. A 16x16x16 (2^4) dimension texture has a 128KB storage requirement where as a 256x256x256 (2^8) dimension texture has a 512MB storage requirement. This requires caution to be exercised when using higher dimension textures.

3.4 Visualization

This section describes the steps taken to actually visualize the CFD results. It takes the pre-processed data and turns it into a real time visualization process. During visualization the user moves the camera freely through out the environment in which the CFD simulation was preformed. At any point only a single quantity of interest can be visualized. Furthermore, this thesis only explores iso-surfaces and volume rendering visualization for scalar values and barbs for vector values. The visualization is completely accomplished within Unity.

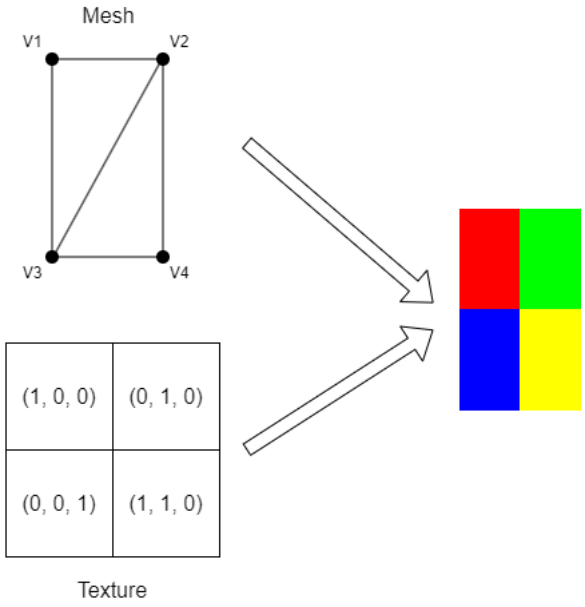


Figure 3.16: A numerical representation of a texture being applied to the mesh. The values in the texture are colors of the format RGB.

3.4.1 Pre-Runtime Setup

The initial setup process involves integrating CAD geometries, provided by OpenFOAM in STL format, into the Unity scene. While Unity does not natively support STL files, they can be converted into OBJ files STL [2023], a format compatible with Unity. The resulting OBJ files can be imported into the scene. It's important to note that certain adjustments might be necessary, such as rotating the OBJ files, as the orientation of the y and z axes could vary between different programs. For this thesis, the resulting imported objects required a rotation of -90 degrees around the x-axis and 180 degrees around the z-axis. Furthermore, a *game object* with a camera component and the movement script defined earlier is added to the scene. The resulting setup can be seen in Figure 3.18. By placing the CAD geometries and camera into the scene before run-time it reduces the initial loading time required.

Additionally, an empty *game object* is added to the scene with only a script attached to it. This is required due to the fact that the only way to run user defined code is through scripts attached to *game objects* . This *game object* will be refered to as the main *game object* and provides the backbone on which the dynamic data loading is performed. The functionality of the script is further explained in the next sections.

Lastly, the user needs to specify some values. While the study area has been discretized into regions, regions need to know when and which data to load/unload into memory. To accomplish this a load, render, and destroy distance for each LoD is specified by the user. Additionally, the file path to the folder containing the pre-processing results is required.

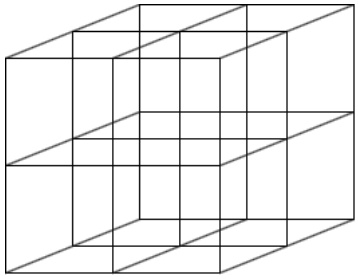


Figure 3.17: A visual representation of how a 3D texture can be thought off. This is a $2 \times 2 \times 2$ texture.

3.4.2 Dynamic regions

During visualization, it is crucial to understand that the entire dataset of a region is not loaded into memory at once. Instead, the data corresponding to a LoD within a region is loaded in or out of memory. At any given time, zero, one, or multiple LoDs for the regions can be present in memory. However, only a singular LoD can be displayed on the screen simultaneously.

To determine which LoD to load and visualize, the program needs to assess the distance between the camera and each region. This necessitates the setup of regions. Within the *start* function of the main *game object* , each region is configured to ensure effective utilization in subsequent operations:

1. The boundary of each region is read and its center is computed.
2. A Unity *game object* is created for each region and positioned within the world at the location of the region.
3. An LODGroup component is attached to the newly created *game object* .
4. A reference to the region is stored in the main *game object* .

During runtime, LoDs within the previously created regions are dynamically managed. This process occurs in three primary phases: loading, rendering, and destruction.

During each update phase in Unity's lifecycle, the main *game object*'s script calculates the distance from the camera to each region. This distance is then compared to user-defined distances set earlier. If the camera's proximity to an LoD of a region is closer than the defined load or render distance, the corresponding data is loaded or rendered, respectively. Conversely, if the camera surpasses the defined destroy distance, the data is unloaded.

A region comprises of multiple LoDs. As each LoD should be loaded, rendered, or destroyed at different distances relative to the camera, distinct distances are designated for each LoD. For instance, consider the variations in distances for LoD1 and LoD2 showcased in Figure 3.19a and Figure 3.19b. While these distances differ for each LoD within a region, they remain consistent across various regions. In other words, $LoadDistance_A = LoadDistance_B$ for LoD_1 in region A and region B.

The distance between the region and the camera is determined as the distance from the camera's position to the center of the region. This choice of using the center of the region rather than its edges was made due to the computational efficiency of a single calculation.

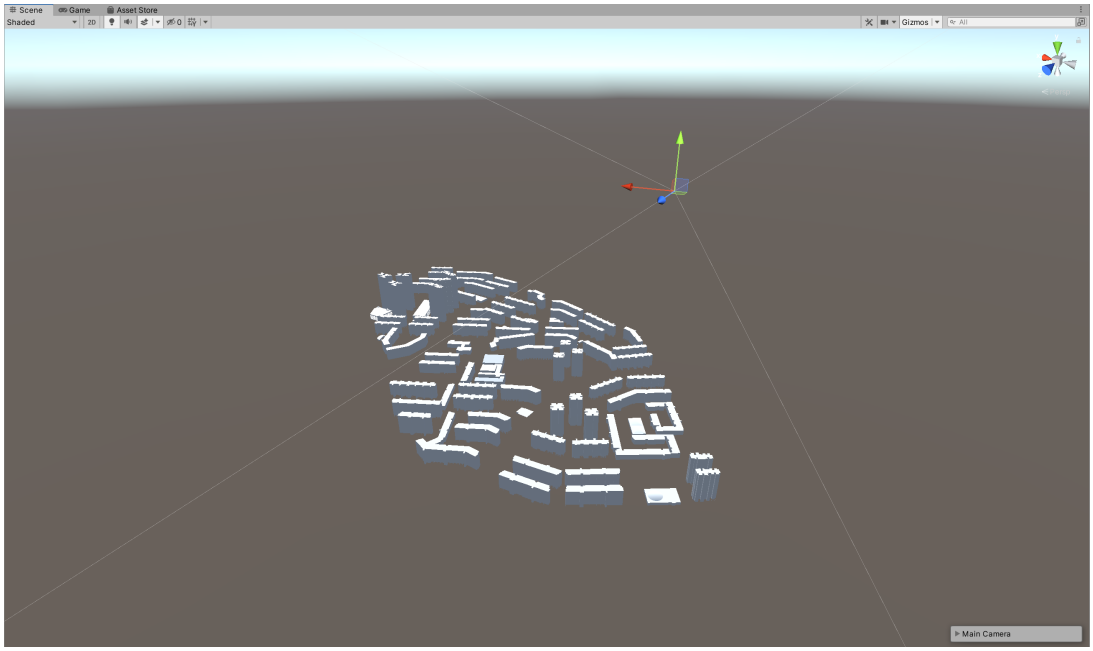


Figure 3.18: The scene after the CAD geometries and the camera have been added.

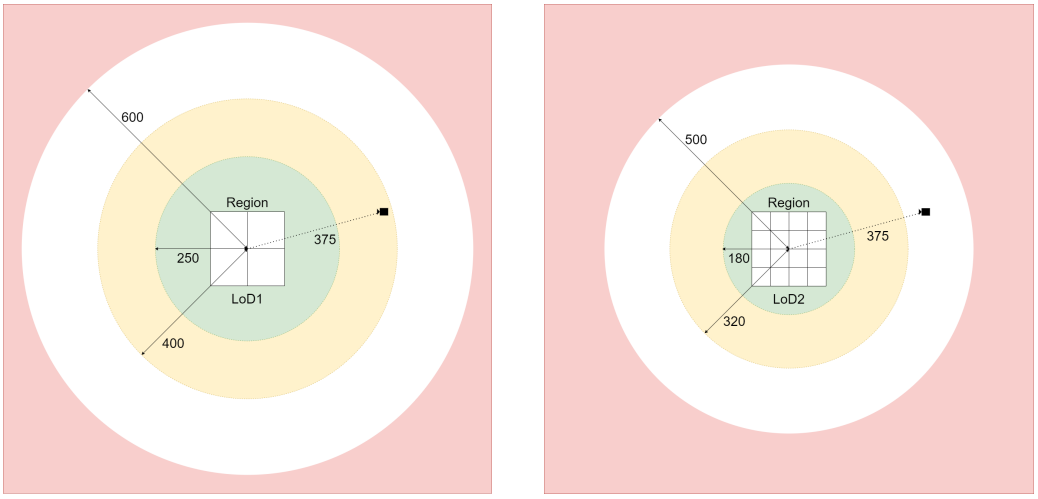
Given that this computation needs to be performed for every region in each iteration of the life cycle loop, the optimization of this calculation's speed is critical. However, this method results in the area in which the region becomes visible to be a spherical area around the center of the region (refer to Figure 3.19), while the region itself is a cubic structure. Consequently, this requires the user to get slightly closer to the region when approaching from a corner

Moreover, the user specified distances are represented as ratios rather than actual distances. These ratios are subsequently multiplied by the size of a region to determine the actual distance (see Figure 3.20). The size of a region is defined as half the average of all its dimensions. This approach of using ratios instead of fixed distances is to facilitate the variations in region sizes across different projects. Larger regions may necessitate being loaded from greater distances due to their visibility from farther away. Without this ratio-based approach, recalculating these distances for each project would be required. While some adjustments to these ratio-based values might still be necessary, they serve as a fundamental guideline adaptable to various situations.

Loading

If any LoD within a region falls within the user-defined loading distance, the data associated with that specific LoD will be loaded into memory. This process involves asynchronous reading of the data from file. The asynchronous reading prevents the program from freezing while the, potentially large, data is being loaded, ensuring smooth execution until the process is complete. Only the LoD data for the current quantity of interest is loaded.

While reading the data into memory, various global statistics of the dataset are computed,



(a) (b)

Figure 3.19: The different load, render and destroy distances for different LoDs of a region. The data for an LoD is loaded if the camera is within the yellow region, displayed in the green region and destroyed inside the red region. (a) LoD1 (b) LoD2

including but not limited to minimum and maximum values. These statistical computations become crucial for subsequent operations. Following the data's successful loading into memory, the subsequent steps depend on whether the current quantity of interest pertains to a scalar or a vector.

