Aleksi Kortesalmi

Dynamic Immersive Massive-Scale CFD Post-Processing in the Unity Game Engine

Metropolia University of Applied Sciences

XR Design

Degree Program in Design

Thesis

December 2024

Abstract

Author:	Aleksi Kortesalmi
Title:	Dynamic Immersive Massive-Scale CFD Post-Processing in the Unity Game Engine
Number of Pages:	28 pages + 2 appendices
Date:	22 December 2024
Degree:	Bachelor of Culture and Arts
Degree Program:	Design
Major:	XR Design
Instructor:	Turkka Loimisto, Senior Lecturer

Currently, neither built-in nor third-party support exists for computational fluid dynamics (CFD) post-processing techniques or the handling of common visualization file formats. This thesis presents a solution for visualizing massive-scale polygonal CFD datasets in Unity. The viability of immersive massive-scale CFD post-processing in Unity was examined through a mixed-reality application developed around this solution utilizing near-state-of-the-art hardware and data from a rapid compression expansion machine simulation. The performance of the solution was measured through benchmarks and the functionality of the application was determined through hands-on testing.

Multiple implemented optimizations are explored, including some improvements on approaches from the literature. These include the parallelization of multiple steps throughout the data pipeline and the use of a low-level rendering interface of Unity.

The developed visualization solution achieves runtime data loading times ranging from 2.9 to 47.1 seconds resulting in pre-processed transient datasets ranging in size from 1.54 to 25.95 gigabytes. Basic post-processing techniques were successfully implemented in the developed application. In benchmarking, the application reached the headset’s optimal frame rate of 90 in static viewing of individual objects and averaged a frame rate of over 58 during transient playback.

Keywords: CFD post-processing, massive-scale, Unity game engine, mixed reality, visualization

Tiivistelmä

Tekijä:	Aleksi Kortesalmi
Otsikko:	Dynaaminen Massiivisen Mittakaavan Laskennallisen Virtausdynamiikan Immersiivinen Visualisointi Unity Pelimoottorissa
Sivumäärä:	28 sivua + 2 liitettä
Aika:	22.12.2024
Tutkinto:	Muotoilun kandidaatti
Tutkinto-ohjelma:	Muotoilu
Pääaine:	XR Design
Ohjaaja:	Lehtori Turkka Loimisto

Tällä hetkellä laskennallisen virtausdynamiikan (CFD) jälkikäsittelylle tai sen visualisoinnin yleisille tiedostomuodoille ei ole lainkaan pelimoottorien sisäistä eikä kolmannenkaan osapuolen tukea. Tässä opinnäytetyössä esitetään ratkaisu massiivisen mittakaavan CFD polygonimuotoisten aineistojen visualisointiin ja tämän päälle kehitetty sovellus näiden aineistojen jälkikäsittelyyn yhdistetyssä todellisuudessa (MR). Unityn mahdollisuuksia CFD-jälkikäsittelyyn selvitettiin tutkimalla ratkaisun suorituskykyä ja jälkikäsittelyä sovellukseen onnistuneesti kehitettyjä toimintoja. Suorituskyvyn ja toimintojen toimivuuden vahvistamiseksi hyödynnettiin visualisaatio dataa oikeasta nopeasta puristus-laajennuskoneen (RCEM) simulaatiosta.

Opinnäytetyön projektissa esitetään monta optimisaatiota, jotka osa ovat jatkokehitetty aiemmasta kirjallisuudesta. Optimisaatiot keskittyvät suuressa osassa datan prosessoinnin eri vaiheiden rinnakkaislaskeistamiseen ja Unityn matalan tason renderöinti rajapinnan hyödyntämiseen.

Käytetyn visualisointidatan eri kappaleiden ajoaikainen lataus tai esiprosessointi mitattiin vaihtelevaksi 2.9 ja 47.1 sekunnin välillä, kun esiprosessoidun datan suuruus vaihteli välillä 1.54 ja 25.95 gigatavua tai miljardia tavua. Keskeiset jälkikäsittelyn toiminnot toteutettiin sovellukseen onnistuneesti. Suorituskyvyn mittauksissa sovellus ylsi MR-lasien optimaaliseen 90 ruutua sekunnissa nopeuteen staattisesti visualisaatiota tarkasteltaessa ja 58 keskiarvoon ajallisen visualisointiaineiston toiston päällä ollessa.

Asiasanat: CFD-jälkikäsittely, massiivinen mittakaava, yhdistetty todellisuus, Unity-pelimoottori, visualisaatio

Contents

1 Introduction

This thesis was written during a traineeship at the Technical Research Centre of Finland (VTT) as part of work in the VTT coordinated ‘Green Engine Computational Fluid Dynamics’ (GECFD) research project. The CFD simulation case for the evaluation of the developed solution is from *Wärtsilä*, a manufacturer and service provider of equipment in the marine and energy sectors and a participant in the GECFD project (1). In this project, a proof-of-concept was developed to research the future of digital collaboration in CFD. The solution presented is the groundwork for enabling fast development of the collaboration features in the future.

The research question, “Is immersive massive-scale CFD post-processing viable in a modern game engine” was answered through the development of a solution that enables CFD visualization. This was then utilized to develop an MR application with basic post-processing features to evaluate its viability through benchmarks and observed functionality.

The phrase, “Mixed Reality” (MR) will describe the experience, as there is interaction between the physical and digital worlds. The difference between augmented reality and mixed reality is small. The distinction was made with *Interaction Design Foundation’s* definitions where they make the distinction from AR with the level of interaction present. Headset developer Varjo’s definition, “In Mixed Reality, virtual objects appear as a natural part of the real world, occluding behind real objects” is also considered.

Massive scale refers less to the “physical” scale of the simulation and more to the amount of data that needs to be processed. In practice and in the context of this thesis, this means high-resolution transient simulation results that can contain anywhere from gigabytes to tens of gigabytes of data (billions of bytes).

2 Literature review

In this chapter, the motivations and approaches chosen for the study are established through exploring recent literature on the topic.

CFD post-processing is the act of analyzing simulation results with purpose-built software to obtain insights into the behavior of fluids in different environments (2). This can involve slicing the 3D simulation domain to visualize the results on a plane cutting into the geometry, which might provide insights that were not visible with the original simulation geometry (Figure 1). However, the most basic form of post-processing can be normalizing the magnitude values of a simulation output and mapping them onto a color gradient, then displaying the resulting color on the original simulation geometry (Figure 2).