Scalar Loading

Upon successful loading of the scalar data the following happens. A game object tailored for the LoD is instantiated, with both a Mesh Filter and a *mesh renderer* component attached. The mesh assigned to the Mesh Filter is a native Unity cube mesh with 8 vertices and 12 faces (2 faces per side of the cube). The *mesh renderer* uses a custom shader which is discussed in more detail in the rendering section.

The earlier loaded LoD data is converted into a texture. Since the data and a texture have the same structure (as described earlier) the value at index i of the data can be assigned to index i in the texture. However, textures require that values are between 0 and 1 (normalized). Therefor, as the values are being assigned from the data to the texture they are normalized. This is accomplished by taking value i from the data, subtracting the minimum data value and dividing it by the range (maximum value - minimum value) of the data. Notice how the minimum value and range are taken from the *entire* data currently read into memory. This is why these values where stored when the data values were read in. Furthermore, if the minimum or maximum value change during run-time all textures will need to be recalculated. The generated texture is then given to the material of the *mesh renderer* component. Finally, the game object is added as a child to the regions game object and assigned to the LODGroup of this region.

Vector Loading

Upon successful laoding of the vector data the following happens. A parent game object pgo_l representing LoD l is created. Following this, for each vecter value v within the LoD a

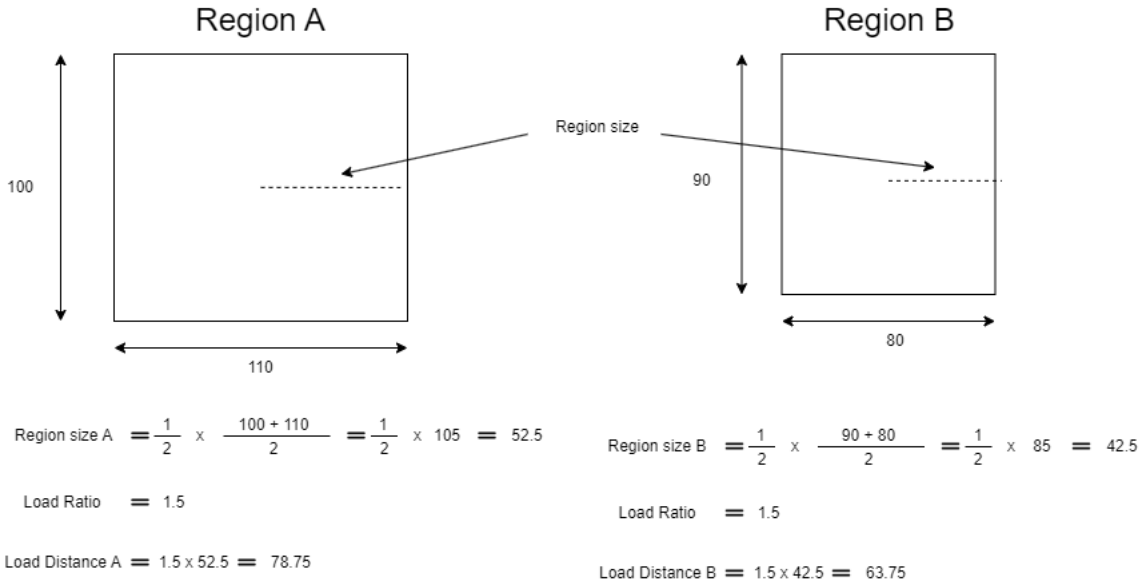


Figure 3.20: Two regions in two different projects. The size of the regions are different. While the load ratios are the same, the actual load distance gets calculated accordingly.

game object *go* is instantiated, see Figure 3.24. These individual *go* instances are positioned within the region. The position is determined based on the index and can be computed using the formula 5.1. Moreover, each *go* instance is oriented to align with the direction of the vector v it represents.

Every *go* game object is equipped with a *mesh filter* and a *mesh renderer* component. The Mesh Filter references an arrow mesh sourced from David Ball's collection Bell [2018]. Meanwhile, the*mesh renderer* utilizes Unity's default material and shader. The RGB color of the material assigned to the *mesh renderer* is configured to reflect the XYZ components of the vector v, that is $color = (v.x, v.y, v.z)$, where v is first normalized.

Each *go* is then attached as a child object to pgo_l. Furthermore, pgo_l is attached as a child to the region's game object. Finally, pgo_l is deactivated to make neither it nor its children render at this point.

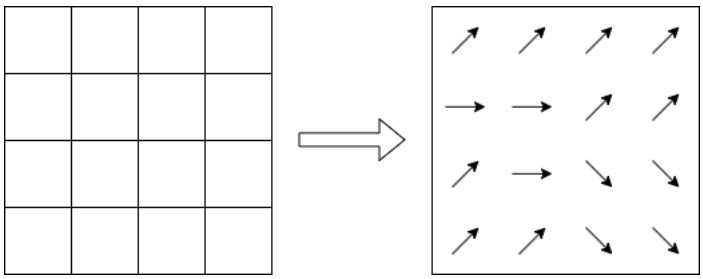


Figure 3.21: For each voxel in the LoD a game object (*go*) is created. In this figure a game object is represented through and arrow.

Rendering

When the camera surpasses the predefined rendering distance threshold specific for a particular LoD, the associated data for that LoD should be rendered. It's essential that the rendering distance should always be less than the loading distance to ensure the data for this LoD is loaded.

By default, all the data loaded would be rendered. However, the user should be able to adjust which data they want to see at any given point. Therefor, the user can provide some addition value of which data they currently want to view. For an iso-surface this is a single scalar value. For volume rendering this is a range specified by the scalar values a & b where $a < b$. For barbs, 4 ranges are provided, one for each of the axis (x, y, z) and one range for the magnitude of the vectors. Furthermore, for the barbs a scale value is provided in the form of a scalar value. The barb meshes are scaled in size based on the scale value provided.

Iso-surface

The visualization of the iso-surface within a region is facilitated by a custom shader, the foundation of which was provided by Matias Lavik Lavik [2020]. This shader relies on a material comprising a texture and a scalar value. The scalar parameter determines the value at which the iso-surface is to be rendered. The texture parameter corresponds with the texture created for this LoD during the loading phase.

Within the vertex shader, there are transformations applied to the position of the vertices of the mesh and its normals, transforming them from world space to screen space. To accomplish this Unity provides the functions UnityObjectToClipPos and UnityObjectToWorldNormal for a vertex's position and normal respectively.

In the fragment shader, it utilizes the interpolated position of the vertex as the initial starting point. Using this position, a ray is cast through the mesh. The direction of this ray coincides with the direction from the camera to the point on the mesh, as depicted in figure 3.22. This ray traverses the mesh in a predetermined number of steps, mainly num_steps. For this thesis $num\_steps = 1024$. The length of each step is determined as $\sqrt{3}/num\_steps$. Here, $\sqrt{3}$ signifies the maximum distance across the cube, spanning from one corner to the opposite corner. Increasing num_steps provides greater accuracy, but also slows down rendering.

At each step of the ray's traversal, the shader samples the texture at the current location of the ray. If the value obtained from the texture corresponds to the desired value, the ray marching stops and the following happens. The value at that particular point goes through a transfer function, yielding a RGB color value to be displayed. Subsequent to this, lighting computations based on the vertex's normal are applied to this color. Finally, the shader sets the opacity (A) of this color to 1 (fully opaque), resulting in the RGBA value which is then returned.

Here it becomes clear why the voxels need to be as cube-like as possible and in extension why the regions need to be as cube-like as possible. If the voxel is dispropotionatly large in one dimension the value that the voxel represents gets stretched in that dimension.

Volume Rendering

Volume Rendeing works in much the same way. The vertex shader preforms the exact same operations. Additionally, the ray marching described is also preformed. However, where the iso-surface returned after hitting its desired value the first time, volume rendering keeps going. The transfer function through which the value goes not only provides an RGB

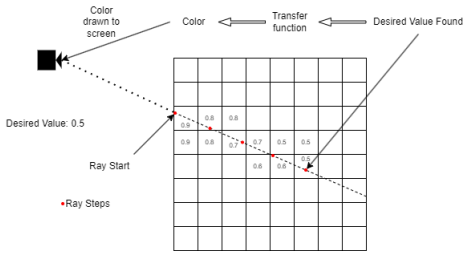


Figure 3.22: Iso-surface rendering. The first step in the ray that find the desired value is returned.

color, but also an opacity A. When the ray marching finds a vlue within the specified range the follwing happens. The value goes through a transferfunction providing a RGB and A value. The resulting RGB value and opacity then go through the equations 3.5 and 3.6. Where *src* is equal to the value calculated up to this point. At the start of the ray marching $src = (0, 0, 0, 0)$. If $src.a >= 1$ the ray marching stops and the current *src* value is returned. Figure 3.23 shows how the ray marching hits multiple desired values. The color and opacity value are then updated at each step. The value of the final step is returned.

$$color.rgb = src.a * src.rgb + (1 - src.a) * col.rgb \tag{3.5}$$

$$color.a = src.a + (1 - src.a) * col.a \tag{3.6}$$

Barbs

When rendering the barbs, the game object pgo_l corresponding to the current LoD l that requires visualization should be activated. Additionally, the scale of the *go* meshes needs to be set equal to the magnitude of the vector value *go* is representing multiplied by the scale value provided by the user. Furthermore, any *go* game object representing a vector that falls outside the boundaries of the four provided ranges must be deactivated.

Destroying

If the camera goes beyond the destroy distance for a specific LoD of a region all data and game objects associated with that LoD in that region need to be destroyed. To destroy a game object Unity provides a GameObject.Destroy() method which both destroys the object and frees its memory. For the data loaded into memory, Unity (or C#) has an automatic

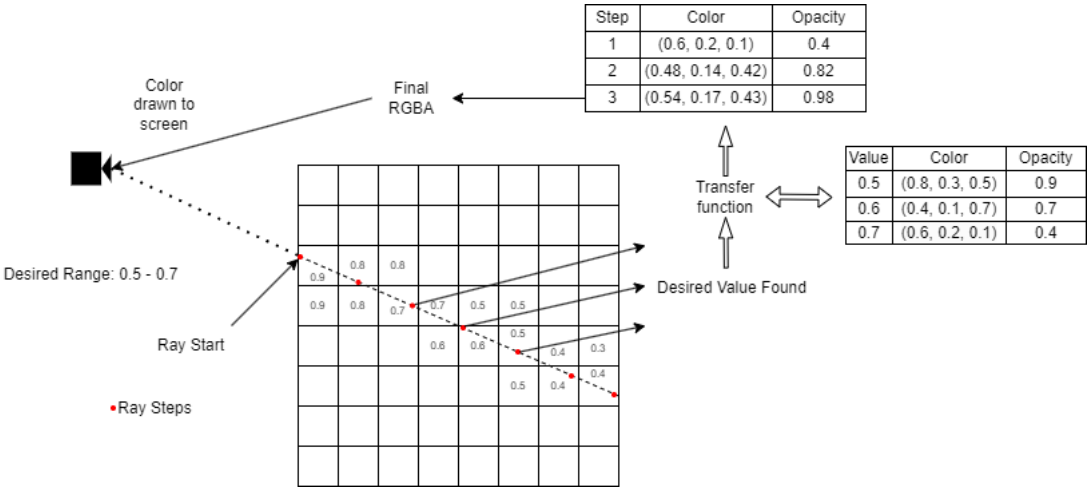


Figure 3.23: All values alone the ray that fall within the range are combined to calculate the final color.

garbage collection system. An automatic grabage collection system automatically frees up any memory which is not references in the program. Therefore, all that is required to free up the memory in which the data is loaded is to set the variable which stores the data to null.

3.5 Unreal Engine

The visualization steps have all been explained in the scope of Unity. However, it is desired that this visualization methodology works in other game engines as well. Therefore, this section will quickly discuss how this concept can be applied in another popular game engine, Unreal Engine.

Like Unity, Unreal Engine uses game objects, materials and textures. This allows for most of the methodology to be used exactly the same way as it was used in Unity. The main differences is whether some of the calculations are performed in the material or in the shader. In Unity the material is only a data container in which no calculations are performed. However, in Unreal Engine the material can perform calculations, such as determining step size of the ray tracing, providing that information directly to the shader. Note that in Unreal Engine the shader does retain the ability perform these calculations.

Furthermore, like Unity, Unreal Engine has build in capabilities to import OBJ files. In addition Unreal Engine also comes with a build in camera game object. The code used in Unity to allow the user to move the camera can also be added to this camera game object (albeit it be in C++ not C#).

In conclusion, almost the entire methodology should be reproducible in Unreal Engine. The main difference should be nuanced implementation details such as how game objects or texture are created.

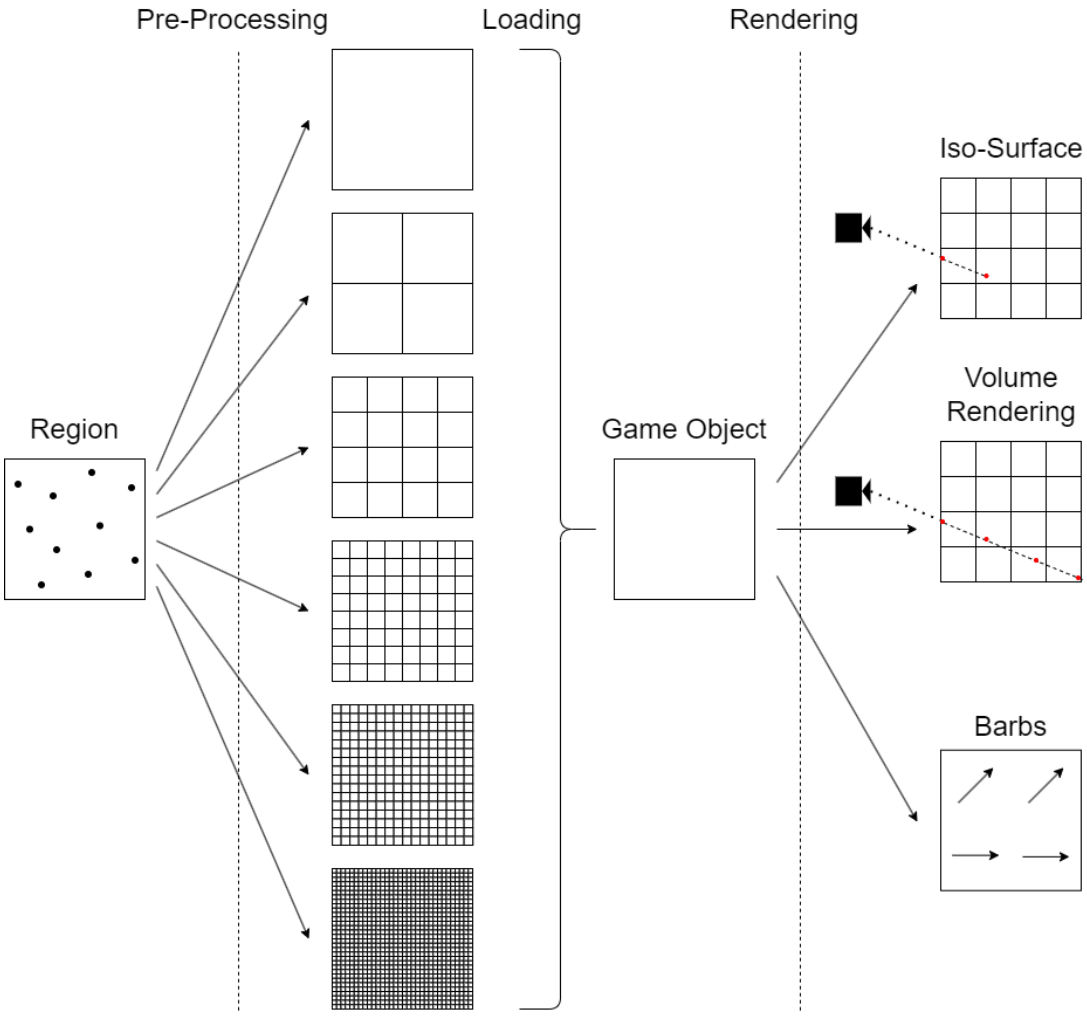


Figure 3.24: This figure shows the general process the data takes to be visualized. First the region data gets turned into LoDs. Secondly, the LoDs can get loaded into the GameObject. Finally, the data gets rendered based on the desired rendering type.

4 Results

In this chapter the result of the methodology will be discussed. This includes the time requirement during the pre-processing to convert the CFD output data format into the required data format for Unity. Furthermore, the visual results of iso-surface and volume rendering for scalars and barbs for vector values will be shown. Finally, the error within the processed data is discussed. All the results are in relation to two quantities of interest, mainly pressure (p) for scalar values and velocity (U) for vector values. Furthermore, for this thesis LoDs 0 to 5 were generated for each region.

The study area used for this thesis comes from the results generated in the paper written by Aviva Opsomer Opsomer [2020]. The study area is located in Clementi, Singapoure and encompasses and area of around 4.7km by 7km and 720m in height. The dataset is composed of 27,365,873 cells.

4.1 Pre-Processing

All pre-processing steps were processed using Python 3.7.7 on a PC with the following specifications: NVIDIA GeForce RTX 2060 graphics card, 8GB RAM and Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz (8CPUs).

4.1.1 Split sizes

As mentioned in 3.2.2, this thesis will analyse the pre-processing results for a couple of different region split sizes. Table 4.1 shows the input measurement m in the first column, which is the desired dimension for a region. The other columns show the number of regions which the study area is split into. Table 4.2 shows the actual dimensions of a region when split with the desired measurement m. As seen in the last column, as the split size decreases linearly the total number of regions increases exponentially.

Desired split size (m)	Number of regions in width	Number of regions in height	Number of regions in depth	Total number of regions
400	12	2	18	432
350	14	2	20	560
300	16	2	23	736
250	19	3	28	1596
200	24	4	35	3360

Table 4.1: Based on the split size (left column), the number of regions in the width, height and depth are shown. The last column shows the total number of regions the study area is split up in.

Both the time and memory stats will be discussed in relation to these split sizes. While these split sizes will be compared to one another there is no claim made about which is optimal. What is optimal can differ based on many variables; such as the time available to perform the pre-processing, the size of the study area, the desired accuracy of the visualization, etc.

Average Split (m)	Width (m)	Height (m)	Depth (m)	Total # of Regions
400	396.43	360.01	391.57	432
350	339.80	360.01	352.42	560
300	297.32	360.01	306.45	736
250	250.38	240.00	251.72	1596
200	198.22	180.00	201.38	3360

Table 4.2: This shows the actual width, height and depth of each region when the study area is split.

4.1.2 Time analysis

The initial stage in the pre-processing involves discretizing the study area into distinct regions. Discretizing the study area is almost instant. However, to efficiently process and store data for later use, a folder for each region are created. Within the region's folder a file containing the boundaries of this region is created. Furthermore, a folder for each quantity of interest is created within the region's folder. This equates to $(2 + quantity\_of\_interest\_count)$ files/folders being created per region. Meaning that for all regions a total of $num\_regions * (2 + quantity\_of\_interest\_count)$ files/folders are created. While $quantity\_of\_interest\_count$ is a variable within the formula, it is a constant within each project. This results in a linear time increase (Figure 4.1b) with respect to the number of regions the study area is discretized into.

Secondly, the points need to be assigned to their corresponding regions. For this part, the algorithm needs to loop over every single points once regardless of the number of regions there are. Furthermore, indexing the point into its corresponding region takes a constant amount of time (see algorithm 5.2) and is not associated with the number of regions there are. Therefor, there is a base time required regardless of the number of regions. Figure 4.1c shows that regardless of the number of regions there is still around 80 seconds required to segment the points into their associated regions. A linear increase is still present due to the fact that for each region a file with the associated points needs to be written to file. This means that the number of files written increases proportionally to the number of regions.

Finally, for each region, multiple LoDs need to be created. Figure 4.1d shows the time it requires to calculate and write these LoD to file. Again it is clearly visible that there is a linear trend between the number of regions and the time it takes to LoD them. Similar to sorting the points into their corresponding region, sorting a point of a region into its corresponding voxel takes constant time. Therefore, the time it takes to create an LoD is correlated to the number of points within the region. Each region will have a different number of points within it, leading to different processing times per region. However, note that the entire study area still contains the same number of points. Therefore, when looping over all regions, the total number of points looped over will be equal regardless of the number of regions. The reason for the linear increase comes from the constant time operations each region needs to complete. A region needs to allocate memory for each LoD and it needs to write this LoD to file.