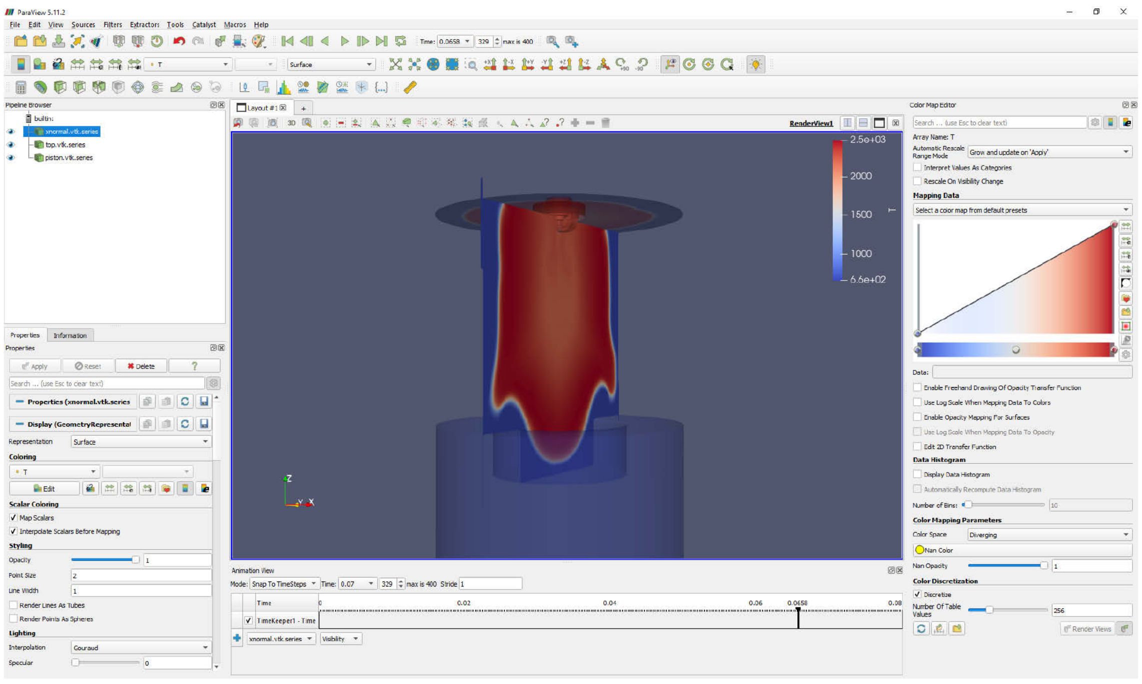


Figure 1. Temperature visualized on a slice of RCEM cylinder internal space

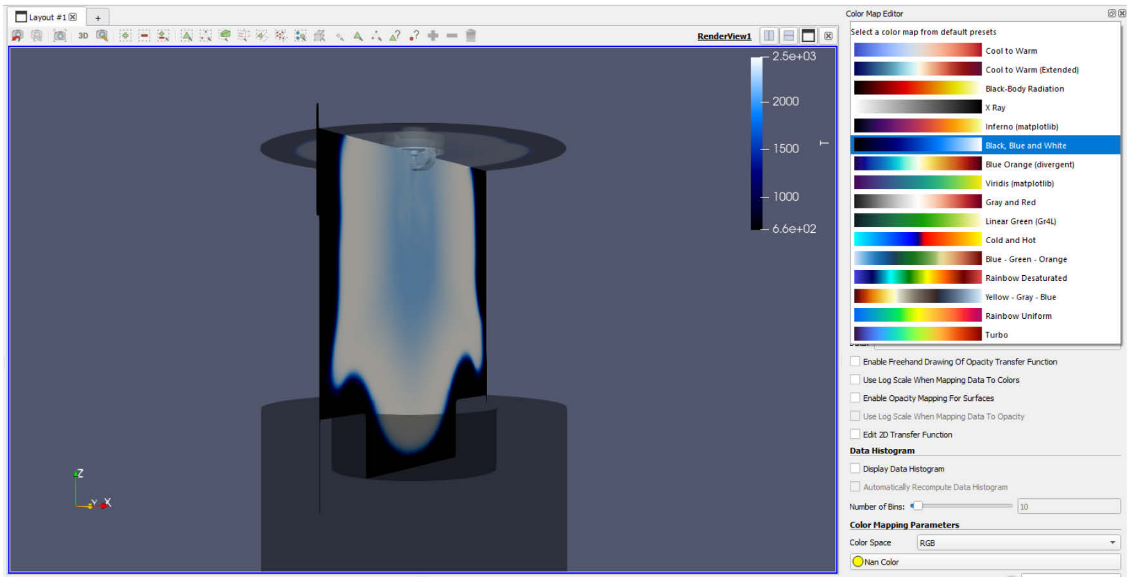


Figure 2. *ParaView* gradient selection highlighted

2.1 Background on CFD post-processing in a game engine

In the paper, *CFD post-processing in Unity3D* Berger and Christie (3) explore the use cases for game engine visualizations for its ease of use compared to more conventional post-processing software like *ParaView*. They found this approach can help non-expert stakeholders make informed decisions based on representative and qualitative visualizations.

There are many ways one can connect CFD simulation software to game engines for post-processing. The method used by Solmaz et al. (4) in their toolkit, *Acrossim* focuses on a cross-platform solution for converting raw CFD results into industry-standard formats, such as *FBX*. This approach is great for already post-processed models used during the application development, but is not as well suited, nor is it meant for runtime loading and post-processing of CFD data.

One method for implementing dynamic post-processing inside Unity, used by Wheeler et al. (5), is the plugin; VTKUnity-Activiz, also known as Activiz.NET, which exposes Visualization Toolkit (VTK) functionality for Unity, but this has many limitations (6). This plugin is a C# wrapper for VTK's C++ functions and requires its own rendering context, in OpenGL, which makes the plugin feel poorly integrated and isolated. This limits projects utilizing it to using OpenGLCore as their graphics API, which consequently limits XR projects to using the Unity legacy VR support, which further limits the Unity version to the 2019 releases (6).

Weaknesses of the game engine post-processing approach are brought up by Solmaz et al. (7) as heavy processing needs for end-user devices and the need to implement the relevant post-processing algorithms from ground up to process CFD datasets. The programming of algorithms for pre-processing CFD data to game engine friendly formats remains a challenge and currently, there are no popular solutions or libraries available for this.

This is what *Wang et al.* (8) did as they explored a highly integrated large-scale dynamic post-processing solution for Unity with promising results in the field of geotechnical engineering. Through optimizations in pre-processing and rendering they created a lightweight solution, called *GIV* (9). The implementation of this thesis utilizes many similar concepts used in *GIV*, applying them to more flexible topology, MR, and building on the rendering optimizations with Unity's low-level mesh drawing functions.

2.2 CFD simulation software

For the CFD simulation software, *OpenFOAM* was used by *Wärtsilä's* experts for exporting the visualization data utilized in this study as it is in large part used in the *GECFD* research project.

2.3 Choice of CFD post-processing software

OpenFOAM comes paired with *ParaView*, a popular open-source visualization software (10). This is how CFD results are usually analyzed (11), in specialized software requiring near-expert if not expert knowledge. *ParaView* is a powerful and versatile tool for analyzing CFD simulation results, but its XR capabilities are limited. In the production build of *ParaView*, the collaboration feature that enables others on the network to view the same visualization scene only works on Kitware's servers, the research and development company behind the software (12). It is possible to self-host an instance of the collaboration server, but this requires the *collaboration_server* executable that might need to be built from source code.

Extending existing features is possible as *ParaView* is open source but also challenging due to relatively poor documentation on the feature and having to work directly with low-level *OpenXR* or *OpenVR* standard application programming interfaces (API). This is in large contrast to game engines where software development kits (SDK) are used with a higher level of code abstraction which speeds up development.