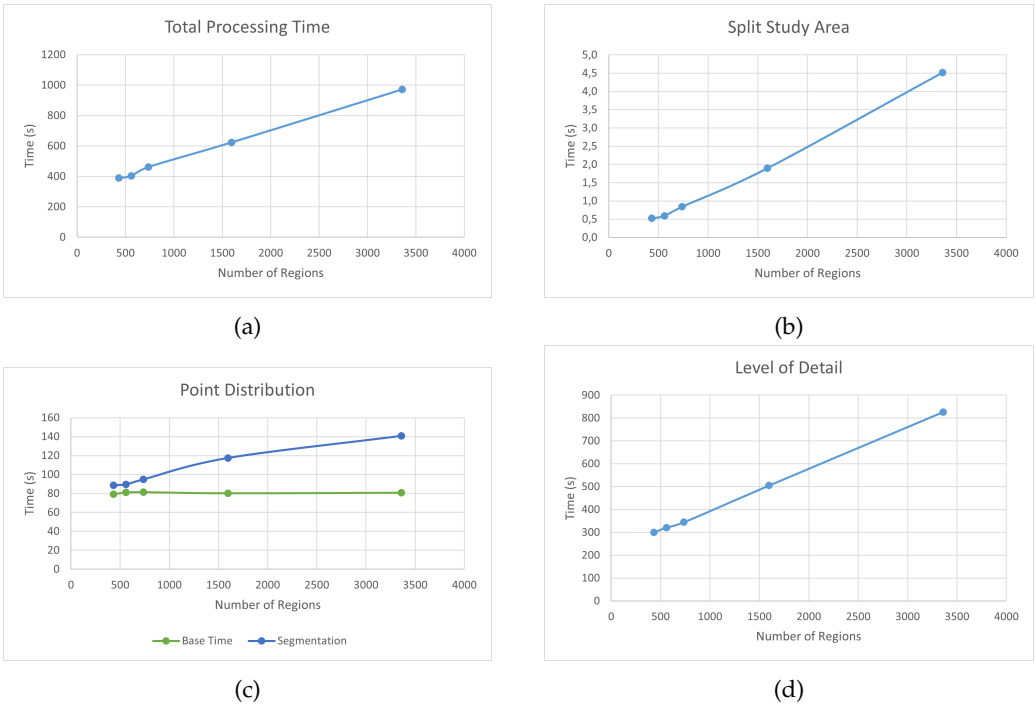


Figure 4.1: (a) The total time required to preprocess the data. (b) Time taken to create folders for all regions. (c) Time taken to assign the points to each region. (d) The time it takes to create all LoDs in correlation to the number of regions.

Figure 4.1a shows the total time taken to pre-process that data for a given number of regions. All operations within the pre-processing steps linearly increased with respect to the number of regions. Therefore, it should be no surprise that the total time also increases linearly.

4.1.3 Data size

During the pre-processing phase the data size is reduced. Through the use of the new structured data format only the data values themselves need to be stored, not their location. The location of each point can be found using the index of the data and the algorithm 5.1. This removes all the data needed to define the structure of each cell in the original CFD results. As seen in figure 4.2, the data requirement is below that of the original data. The data requirement does increase linearly in relation to the number of regions. However, it seems unlikely that the desired number of regions increases to the point that the memory requirement surpasses the original memory requirement.

In spite of the data reduction, the number of data points has actually increased. For instance, at LoD 5, the dataset comprises of 110,100,480 data points over the 3360 regions, which is much greater then the 27 million original data points. Unfortunately, most of these data points fall within spare regions of the original data making them less significant. However, this provides a good sign for solutions that have smaller regions or higher LoDs in dense areas, discussed further in chapter 5.2.

Number of Regions	Memory Reduction (%)
432	84.47
560	83.33
736	81.79
1596	73.90
3360	57.80

Table 4.3: The percentage of memory reduction between the original data and the data resuling from the pre-processing.

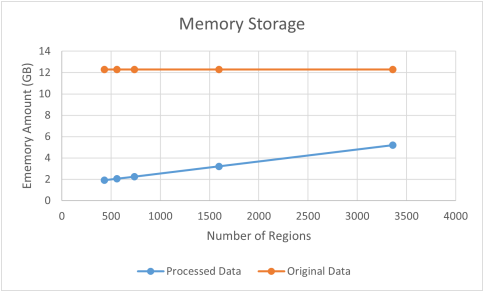


Figure 4.2: The data size of the processed data in relation to the original data size

4.2 User Inputs

As described in Chapter 3.4.1 the user needs to provide some inputs, mainly the load, render and destroy distance for each LoD. This section provides an overview of the inputs used within this thesis. These input where manually tested and this combination was found to work best for this data.

It should be noted that ratios for higher LoDs should always be smaller than ratios for lower LoDs (see Table 4.4 and 4.5). The reason behind this is that higher LoDs should only be loaded and rendered as the camera gets closer to the region. Subsequently, higher LoDs should be destroyed earlier as the camera moves away from the region. Higher LoDs take up more memory and should thus be removed from memory as soon as possible to make space for new data. Additionally, as the camera moves away from the region, lower LoDs might still be visible and should thus not be destroyed.

Table 4.4 shows the ratios chosen for the scalar values. The only exception to the ratio rule is present here. The distance at which an LoD gets rendered is not determined by a distance ratio, but rather the percentage of the screen height that region currently takes up.

LoD	Load Distance Ratio	Swapping Height (%)	Unload Distance Ratio
0	12	10	16
1	10	20	14
2	8	30	12
3	6	40	10
4	3	50	9
5	2	70	8

Table 4.4: Transition points for scalar values.

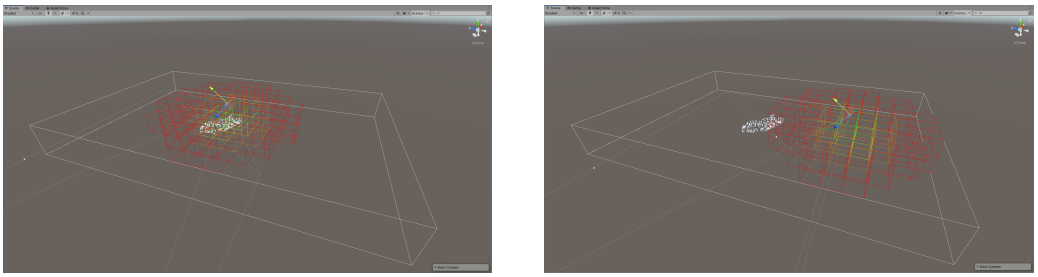
LoD	Load Distance Ratio	Render Distance	Unload Distance Ratio
0	4.5	3.5	11
1	4	2.8	10
2	3.5	2.3	9
3	3	2	8
4	2.5	1.7	7
5	2	1.5	6

Table 4.5: Transition points for vector values.

It should be noted that the destroy ratio is generally larger than the load ratio, at the very least it can never be smaller. The consequence of this is that there are times when the data is loaded while the camera is further then the load ratio. The reason to do this is the keep recently viewed LoDs in memory. As the user has not moved very far from the LoD they might come back into range shortly. By keeping the data in memory for at least some extra time it reduces the number of times the data needs to be loaded.

4.3 Data visualization

The purpose of this thesis was the ability to visualize massive CFD results. In this section an overview of how visualization turned out is discussed. It show cases the visualization of iso-surfaces, volume rendering and barbs. Furthermore, it displays how the visualization change as the LoD change. Additionally, the change in visualization due to parameters is outlined. This is done for both a single region and the entire data set. A single region provides a better understanding of how changing the LoD effects the visualization.



(a) (b)

Figure 4.3: The highlighted regions have one or multiple of their LoDs loaded into memory.

Firstly, an overview of how regions are dynamically loaded. Based on the position of the camera data from different regions is loaded into memory. In Figure 4.3 the regions which are loaded are visualized. As the camera moves from one location in 4.3a to different location in 4.3b the set of regions which is loaded into memory changes. As mentioned in chapter 3.4 the region's data does not get loaded, but rather the LoDs of a region. In Figure 4.3 regions in the center of the cluster of loaded regions have higher LoDs loaded then their outer counterparts.

Furthermore, as mentioned earlier, the data does not get destroyed as soon as it falls outside the loading distance. The destroy distance determines this. In Figure 4.4 this phenomenon

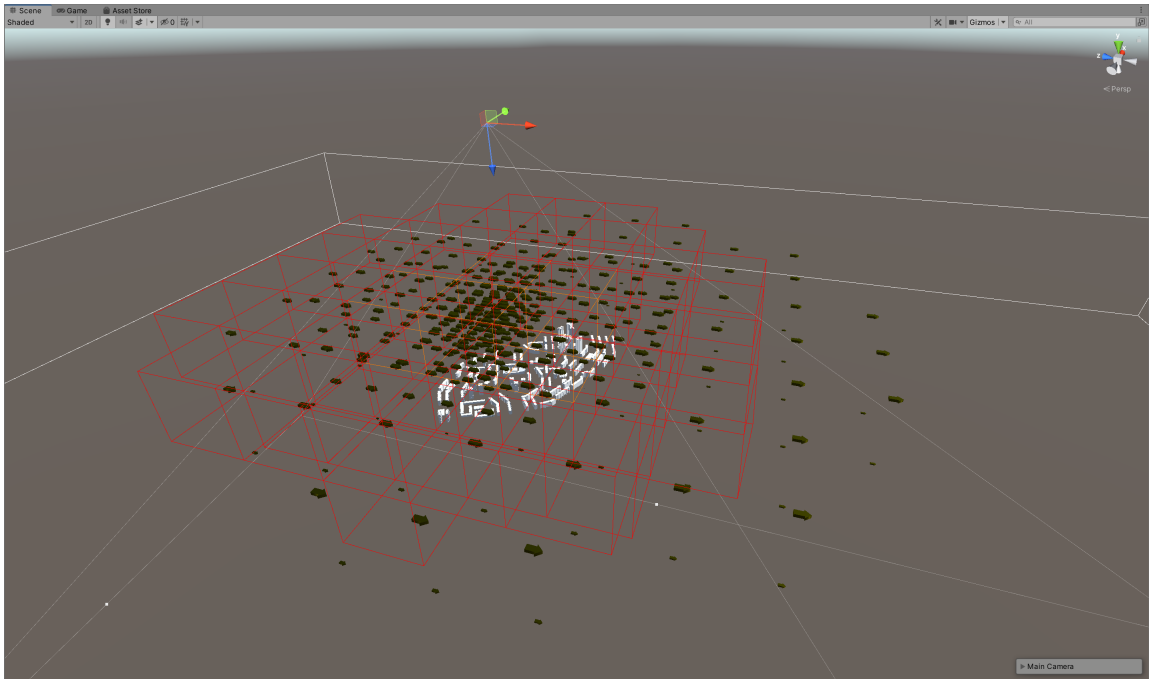


Figure 4.4: Barbs outside the highlighted regions showcases how data is not directly destroyed once the camera moves past its loading distance.

can be seen as there are arrows showing in regions that are not currently in loading range (highlighted regions).

4.3.1 Visualization of Scalars

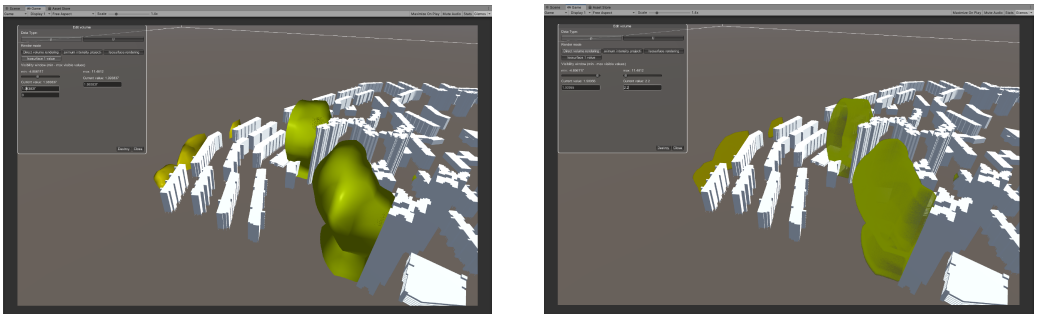
The purpose of iso-surfaces and volume rendering is to display which areas contain a set of values, either with a singular value or a range of values. Figure 4.5 depicts an example of each.