Game engines are built specifically for the quick development of interactive 3D applications (13), but CFD data processing and analysis functionality must be created from near ground up. Although popular game engines such as Unity and Unreal Engine have invested a lot into visualization, they mostly focus on architectural visualizations and computer-aided design (CAD) instead of CFD (14, 15). This still means that advanced rendering tasks are possible, with the help of low-level cross-platform interfaces, such as Unity's *Graphics* interface (16).

Unity was chosen as the game engine for this study based on personal preference. There is also a precedent for using Unity to visualize CFD results in MR (17) and to be capable of handling large-scale simulation data processing by Wang et al. (8).

3 Methodology

This thesis utilizes case study as the primary research method to explore the question: *Is immersive massive-scale CFD post-processing viable within a modern game engine?*

A solution is developed for the basic post-processing of polygonal CFD datasets utilizing a simple and widely supported file format. Using this solution a Unity-based MR post-processing application is built to evaluate the capabilities of the solution with a real-world massive-scale CFD simulation case based on a rapid compression expansion machine (RCEM).

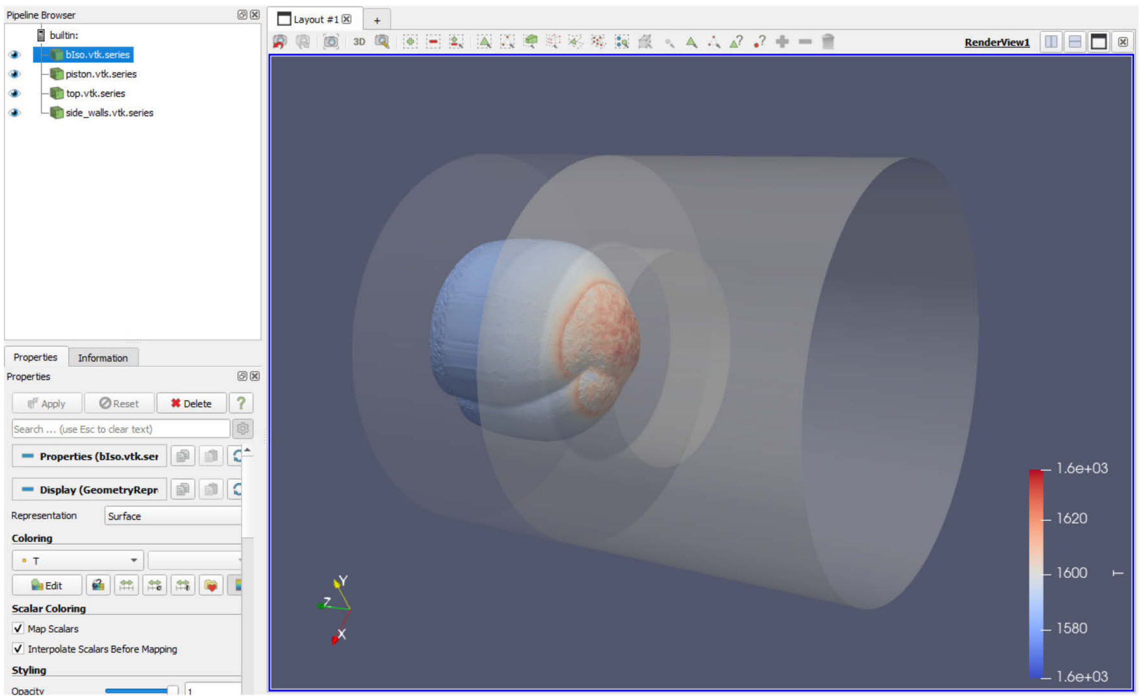
In practice, “immersive” in this question refers to visualizing in extended reality (XR) environments. This is a part of the study as conventional post-processing software is not as well equipped for developing software for immersive environments and the promising benefits, they show in the field of visualization (18). An example of this could be for instance overlaying CFD simulation results on the physical object that serves as the counterpart to its digital twin in the simulation, providing insights quickly and with great real-world context.

“Massive scale” in the context of the question defines going as close as possible to the limit of current state-of-the-art (SOTA) hardware as a goal. The immersive aspect in the case of the *Varjo XR3 Focal Edition* brings with it the cost of rendering three additional screens.

3.1 Simulation case

To evaluate the real-world capabilities of the software, simulated data from a rapid compression expansion machine (RCEM) was generated and exported for visualization in the simple binary VTK format. This is explored further in the chapters below. RCEM is a machine designed for chemical experimentation that can quickly compress a fuel/air mixture by pushing a piston into a cylinder and the resulting high temperatures and pressures initiate chemical reactions (19). The expansion in RCEM means that the machine can also pull the piston rapidly outwards to expand the gas.

Iso-surfaces were generated at each timestep for the expanding hot gas inside the cylinder for the purposes of visualization. This VTK series file is labeled “bIso” (Figure 3). Iso-surface represents a boundary in volumetric data, delineating regions with distinct value differences. In this case, the boundary is formed where hot gas meets cold.



Representation Surface

Coloring

T

Scalar Coloring

✓ Map Scalars

✓ Interpolate Scalars Before Mapping

Styling

Opacity 1

Figure 3. *bIso* expanding hot gas visualization in *ParaView*

This iso-surface was used as the minimum requirement when evaluating the performance of algorithms during the development as it is distinctly generated for visualization and as the others are derived from objects directly used in the simulation and have excessively high polygon counts. Although *bIso* has less geometry than most others, the total data consisting of the geometry for each timestep along with eleven attribute arrays is still 1.54 gigabytes (billions of bytes).

4 Production

In this chapter, software design challenges, solutions to these challenges as well as the reasoning behind the choices made during production are explored.

4.1 Pipeline overview

ParaView uses the popular open-source VTK file formats designed for visualizing scientific data to import data from CFD software. The “simple legacy” VTK format was chosen for use in the implementation of this thesis for its simplicity, ease of use, and information density in the binary variant.

Loading geometry from a VTK file sequence into memory would guarantee fast rendering but requires the whole sequence to exist in memory which in the case of the sample data can be over 64GB in the worst-case scenario. This means that systems with less than this amount of random-access memory (RAM) would not be able to achieve the benefits of this approach. Considering this, streaming the data from files was chosen, as this avoids workarounds that would be needed when exceeding the system’s memory capacity. This means that this approach will downscale easier to lower-end devices and mainly depends on the capabilities of the processor and file storage medium.

The word “streaming” in this context is used to describe the process of loading data into system memory only for the duration of its active use, which in this case is the rendering. This means that data might have to be read from a file to memory every single frame, overwriting the data of the previous frame.

It is possible to stream geometry data directly from VTK files, but pre-processing steps can be made with an intermediate format to optimize streaming performance. In this thesis there are two optimizations made during pre-processing, converting polygons into triangles and normalizing attribute values. Unity requires triangles for rendering and this step is best made beforehand and not during streaming of the geometry which would introduce needless overhead. Attributes are normalized to enable fast gradient sampling in shaders. After these steps, the geometry and attributes are written into individual files with a simple custom binary file format to be later streamed during the runtime of the visualization. These steps can be seen on the left side of Figure 4. These preprocessing steps will be further explored in the chapters below.