As seen in Figure 4.5a, the iso-surface looks like a solid surface. At any point when the desired value is found the opacity is set to 1. This means that any data behind it does not get included in the calculation for the final color.

However, in Figure 4.5b the volume rendering does not create a solid surface. Some of the data behind it can be seen. For a clear example of this see Figure 4.6. While data is rendered in-front of the building, the building can still be seen through the data.

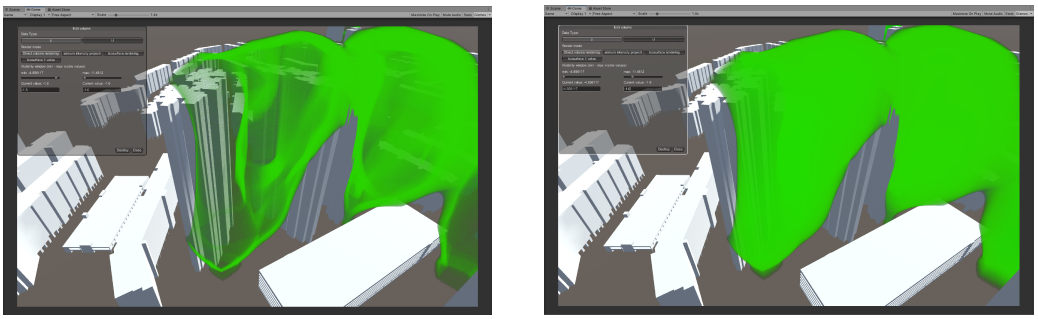
Furthermore, as the camera gets closer to the region it is representing the LoD of that region should increase. Figure 4.7 depicts a singular region as the camera approaches. It is clearly visible that as the camera approaches the LoD increases and provides a more detailed outline of the data.

There are two main issues with the visualization of the iso-surfaces and volume rendering. Firstly, at region boundaries the data can seem like it jumps. As seen in Figure 4.8, the data within one regions displays the iso-surface in one location. At the surface boundary the data



(a) (b)

Figure 4.5: This figure depicts how scalar values are presented. (a) Iso-surface (b) Volume Rendering



(a) (b)

Figure 4.6: Volume rendering with different parameters. (a) -1.8 - -1.6 (b) -4.8 - -1.6

jumps to a new location. While jumps like these can happen, the more likely situation is that there is a gradual transition from one to the other. However, as regions are each rendered separately they have no way to compunicate this and display a gradual transition.

The second issue occurs mainly in outer regions. In the outer regions most of the voxel values within a region contain the same, or very close to the same, value. This results in either the entire region being rendered or none of the region (see Figure 4.9). Consequently this provides little to no data about those regions. Of course, for sections within the data that contain mainly the same value, iso-surfaces and volume rendering will always provide little value.

4.3.2 Visualization of Barbs

The purpose of each barb is to indicate the direction and magnitude of the value it is representing. This is accomplished through the use of a mesh arrow. The size of the arrow mesh is proportionally to the magnitude of the value. Furthermore, the color of the mesh changes depending on the direction of the value, see Figure 4.10. For this thesis, the velocity (U) is the quantity of interest shown during these examples.

The number of barbs within a region is equal to the number of voxels within the current LoD

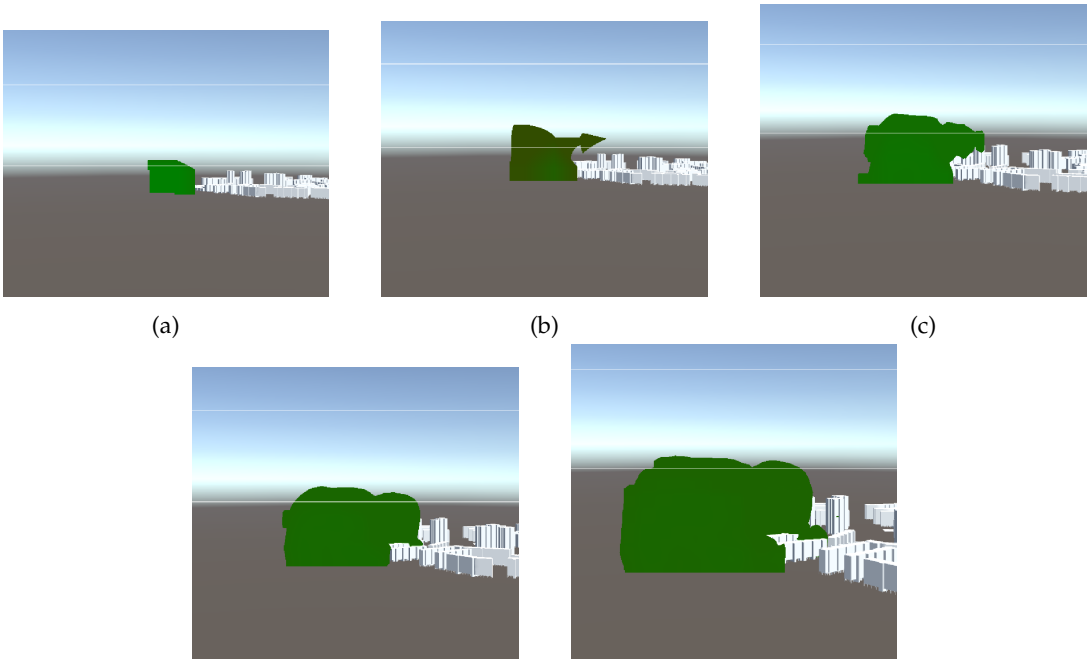


Figure 4.7: Pressure value in Region 245 between -3 and -1.5. (a) LoD 1. (b) LoD 2. (c) LoD 3. (d) LoD 4. (e) LoD 5.

being rendered. Figure 4.11 depicts how the number of barbs increase as the LoD increases for a single region. While increasing the number of barbs within a region does increase the accuracy of each, it does not provide a clear overview. Therefore, the parameters of which barbs should be visualized can be altered. Figure 4.12 depicts two scenarios. In the first scenario all barbs are shown, clearly this does not provide any detail about the environment. However, in the second scenario the parameters have been adjusted to only show values with a magnitude between 0 to 2. This provides a much better overview of the study area.

Figure 4.13 provides an overview of how the LoD effects the visualization for the entire study. As the camera gets closer to the taller buildings the number of barbs increases. As the LoD increases a better overview of the velocity of the wind within the area can be seen.

4.3.3 Frame rate

Frame rate, denoted as fps (frames per second), represents the frequency at which individual frames are presented during the visualization process. Each frame serves to display a distinct image on the user's screen. Higher frame rates typically result in smoother motion. fps serves as a quantitative metric for measuring smoothness during application runtime.

In the runtime of the application, the FPS commonly operates slightly above 30 fps. Illustrated in Figure 4.14, the fps trajectory generally maintains smoothness over its history, albeit with occasional troughs while data is loaded into memory. However, it tends to recover to

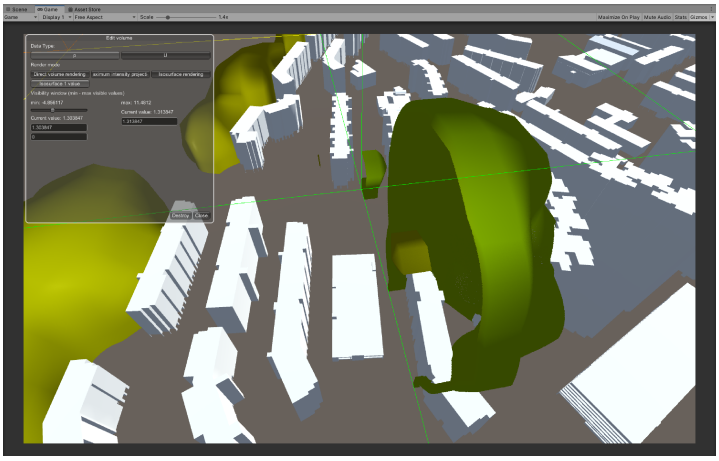
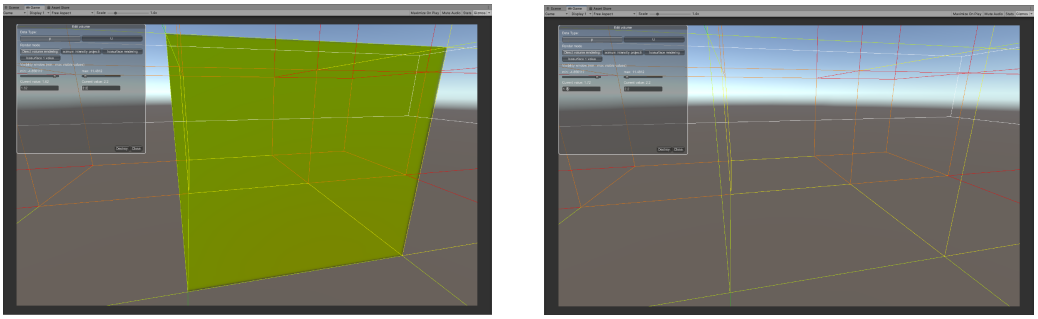


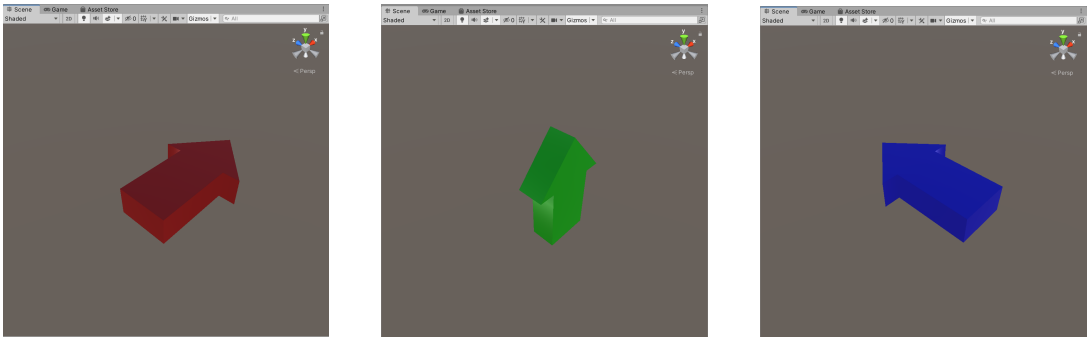
Figure 4.8: A jump within the iso-surface



(a) (b)

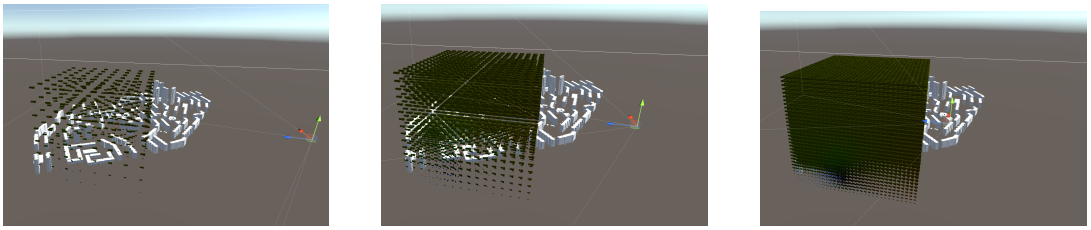
Figure 4.9: Volume rendering with different parameters. (a) -1.8 - -1.6 (b) -4.8 - -1.6

the 30 fps benchmark soon thereafter.



(a) (b) (c)

Figure 4.10: The Barb colors in their different direction (a) Barb in the X direction (b) Barb in the Y direction (c) Barb in the Z direction



(a) (b) (c)

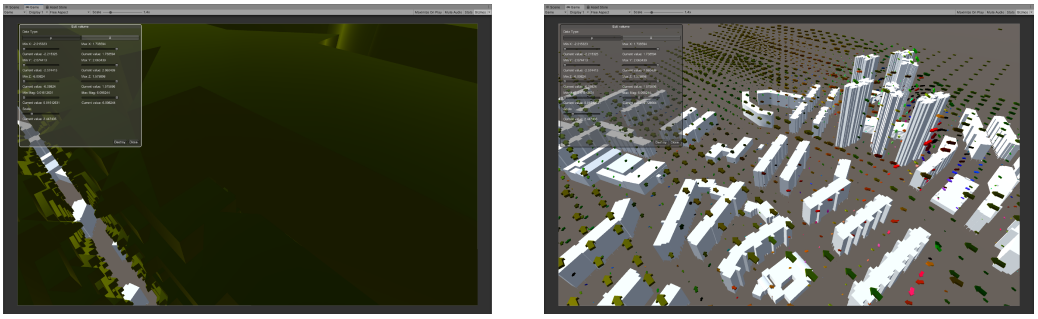
Figure 4.11: As the LoD of a region increases so do the number of barbs present in the region. This figure displays the barbs at various LoDs. (a) Barbs at LoD 3. (b) Barbs at LoD 4. (c) Barbs at LoD 5.

4.4 Numerical Errors

During the pre-proecessing phase, the transformation applied to the data lead to alterations in both the structure and values of the dataset. Consequently, there exists a likelihood of discrepancies between the new transformed values and their original counterparts. This section delves into the calculation of this error and presents an analysis of the error in two distinct regions: a dense region and a sparse region.

The dense region is situated near the center of the simulation and characterized by an abundance of original data points, while the sparse region is positioned closer to the edge of the simulation and consists of a limited number of data points. Given the computational expense associated with calculating the error, only two regions have been evaluated in this study. The selection of a dense and a sparse region aims to provide a good representation of the entire study and represent a broader trend across the simulation.

The methodology for assessing the error of the new data points went as follows (see figure 4.15). The center of the voxel of that data point was computed. The cell from the original dataset in which the center falls was determined. The corresponding value of the new dataset is then the value of the cell in which the voxel center falls. The error between these two value was computed using the following formula:



(a) (b)

Figure 4.12: Overview of the study area in two scenarios. Scenario (a) provides little to no understanding of the data. After adjusting the visualization parameters, (b) provides a much better overview of the data. (a) Barbs for all magnitude values are shown (b) Barbs for magnitude values between 0 and 2 are shown.

$$error = (newValue - oldValue)/oldValue \tag{4.1}$$

The error of each voxel value is then calculated followed by finding the average of all the errors. This average is considered the error percentage for that LoD. The results of these calculations can be found in table 4.6 for the dense region and in table 4.7 for the sparse region. In general, a steady downward trend in the average error can be seen in both the dense and sparse regions.

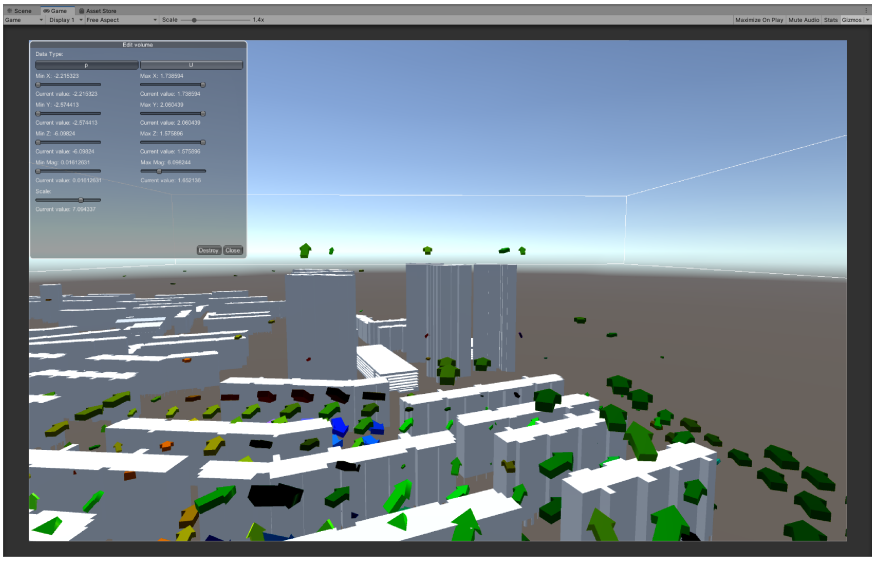
LoD	Average Pressure Error (%)	Pressure standard Deviation	Velocity Magnitude Error (%)	Velocity Magnitude standard deviation
0	23.42	0	92.61	0
1	15.11	0.13	35.11	0.14
2	14.05	0.33	24.31	0.12
3	77.76	161.91	2.96	0.11
4	1.55	4.56	1.14	0.37
5	6.50	121.93	0.42	0.03

Table 4.6: Dense region errors. (region 245)

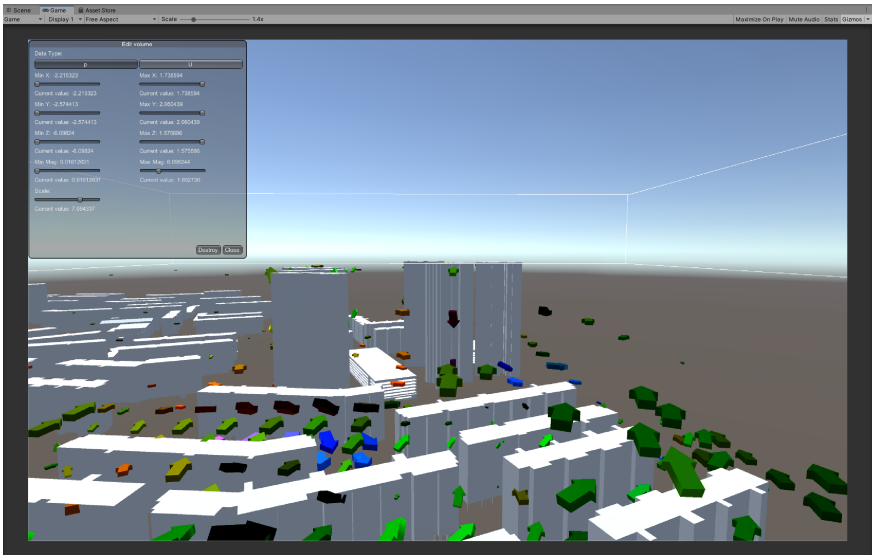
Furthermore, we can visualize where the error is greatest and smallest. As seen in both figure 4.16b and figure 4.17b a large portion of the data has no error compared to the original data. However, areas that have a larger error (figure 4.16a and 4.17a) bring up the average significantly.

LoD	Average Pressure Error (%)	Pressure standard Deviation	Velocity Magnitude Error (%)	Velocity Magnitude standard deviation
0	0.96	0	46.49	0
1	2.29	0.0009	25.58	0.06
2	2.68	0.0006	11.81	0.04
3	2.38	0.008	3.98	0.016
4	1.87	0.0092	1.70	0.0063
5	2.27	0.153	0.09	0.002

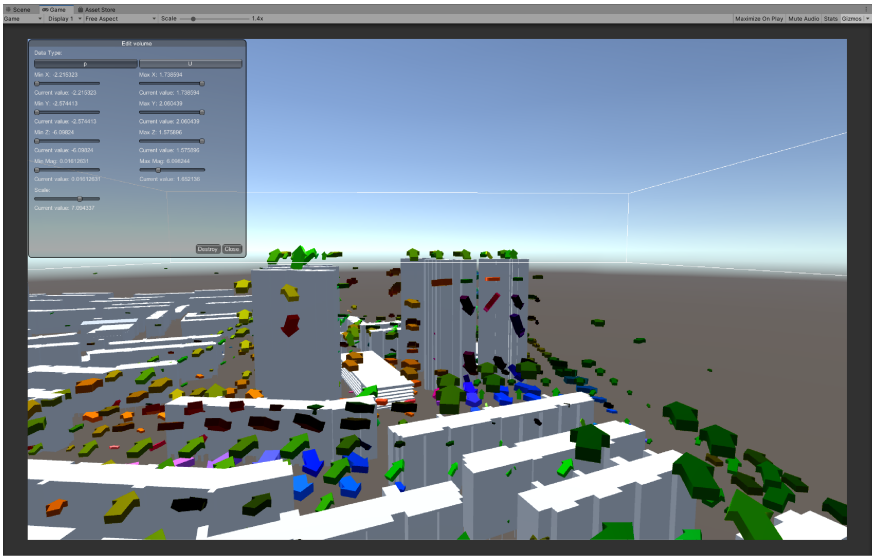
Table 4.7: Sparse region errors. (region 0)



(a)



(b)



(c)

Figure 4.13: A scenario in which the user moves closer to an area of interest. As the user moves closer the LoD of the visualization increases. (a) LoD 2 (b) LoD 3 (c) LoD 4

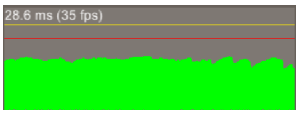


Figure 4.14: A chart depicting the fps over the history of the application. The green region represents the fps.