After pre-processing, information is needed on where geometry and attribute data for each timestep are located. For this, a single file is written containing this information for a given VTK file sequence, which is hereon referred to as the visualization object descriptor (Figure 4). This information is then passed onto an instance of the *StreamingMeshPlayer* class that handles streaming.

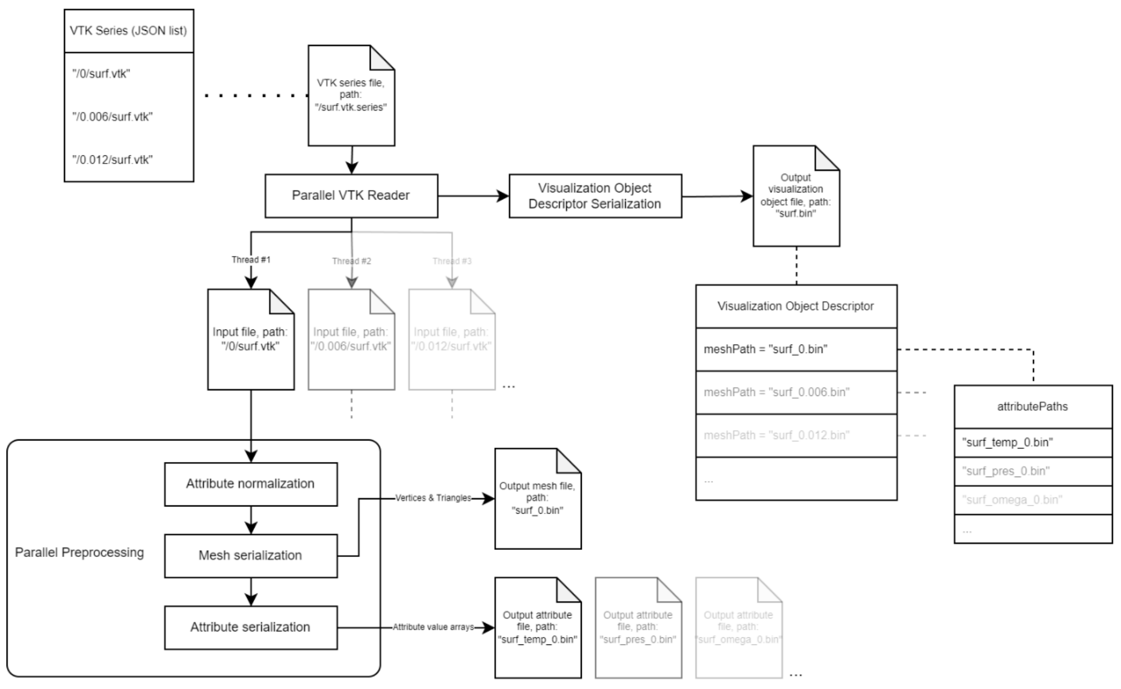

VTK Series (JSON list)
"/0/surf.vtk"
"/0.006/surf.vtk"
"/0.012/surf.vtk"
VTK series file, path: "/surf.vtk.series"
Parallel VTK Reader
Visualization Object Descriptor Serialization
Output visualization object file, path: "surf.bin"
Thread #1
Thread #2
Thread #3
Input file, path: "/0/surf.vtk"
Input file, path: "/0.006/surf.vtk"
Input file, path: "/0.012/surf.vtk"
...
Visualization Object Descriptor
meshPath = "surf_0.bin"
meshPath = "surf_0.006.bin"
meshPath = "surf_0.012.bin"
...
attributePaths
"surf_temp_0.bin"
"surf_pres_0.bin"
"surf_omega_0.bin"
...
Parallel Preprocessing
Attribute normalization
Mesh serialization
Attribute serialization
Vertices & Triangles
Output mesh file, path: "surf_0.bin"
Attribute value arrays
Output attribute file, path: "surf_temp_0.bin"
Output attribute file, path: "surf_pres_0.bin"
Output attribute file, path: "surf_omega_0.bin"
...

Figure 4. Pre-processing data flow

4.2 Pre-processing

4.2.1 VTK file reader

The file formats utilized for transferring data between *OpenFOAM* and Unity are VTK's "simple legacy" formats in binary form. These are generated using *OpenFOAM's* VTK file writer utility, *foamToVTK,* and are, by default, written to binary (20). These files must be written in binary instead of text characters encoded to the American Standard Code for Information Exchange (ASCII) as the time overhead from converting binary values to ASCII and back is too great for the benefits it gives in human readability. The time to access binary files can be 10 percent of that with ASCII as shown by Wang et al. (8, p. 22).

VTK also has formats based on extensible markup language (XML) and hierarchical data format (HDF) which are more flexible and more performant when best utilized, but with that comes complexity and little benefits in the context of this study (21). The goal is a proof-of-concept level implementation and supporting these formats might allow more performant streaming of content, but due to the development time constraints of the thesis, this is left for future research. An example of a *VTK* legacy file in the human-readable ASCII encoding can be seen in Figure 5.

```
# vtk DataFile Version 2.0
Cube example
ASCII
DATASET POLYDATA
POINTS 8 float
0.0 0.0 0.0
1.0 0.0 0.0
1.0 1.0 0.0
0.0 1.0 0.0
0.0 0.0 1.0
1.0 0.0 1.0
1.0 1.0 1.0
0.0 1.0 1.0
POLYGONS 6 30
4 0 1 2 3
4 4 5 6 7
4 0 1 5 4
4 2 3 7 6
4 0 4 7 3
4 1 2 6 5
POINT_DATA 8
FIELD FieldData 2
temperature 1 8 float
466.63 467.62 463.56 465.05 466 466.43 466.87 466.89
pressure 1 8 float
1.52 1.55 1.56 1.67 1.6 1.52 1.53 1.566
```

Figure 5. VTK legacy file format ASCII encoded example

The most essential sections in this project are the POINTS, POLYGONS, and POINT_DATA. The POINTS and POLYGONS sections form the description for the geometry. The POINT_DATA section provides the simulated physical property values for each point.

As said it is not ideal to encode the numbers in this way due to memory and computation overhead. Therefore, a copy of the Figure 5 example in this project’s supported binary format can be seen in Figure 6 as viewed in a text editor.

```
# vtk DataFile Version 2.0
Cube example
ASCII
DATASET POLYDATA
POINTS 8 float
        €?      €?  €?          €?    €?    €?    €?  €?  €?  €?    €?  €?
POLYGONS 6 30

POINT_DATA 8
FIELD FieldData 2
temperature 1 8 float
¤PéC\ÏéC®ÇçCf†èC  éC
7éC\oéCìqéC
pressure 1 8 float
\□Â?ffÆ?_®Ç?□ÂÕ?ÎÌÌ?\□Â?
×Ã?ºrÈ?
```

Figure 6. VTK legacy file format binary example

4.2.2 Attribute normalization

Scalar and vector attribute values are normalized during the pre-processing step as done in the paper by Wang et al. (8, p. 11) with some notable differences. First, mesh normalization is not performed as the transformations of the objects can be controlled through the application’s UI. Second, scalar and vector attribute values are not converted into the 4-component Color data type to reduce the memory footprint of the output and enable changing the visualization color scheme dynamically. What enables this is a rendering method which will be explored further in *Chapter 4.4 Rendering*.