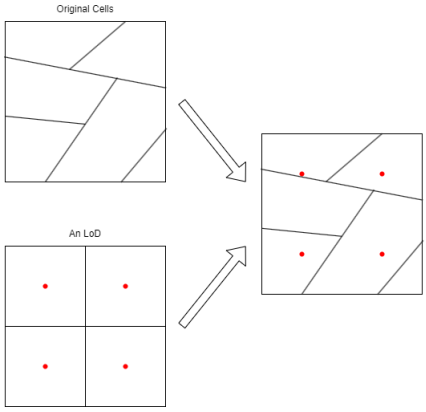
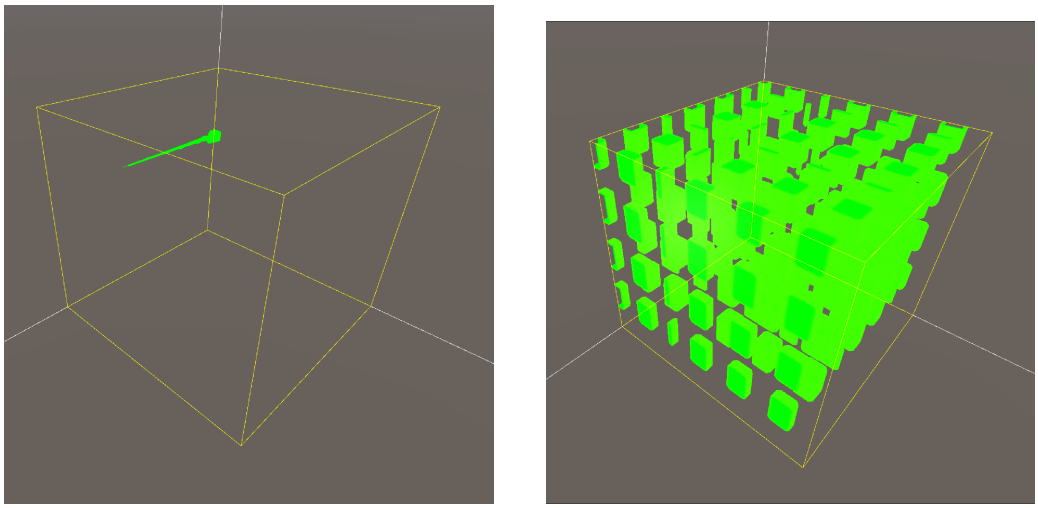
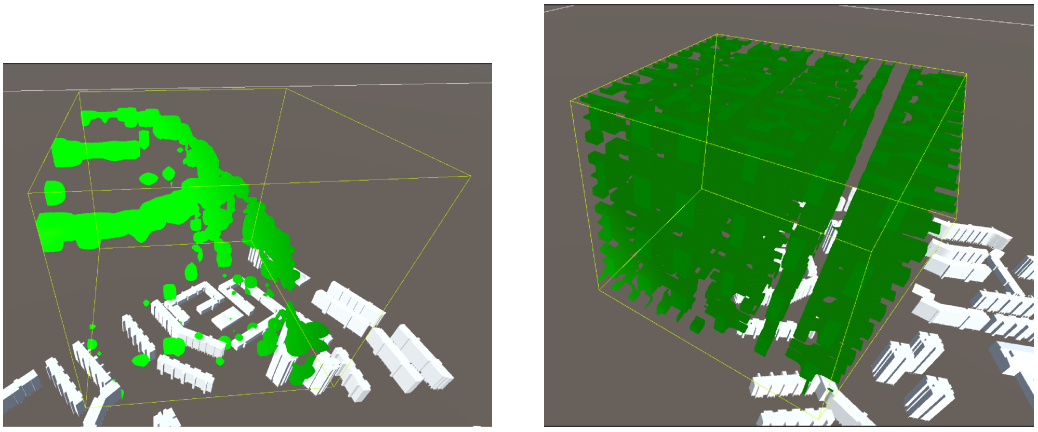


Figure 4.15: The center of the LoD voxels are compared to the cell from the original CFD results in which the center of the voxel is located.



(a) (b)

Figure 4.16: (a) All areas in the sparse region which the error is greater then 2% for LoD 5. (b) All areas in the dense region in which the error is 0%. for LoD 5.



(a) (b)

Figure 4.17: (a) All areas in the dense region which the error is greater then 2% for LoD 5. (b) All areas in the dense region in which the error is 0%. for LoD 5.

5 Discussion and Conclusion

5.1 Research overview

The aim of this thesis was to develop a method to visualize *massive* CFD simulation results, with a focus on personal computers. The amount of RAM is a limiting factor in allowing data into memory and thus visualization. Dynamic loading was proposed to solve this issue by automatically loading data in and out of memory as needed. Furthermore, a data transformation was proposed to provide a structured data set to work with for easier data mangement. The questions defined in section 1.2 are reviewed below.

Can massive Computational Fluid Dynamic results be visualized using Game Engines in real-time while presenting valuable information?

Answer: Mostly yes

This thesis has demonstrated that through the use of automatic dynamic loading the entire study area could be visualized without requiring human input. As shown in chapter 4.3, both scalar and vector values can be visualized within Unity through iso-surfaces, volume rendering and barbs. The user can then move through the environment in real-time to visualize that data from different locations and angles.

To accomplish real-time visualization the original CFD results had to go through a pre-processing phase. The methodology proposed in this thesis only looks at CFD results generated from OpenFOAM. However, much of the methodology can work for other CFD results as well. The main requirement for each quantity of interest value is that it can be associated with a singular point in the study space. Therefore, this methodology should easily expand to other CFD results.

By discretizing the study area into regions during the pre-processing phase, relevant data could be loaded/unloaded as desired. Through unloading data that was no longer relevant memory issue were avoided. Through the proposed methodology the amount of data in memory at any given point is only correlated to the number of regions loaded. Therefore, it is not correlated to the size of the study area. This allows the study area to grow in scale without having to worry about memory issues.

However, some visualization problems are still present. Mainly, for iso-surfaces and volume rendering there can be jumps in the data between region boundaries. Furthermore, the data is not as accurate as the original CFD results. Nevertheless, the visualization still provides an overview of the data which can be used for analysis.

What is the time requirement for pre-processing the CFD data?

Answer:

- With respect to the number of regions: **linear**
- With respect to the number of points within the CFD simulation: **linear**

Since the number of regions the study area is split into depends on user input during the pre-processing steps there is no definitive answer as to the time requirement. However, from the results it can clearly be seen that as the number of regions increase, the time required to pre-process all of these regions also increases with a linear correlation (see Figure 4.1a). This comes as a result from the fact that for each region an equal number of LoDs need to be constructed.

Furthermore, while the results section displayed all times in respect to the number of region it is worth discussing the effect of the total number of points within the CFD simulation. Despite the number of regions changing depending on the user input, the total number of points do not change. Unfortunately, no study areas with a different number of points was analyzed and no results can be shown. However, each step within the pre-processing algorithm looped over all points within the CFD simulation exactly once. Therefore, the time requirement increases linearly with respect to the number of points in the CFD simulation.

Overall, the total amount of time required for pre-processing was between 390 seconds (6 minutes and 30 seconds) and 972 seconds (16 minutes and 12 seconds). Fortunately, unlike the visualization, the pre-processing does not have to take place in real time.

How much reduction in memory does the modifications of the data provide?

Answer: As the number of regions increases the memory reduction decreases.

During the pre-processing steps the data was altered such that the resulting data had a structured format. Through using a structured format many of the attributes (such as points, faces and cells) originally needed to define the layout of the original data were no longer needed. This provided a data reduction from 84% for 432 regions to 57% for 3360 regions.

The original data requirement was static. However, with the methodology proposed in this thesis the data requirement does increase as the number of regions increases. Therefore, eventually the amount of data required will surpass that of the original data. This should be taken into consideration when deciding the number of regions to split the study area into. Regardless, each case studied within this thesis provided a reduction in data requirement.

What is the error introduced into the data by the modifications?

Answer: As the LoD of a region increases the error decreases.

Pre-processing the data introduces some errors within the resulting data. From the results in section 4.4 it can be seen that as the LoD of a region increases the error percentage decreases. This trend holds true for pressure and velocity in the dense region and velocity in the sparse region. The only exception is the pressure in the sparse region. However the error percentage hovers around the 2% and notably does not increase. Furthermore, in relation to the other error percentages this can be considered a low error percentage, especially for the lower LoDs.

A notable exception is the 77% error for pressure for a dense region at LoD 3. This is likely due to the centers of the voxels falling within a cell of the original data that is far from the average. This is further supported by the standard deviation of 161.91 seen for this data.

Overall the error introduced decreases as the LoD increases. For dense regions this downward trend is likely to continue. Due to the higher number of points within these regions higher LoDs will still contain points within all voxels.

5.2 Discussion

Contributions

Through out this thesis several concepts have been presented. These allowed the proposed methodology to take shape. While none of these techniques were novel ideas by themselves, they were put together to formulate a new strategy and provide the ability to visualize *massive* CFD results. The most important are discussed:

- **Structured CFD results:** Structured CFD results are not a new concept. However, they provide some distinct advantages over their unstructured counter parts. The use of a structured data provided a reduction in memory requirement. Furthermore, it provided the capabilities to load/unload the desired data with ease during the programs run-time. Unfortunately, the use of a structured dataset did reduce the accuracy of the data.
- **LoDs and voxelization:**

  This thesis utilized LoDs to manage memory requirement during run-time. Through using multiple LoDs the desired amount of data could be load/unloaded from memory. While lower LoDs provided less accurate visualizations, their lower memory requirement provided a valuable trade-off which could be used display data at far distances from the user. To generate the multiple LoDs, voxelization was utilized. For a given LoD_d in a region, 8^d voxels were created to represent the data at the desired accuracy.
- **Automatic Dynamic Loading:** Perhaps the biggest contribution of this thesis lies in its demonstration of the efficacy of automated dynamic loading in handling massive CFD results. It elucidates the feasibility of leveraging the users location within the study area to dynamically load and unload the visible data. This provides a smooth visualization experience. Notably, during visualization this methodology operates independently of study area's size. Consequently, this approach functions regardless of the scale of the study area.

Limitations

- **Voxels inside CAD geometries:** As mentioned earlier in section 3.2.3, voxels of any LoD may not contain a value within them. This can happen due to two reasons: no points inside the voxel or the voxel falls within a CAD geometry. At present, there exists no efficient means to determine in which of the two cases the voxel falls. With the current methodology, voxels for both cases are handled in the same manner. Due to the inability to determine which case applies to a voxel errors occur in the resulting data. For example, barbs are displayed inside CAD geometries and iso-surfaces go straight through CAD geometries. This creates a limitation in the ability to accurately visualize the CFD simulation data.
- **Cannot export visualized data:** To visualize the iso-surfaces and volume rendering results no actual mesh or data is created. The shader simply draws the correct color to the screen when required. Because no mesh or data is created the result of the iso-surface and the volume rendering cannot be exported to file to be used later. Techniques do exist to extract iso-surfaces from voxels, such as the marching cubes algorithm, however,

they are not implemented in this thesis.

- **None rectangular study areas:** The methodology proposed in this thesis relies on a study area that is rectangular in shape to be able to split up the study area up into cube-like regions. For study areas that are not rectangular (e.g.spheres, domes, cylinders, etc) some regions will completely fall outside of the study area. While this thesis proposes a solution to deal with voxels that do not contain any data points it does not so for entire regions that have no data points. This problem can be resolved by simply removing any regions that do not contain any data points. However, the methodology in this thesis does not explore this path and undesirable side effects may occur.

Recommendations and future work

- **Voxels inside CAD geometries:** As mentioned earlier, errors are introduces into the resulting data due to the inability to distinguish whether a voxel falls within a CAD geometry or contains no data points. Therefore, they are currently handled in the same manner. However, if a distinction could be made between the two cases a predetermined special value could be assigned to voxels t hat fall within a CAD geometry. Barbs could then be disabled if they have this value and the shader could handle the special value as desired.

  Methods do currently exist to determine if a voxel falls within a CAD geometry. However, the majority of the CAD geometries within the study area are combined into a single large CAD geometry. Due to this the number of faces within the CAD geometry is very large. Therefore, the time required to determine if a voxel falls within a CAD geometry is incredibly long. Future solutions should look into methods that speed up the process to determine if a voxel falls within a CAD geometry. A suggestion could be to split up the large CAD geometry into smaller CAD geometries. The bounding box of these smaller geometries can then be utilized to quickly determine if a voxel could possible fall within the geometry.

- **Slicing:** Slicing can be used to gain a better understanding of the internal part of a volumetric dataset. Currently the methodology provides no way to incorporate this technique into the visualization. However, it is possible to include it. The part of the data which the user wants to slice can be represented as a plane. The general equation of a plane can be defined as $ax + by + cz + d = 0$. The parameters of this equation a, b, c and d can be passed into the shader. These parameters can then be used inside the shader to determine on which side of the plane the sampling point falls. If the sampling point falls on the 'wrong' side of the plane the value can be occluded from the resulting value. The rest of the shader can work as it previously did.

- **Sub-regions:** Currently, this thesis suggests discratizing the study area into regions and generating multiple LoDs for each region. However, due to the exponential increase in the number of data points for successive LoDs, there exists a limitation on the highest attainable LoD. The pre-processing and loading during run-time for high LoDs can demand excessive time. Moreover, despite LoDs enabling observation of the entire region at various amounts of detail, the entirety of the region can only be represented as a single LoD. This can be undesirable in situations where smaller sections of the region are desired to be inspected.

Therefore, to solve this issue, the potential to split up the region into sub-regions should be considered. I propose one possible methods however other likely exist and should be explored:

One option to create sub-regions is to repeat the pre-processing step proposed in this thesis but treating a region as the study area. That is, pick a desired measurement m, split the region into sub-regions of size m and finally create LoDs for each sub-region. Each sub-region can then be displayed at a different LoD.

- **None-linear voxel averages:** For any voxel its value is calculated through taking the average of all points that fall within that voxel. Currently an equal weight is allocated to each point within the voxel. However, alternative weights could be assigned to points based on various parameters. For example, a weight could be assigned depending on the point's distance from the center. This method could lower the error within the data set as points closer to the center could more closely resemble the actual value at the voxel's center.
- **Optimal Parameters:** The processes proposed in this thesis requires multiple parameters; the region size, number of LoDs to create per region, the load, render, and destroy distance. This thesis proposes some values for each of these parameters, however, they were found through trial and error. No statistical analysis was preformed to find the optimal parameters. Although optimal parameters may vary depending on the system, due to factors such as processor specifications, available RAM, time constraints and others, it remains feasible to establish guidelines towards favorable parameter settings.
- **Pre-processing in parallel:** To continue the path of optimization, the pre-processing steps can be further optimized. All pre-processing steps lend themselves to parallel execution. An initial improvement could involve parallelizing the process across the CPU, utilizing all available cores rather than the current singular core. However, a more advanced optimization could leverage the GPU for parallelization. With the number of cores being greater than that of the CPU, employing the GPU offers further optimization.
- **2D visualization techniques:** This thesis only explored visualizing the data in 3D. However, analyzing CFD results can also entail 2D visualizations in the form of graphs. Future research could look into incorporating 2D visualizations into the game engines.

Algorithm 5.1: IndexToCoordinates (*index*, *LoD*, *region_width*, *region_height*, *region_depth*)

Input: *index*, *LoD*, *region_width*, *region_height*, *region_depth*
Output: *region_index*: The index of the region to which the *point* belongs

$voxels\_per\_dimension \leftarrow 2^{LoD}$;
$voxels\_per\_side \leftarrow voxels\_per\_dimension * voxels\_per\_dimension$;
$voxel\_width \leftarrow region\_width / voxels\_per\_dimension$;
$voxel\_height \leftarrow region\_height / voxels\_per\_dimension$;
$voxel\_depth \leftarrow region\_depth / voxels\_per\_dimension$;
$depth\_index \leftarrow floor(index / voxels\_per\_side)$;
$index \leftarrow index - (depth\_index * voxels\_per\_side)$;
$height\_index \leftarrow floor(index / voxels\_per\_dimension)$;
$index \leftarrow index - (height\_index * voxels\_per\_dimension)$;
$width\_index \leftarrow index$;
$x \leftarrow width\_index * voxel\_width$;
$y \leftarrow height\_index * voxel\_height$;
$z \leftarrow depth\_index * voxel\_depth$;
return (x, y, z);

Algorithm 5.2: RegionIndex (*point*, *min_coord*, *region_width*, *region_height*, *region_depth*, *num_width*, *num_height*)

Input: The point for which the region is being determined *point*, the smallest coordinate of the study area *min_coord*, the width of a single region *region_width*, the height of a single region *region_height*, the depth of a single region *region_depth*, the number of regions in the width direction *num_width*, the number of regions in the height direction *num_height*
Output: *region_index*: The index of the region to which the *point* belongs

$width\_index \leftarrow floor((point.x - min\_coord.x) / region\_width)$;
$height\_index \leftarrow floor((point.y - min\_coord.y) / region\_height)$;
$depth\_index \leftarrow floor((point.z - min\_coord.z) / region\_depth)$;
$region\_index \leftarrow width\_index + (height\_index * num\_width) + (depth\_index * num\_width * num\_height)$;
return *region_index;*

Bibliography

(2004). *OpenFOAM Documentation*. OpenCFD Ltd.

(2012). Google maps structure.

(2023). 2d tiles overview.

(2023). Stl to obj.

Ahrens, J., Geveci, B., and Law, C. (2005). *Visualization Handbook*, chapter ParaView: An End-User Tool for Large Data Visualization. Elsevier Inc., Burlington, MA, USA.

Anderson, J. D. and Wendt, J. (1995). *Computational fluid dynamics*, volume 206. Springer.

ANSYS, I. (2023). Ansys fluent.

Bell, D. (2018). Arrow pointer: 3d model.

Berger, M. and Cristie, V. (2015). Cfd post-processing in unity3d. *Procedia Computer Science*, 51:2913–2922. International Conference On Computational Science, ICCS 2015.

Blocken, B., Janssen, W., and van Hooff, T. (2012). Cfd simulation for pedestrian wind comfort and wind safety in urban areas: General decision framework and case study for the eindhoven university campus. *Environmental Modelling Software*, 30:15–34.

Canepa, E. (2004). An overview about the study of downwash effects on dispersion of airborne pollutants. *Environmental Modelling and Software*, 19:1077–1087.

Chen, J., Wu, B., Delap, M., Knutsson, B., Lu, H., and Amza, C. (2005). Locality aware dynamic load management for massively multiplayer games. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, page 289–300, New York, NY, USA. Association for Computing Machinery.

Chen, Q. (2009). Ventilation performance prediction for buildings: A method overview and recent applications. *Building and Environment*, 44:848–858.

Chu, A., Kwok, R., and Yu, P. (2005). Study of pollution dispersion in urban areas using computational fluid dynamics (cfd) and geographic information system (gis). *Environmental Modelling Software*, 20:273–277.

Clyne, J., Mininni, P., Norton, A., and Rast, M. (2007). Interactive desktop analysis of high resolution simulations: application to turbulent plume dynamics and current sheet formation. *New Journal of Physics*, 9(8):301.

Defraeye, T., Blocken, B., and Carmeliet, J. (2010). Cfd analysis of convective heat transfer at the surfaces of a cube immersed in a turbulent boundary layer. *International Journal of Heat and Mass Transfer*, 53:297–308.

Defraeye, T. and Carmeliet, J. (2010). A methodology to assess the influence of local wind conditions and building orientation on the convective heat transfer at building surfaces. *Environmental Modelling and Software*, 25:1813–1824.

Drebin, R. A., Carpenter, L., and Hanrahan, P. (1988). Volume rendering. *SIGGRAPH Comput. Graph.*, 22(4):65–74.

Evola, G. and Popov, V. (2006). Computational analysis of wind driven natural ventilation in buildings. *Energy and Buildings*, 38:491–501.

Friston, S., Fan, C., Doboš, J., Scully, T., and Steed, A. (2017). 3drepo4unity: Dynamic loading of version controlled 3d assets into the unity game engine. In *Proceedings of the 22nd International Conference on 3D Web Technology*, Web3D '17, New York, NY, USA. Association for Computing Machinery.

Gromke, C., Buccolieri, R., Di Sabatino, S., and Ruck, B. (2008). Dispersion study in a street canyon with tree planting by means of wind tunnel and numerical investigations – evaluation of cfd data with experimental data. *Atmospheric Environment*, 42:8640–8650.

Hanna, S., Brown, M., Camelli, F., Chan, S., Coirier, W., Hansen, O., Huber, A., Kim, S., and Reynolds, R. (2006). Detailed simulations of atmospheric flow and dispersion in urban downtown areas by computational fluid dynamics (cfd) models - an application of five cfd models to manhattan. *Bulletin of the American Meteorological Society*, 87:1713–1726.

Jiang, Y. and Chen, Q. (2002). Effect of fluctuating wind direction on cross natural ventilation in buildings from large eddy simulation. *Building and Environment*, 37:379–386.

Lavik, M. (2020). Unityvolumerendering. https://github.com/mlavik1/UnityVolumeRendering.

Milashuk, S. and Crane, W. (2011). Wind speed prediction accuracy and expected errors of rans equations in low relief inland terrain for wind resource assessment purposes. *Environmental Modelling and Software*, 26:429–433.

Mochida, A., Tominaga, Y., Yoshino, H., Okaze, T., and Shida, T. (2007). Cfd prediction of wind environment and snowdrift around a building. volume 14.

Muñoz-Esparza, D., Shin, H. H., Sauer, J. A., Steiner, M., Hawbecker, P., Boehnert, J., Pinto, J. O., Kosović, B., and Sharman, R. D. (2021). Efficient graphics processing unit modeling of street-scale weather effects in support of aerial operations in the urban environment. *AGU Advances*, 2(2):e2021AV000432. e2021AV000432 2021AV000432.

MystiveDev (2021). 3 things i wish i knew when i started level design tutorial.

Opsomer, A. (2020). Exploring the effects of void decks on urban ventilation in singapore: A computational design simulation approach for wind microclimate-informed urban planning.

Sousa, J., García-Sánchez, C., and Gorlé, C. (2018). Improving urban flow predictions through data assimilation. *Building and Environment*, 132:282–290.

Tecplot, I. (2023). tecplot-360.

TomTom (2020). Understanding map tile grids zoom levels. *TomTom Developers*.

Verma, G. and Samar (2018). *Autodesk Cfd 2018 Black Book (Colored)*. CADCAMCAE Works, Gurgaon, Haryana.

Weisstein, E. W. (2023). Geometric centroid.

Westover, L. (1990). Footprint evaluation for volume rendering. *SIGGRAPH Comput. Graph.*, 24(4):367–376.

Colophon

This document was typeset using LaTeX, using the KOMA-Script class `scrbook`. The main font is Palatino.