With the example iso-surface data “*bIso”* with 165 meshes and a total of over 17 million vertices each having 11 attribute values. Out of these, 10 are scalar and 1 is of a 3-component vector type. This comes out to a total of over 227 million values of type float which is a 4-byte long data type, meaning a memory footprint of 911.8 megabytes (MB) as in millions of bytes. If each were to be 4 components as required by the Color data type this would be 3085.8 MB, meaning memory savings of just over 70% or 2 gigabytes (GB) or billions of bytes.

Normalization is calculated with a simple inverse linear interpolation equation and is dependent only on the minimum and maximum values from the data, which makes it an embarrassingly parallel problem. This equation finds the relative position of a value within the range between the minimum and maximum. When the value of the attribute is of vector type, magnitude is calculated and given to the equation. This equation goes as follows:

$$\frac{v - v_{min}}{v_{max} - v_{min}}$$

In this equation, *v* is the value of the attribute and v_{min} as well as v_{max} are the minimum and maximum values, respectively, in the present data. The same equation with similar, but adapted naming in high-level shading language (HLSL) goes as follows:

```
float inverseLerp(float minValue, float maxValue, float value) {
    return (value - minValue) / (maxValue - minValue);
}
```

Attributes of vector type are then first normalized to have a magnitude of 1 with the HLSL normalize function and then component-wise multiplied by the result of the inverse linear interpolation equation to achieve the desired magnitude.

4.3 Data streaming

Visualization data streaming is achieved by first deserializing or reading the pre-processed data for the current timestep into memory and updating graphics buffers that hold geometrical and attribute data which is then passed onto the *Graphics.RenderPrimitives* method (Figure 7). The *Graphics.RenderPrimitives* method and the reasons behind its use are explored further in *Chapter 4.4 Rendering*.

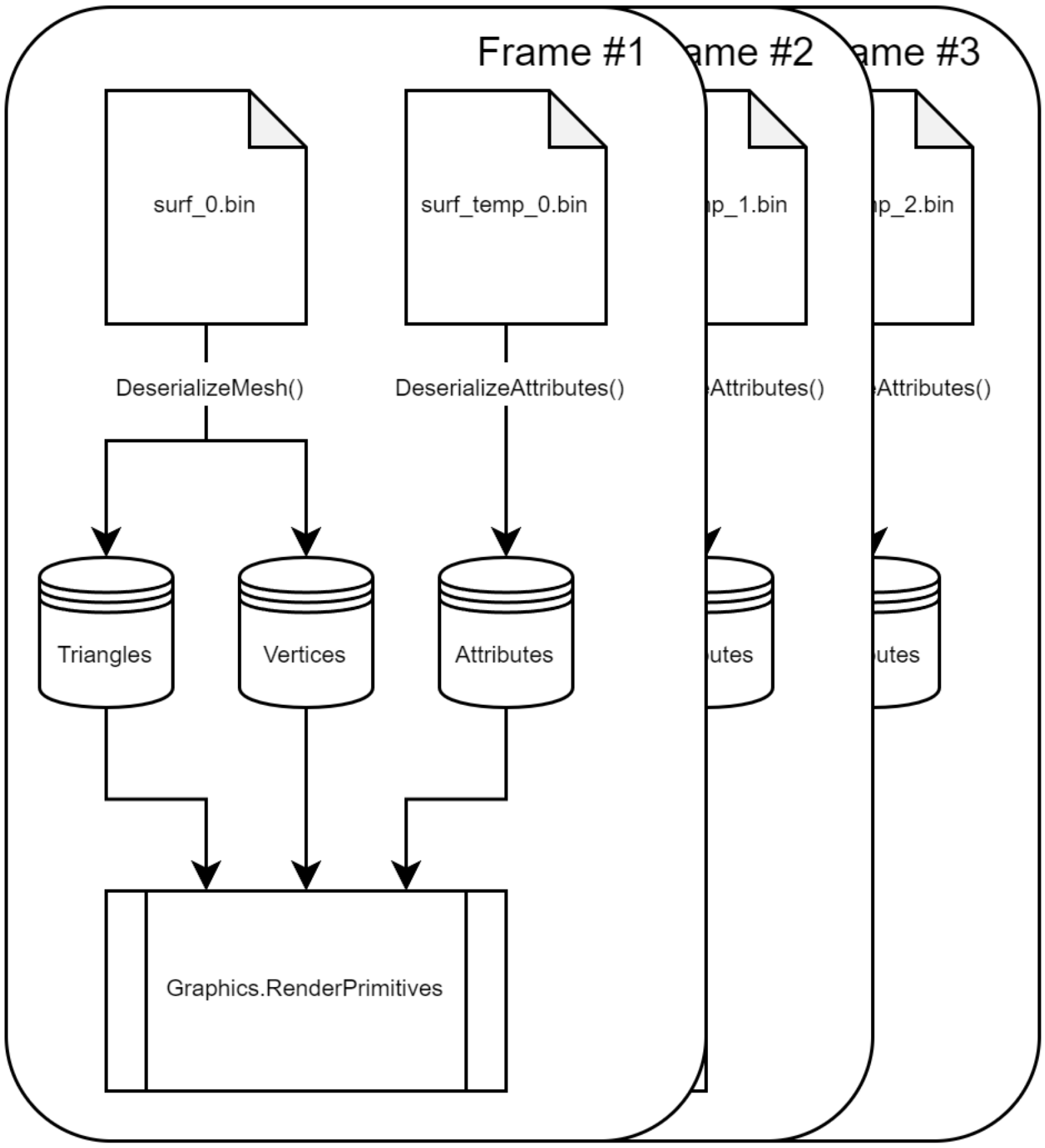


Figure 7. Flowchart visualizing the data streaming

4.3.1 Performance requirements

Streaming the geometry and attribute data coming from massive-scale CFD simulations is a major challenge. Without great optimizations reaching the reasonable frame rates required by an immersive platform like MR is impossible. Varjo strongly recommends that applications be designed to run at the headset's maximum framerate of 90 frames per second (FPS) (22). For static scenes, it is said that 60 FPS can be sufficient for shorter durations of a few minutes. In terms of computation time, 60 FPS equates to approximately 16.6 milliseconds. This time includes rendering and computation of the game logic from the rest of the application, so the time left for deserializing geometry and attributes is even less.

To achieve these numbers, not only a performant method is needed for accessing the files, such as the memory-mapped file approach used by GIV (8, p. 21)**,** but also the efficient parallelization enabled by this very approach. Memory-mapped files are a feature found in the *.NET* C# libraries bundled with Unity, which allow opening of a single file to then share it across multiple processes, and access it in these processes to either read or write data concurrently (23).

4.3.2 Parallelization

To parallelize reading geometry data, the task must be split into multiple tasks and then be executed, after which results can be collected into a single data object. In this implementation, the task of reading vertices and triangles is split individually, meaning that first vertices are read on multiple threads and then the same is repeated for triangles. Before splitting the reading into individual tasks is possible, exact byte positions and counts of the vertex and triangle sections in the file must be known. This is achieved by skimming the file to read only the metadata, after which splitting into multiple tasks is possible.

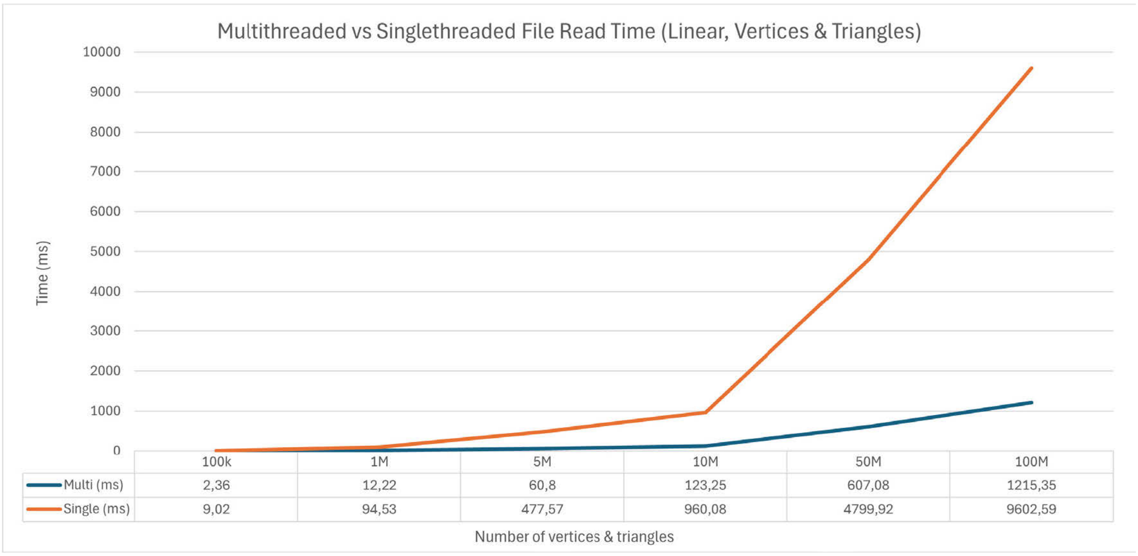
To efficiently split the file sections of vertices and triangles the number of available threads on the current system is taken and this is used to divide the number of vertices to map start and end byte positions for each thread. Tasks are launched with the .NET method *Parallel.ForEach* and a two-dimensional array is passed by reference to be filled by the running tasks. The core elements of the implementation for reading vertices in pseudo-C# can be seen in Figure 8.

```
Vector3[][] vertexResults = new Vector3[SystemInfo.processorCount][]; // Vertex results
... // Triangle results

public static void DeserializeMeshNonAlloc(string filePath, ref Vector3[] vertexCache ref int[]
triangleCache) {
  // Read only metadata from file
  MapFile(filePath, out long vertexCount, out long triangleCount)

  verticesPerProcessor = vertexCount / SystemInfo.processorCount;
  ...

  using (var mmf = MemoryMappedFile.CreateFromFile(filePath, FileMode.Open)) {
    // Read vertices in batches into subarrays
    Parallel.ForEach(vertexResults, (results, index) => {
      // Calculate start and stop index for this thread
      startIndex = index * verticesPerProcessor;
      stopIndex = startIndex + verticesPerProcessor;

      count = stopIndex - startIndex;

      using (MemoryMappedViewAccessor accessor = mmf.CreateViewAccessor()) {
        for (int i = 0; i < count; i++)
        {
          results[i].x = accessor.ReadSingle(position);
          results[i].y = accessor.ReadSingle(position + sizeof(float));
          results[i].z = accessor.ReadSingle(position + 2 * sizeof(float));
          position += 3 * sizeof(float);
        }
      }
    });

    // Read triangles in batches into subarrays
    ...
  }

  // Copy the resulting vertex subarrays into the vertex cache
  for (int i = 0; i < SystemInfo.processorCount; i++)
    Array.Copy(vertexResults[i], 0, vertexCache, i * verticesPerProcessor, verticesPerProcessor);

  // Copy the resulting triangle subarrays into the triangle cache
  ...
}
```

Figure 8. Pseudo-C# for parallelized reading of vertices and triangles from a single file

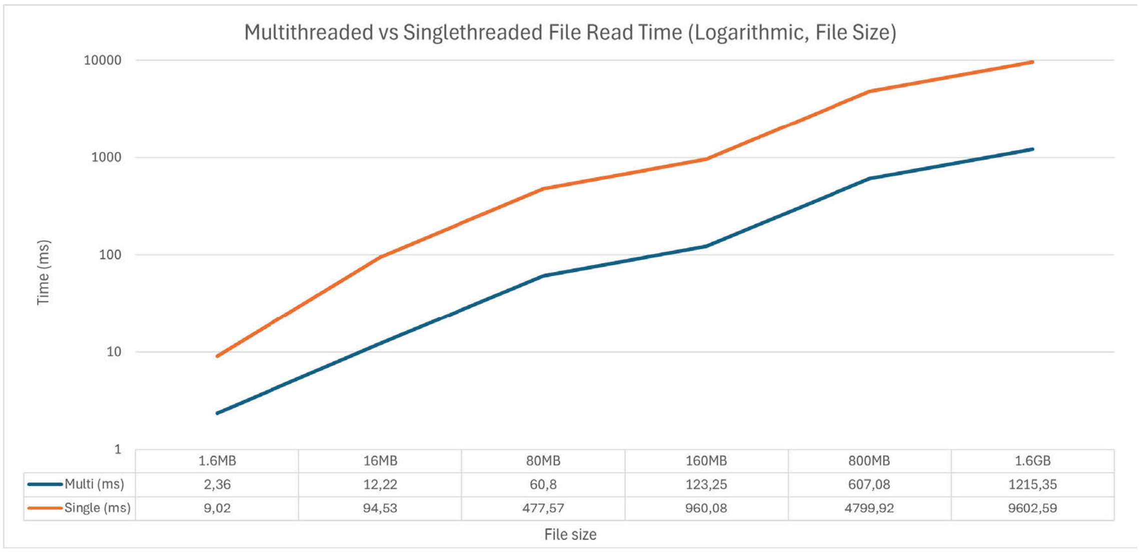
A benchmark shows an average time save of 87.3% over a singlethreaded equivalent with six files consisting of widely ranging vertex and triangle counts.

The processor utilized for this benchmark is the *AMD Threadripper PRO 7965WX* with 24 cores and 48 threads. The sample files for the benchmark were generated with randomized *Vector3* and integer data. The files contain the reported number of vertices and triangles each. Samples were collected in a built application with Unity’s logging features disabled to avoid editor-specific resources and processes from interfering. The reported times are the result of averaging over 100 samples per file. In Figure 9 the average times are plotted onto a linear scale graph with the number of geometrical elements on the X axis and in Figure 10 the same times are plotted onto a logarithmic scale graph with file size on the X axis. Metric binary prefixes apply for reported file sizes.



	100k	1M	5M	10M	50M	100M
Multi (ms)	2,36	12,22	60,8	123,25	607,08	1215,35
Single (ms)	9,02	94,53	477,57	960,08	4799,92	9602,59

Figure 9. Benchmark results in linear scale graph and geometrical element count



	1.6MB	16MB	80MB	160MB	800MB	1.6GB
Multi (ms)	2,36	12,22	60,8	123,25	607,08	1215,35
Single (ms)	9,02	94,53	477,57	960,08	4799,92	9602,59

Figure 10. Benchmark results in logarithmic scale graph and file size

4.4 Rendering

4.4.1 Environment

In Unity, there are many rendering pipelines to choose from, the older built-in render pipeline and the ones deriving from scriptable render pipeline (SRP). SRP allows developers to create their render pipelines but is also used internally at Unity to create modern default pipelines, which are the universal render pipeline (URP) and high-definition render pipeline (HDRP). The pipeline chosen for this project is HDRP, as this is the render pipeline that at the writing of this thesis supports Varjo's MR features such as video passthrough.

One downside of HDRP is that URP is the more performant one out-of-the-box, so for a more standard 2D application, it would be the clear choice. Fortunately, this does not affect the relevancy of the results or CFD data pipeline implementation, as scripts and shaders can be reproduced in URP with little to no effort, for they both derive from SRP.

4.4.2 Implementation

The biggest challenges in the rendering of the CFD visualization data are the size and frame-by-frame changing topology. This means that the data cannot be converted into a vertex animation due to the dynamic topology. However, in Unity, there are a couple of ways to approach this. The first one involves placing all the meshes in the scene, then enabling and disabling them one by one or changing the mesh on a *GameObject* as done by Wang et al. (8, p. 25) This is not ideal as this creates lots of needless memory allocations from destroying *GameObjects* which in Unity shows up as lag spikes when the C# garbage collector collects the now unused memory. Then there is the Graphics interface for Unity's optimized drawing functions, which was chosen due to flexibility and performance. The function *Graphics.RenderPrimitives* fits the needs perfectly as it can simply be given geometry, a material, and a transformation matrix, which can all be changed in each frame without allocating unwanted memory. Similar functionality can be found in *Graphics.RenderMesh*, which would enable the use of Unity's Mesh data type, but there is no way to avoid unnecessary memory allocation when changing the topology of the mesh. The chosen function *RenderPrimitives* is essential to the rendering performance. Therefore, its implementation will be explored further in the following paragraph.

Instead of needing to modify a Unity *Mesh* object, for *RenderPrimitives,* references to buffers of *GraphicsBuffer* type can be passed, which is similar to an array in C# and used to transfer data between Unity's scripting backend and the rendering pipeline (24). This implementation uses three buffers: one for triangles, one for verticeswhich only happens until the frame with the most items. After this, only values that are needed will be changed in the buffer which helps avoids garbage collection in the long run. Through *RenderPrimitives,* it is in part required that information is passed via rendering parameters. The required pieces of information are the material to be used and the bounds of the object for sorting and culling the rendered geometry (25). We can also pass a material property block for additional instance-specific information, and this is how the buffers are passed along with a gradient texture and a transformation matrix. Using the material property block makes the resulting draw call incompatible with Unity’s scriptable render pipeline (SRP) batcher which might result in an impact on performance, but this is not a concern due to the small number of instances common in CFD.

Attributes in traditional post-processing software are visualized using gradients and Unity has built-in support for their easy creation and use. As mentioned in the above chapters, this implementation performs the act of mapping the normalized attribute values to colors dynamically, and more specifically in the shader stage. The challenge is that Unity’s gradient data type does not have direct support to be passed to the vertex shader stage. Therefore, the easily configurable gradient data must be converted to a texture. This is accomplished with minimal visual and performance hit by creating a texture with a height of 1 and a width of 32. The gradient is then sampled for each pixel with the X coordinate of the pixel. This texture is then passed onto the material property block with a *SetTexture* call.

In the shader, this texture is then sampled with the value of the attribute for that specific vertex using a Sample Texture 2D LOD node. This is required as the regular Sample Texture 2D node is unavailable in the vertex shader stage, as the level of detail (LOD) information is unavailable and in the above-mentioned node this information must be provided manually. (26)

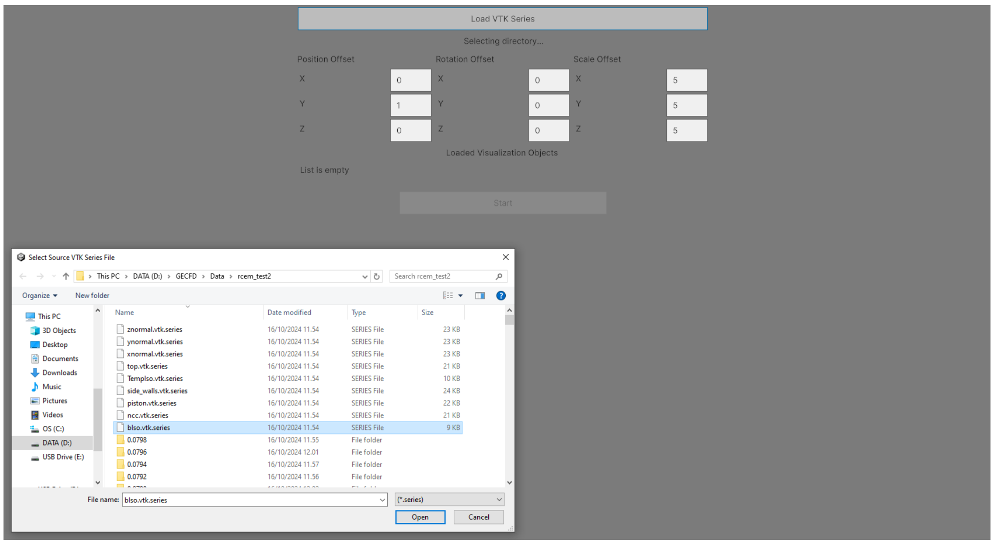
4.5 Application

The application developed around the visualization-enabling solution was kept minimal to save time for the development of the solution. Notable features of the application are the placing of *StreamingMeshPlayers* onto *Varjo* MR markers and the user interface used to expose the underlying functionality of the solution.

4.5.1 User interface

The user interface was kept as simple and reproducible as possible using Unity’s XR Interaction Toolkit package for handling interactions between the controllers and world-space canvases holding basic Unity UI components. These components include a slider for timeline control, a button for triggering playback, and dropdown elements for various visualization settings. Hand controller 3D models are from Varjo’s Unity plugin.

Upon launching the application, the user is greeted with the dynamic loading menu. Here the user can browse for a *VTK* series file or folder containing a file structure in the format of a *VTK* file sequence. Loaded sequences are then converted to the streamable intermediate format and will be referred to as visualization objects from here on out. These can then be seen in a list and a single global transformation offset can be set for them that applies when the visualization is launched. This offset is relative to the MR marker position. When the user is ready, the visualization can be launched by pressing the Start button.



Selecting directory...

	Position Offset	Rotation Offset	Scale Offset
X	0	0	5
Y	1	0	5
Z	0	0	5

Loaded Visualization Objects

List is empty

This PC › DATA (D:) › GECFD › Data › rcem_test2

Search rcem_test2

This PC
3D Objects
Desktop
Documents
Downloads
Music
Pictures
Videos
OS (C:)
DATA (D:)
USB Drive (E:)

Name	Date modified	Type	Size
znormal.vtk.series	16/10/2024 11.54	SERIES File	23 KB
ynormal.vtk.series	16/10/2024 11.54	SERIES File	23 KB
xnormal.vtk.series	16/10/2024 11.54	SERIES File	23 KB
top.vtk.series	16/10/2024 11.54	SERIES File	21 KB
TempIso.vtk.series	16/10/2024 11.54	SERIES File	10 KB
side_walls.vtk.series	16/10/2024 11.54	SERIES File	24 KB
piston.vtk.series	16/10/2024 11.54	SERIES File	22 KB
ncc.vtk.series	16/10/2024 11.54	SERIES File	21 KB
bIso.vtk.series	16/10/2024 11.54	SERIES File	9 KB
0.0798	16/10/2024 11.55	File folder	
0.0796	16/10/2024 12.01	File folder	
0.0794	16/10/2024 11.57	File folder	
0.0792	16/10/2024 11.56	File folder	

File name: bIso.vtk.series

(*.series)

Figure 11. Application load menu

Automatic looping playback starts for the visualization objects, which iterates over each simulation timestep over time. This playback speed is controllable, and the user can also scrub through the timeline of timesteps by dragging the XR controller’s pointer over the timeline viewport. The user can now change the gradient used for each visualization object and select which physical property is used to calculate the surface color.

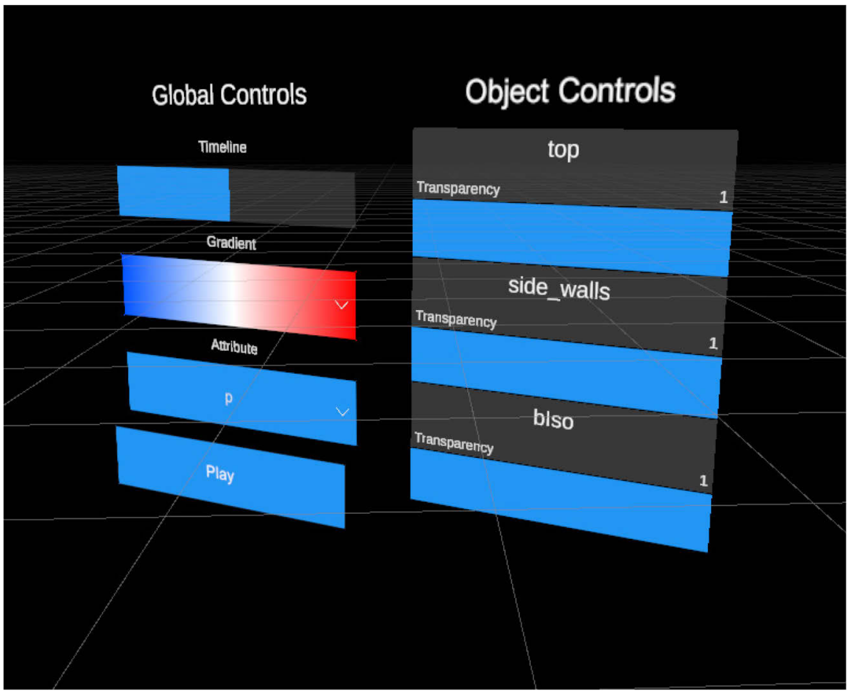


Figure 12. Application MR visualization menu

4.6 Evaluation hardware

As performance is a great factor in this study, components with the greatest relevance to the benchmarks will be listed. Most important in this case is the central processing unit (CPU) of the computer, with logical processor count having a significant impact on performance, due to the file reading algorithm's reliance on taking use of as many logical processors as possible. The computer system used for the evaluation has a 24-core, 48-thread *AMD Ryzen Threadripper PRO 7965WX* for the CPU and *NVIDIA GeForce RTX 4090* for the graphics processing unit (GPU) which in part enables rendering the vast amount of geometry and attribute data common in CFD visualization to the *Varjo XR3 Focal Edition* headset. *Valve Index* controllers were used for hand interactions. Four *SteamVR* base stations were used for tracking, located in four corners of the room.

5 Findings

All the objects of the real-world RCEM simulation case are successfully pre-processed and rendered. The use and functionality of the application are demonstrated in Appendix 2. A benchmark on the streaming of geometrical data from files indicates a speed of ~1.2 gigabytes per second or a combined count of vertices and triangles of ~150 million per second. This means ~1.26 million vertices and triangles combined can be streamed in 8.3 repeating milliseconds, which is half of the time required to fit the minimum recommended framerate of 60 frames per second. The optimization enabling this speed is not implemented for attribute streaming due to the time constraints of this thesis but is entirely possible, if not easier.

Attributes are successfully rendered with performant and memory-efficient gradient sampling with parallelized normalization as a pre-processing step. Overall dynamic runtime pre-processing of VTK file sequences was achieved in a reasonable amount of time via parallelization of aforementioned attribute normalization and VTK file parsing as seen in Figure 13.

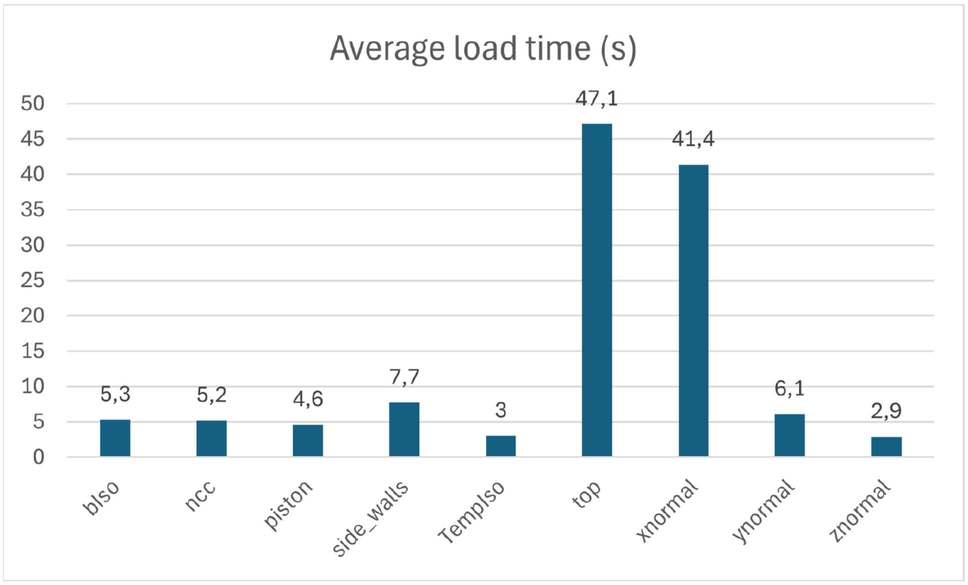


Figure 13. Averaged loading and pre-processing times of visualization sequences in seconds

Essential mixed reality features, such as passthrough and marker tracking were successfully implemented despite the use of a lower-level graphics interface and custom shader implementation required for efficient rendering of the streamed data. The application reaches Varjo's recommendation for minimum framerate in the playback of the *bIso* iso-surface along with the similarly sized sequences. Frame times were measured and averaged over 2000 samples. The frame times can be seen in Figure 14. However, it must be noted that *bIso* and *TempIso* sequences do not cover the full width of the timeline, which affects the averaged frame times. Therefore, values closer to those of others of similar size should be presumed, which can be attained from Figure 15.

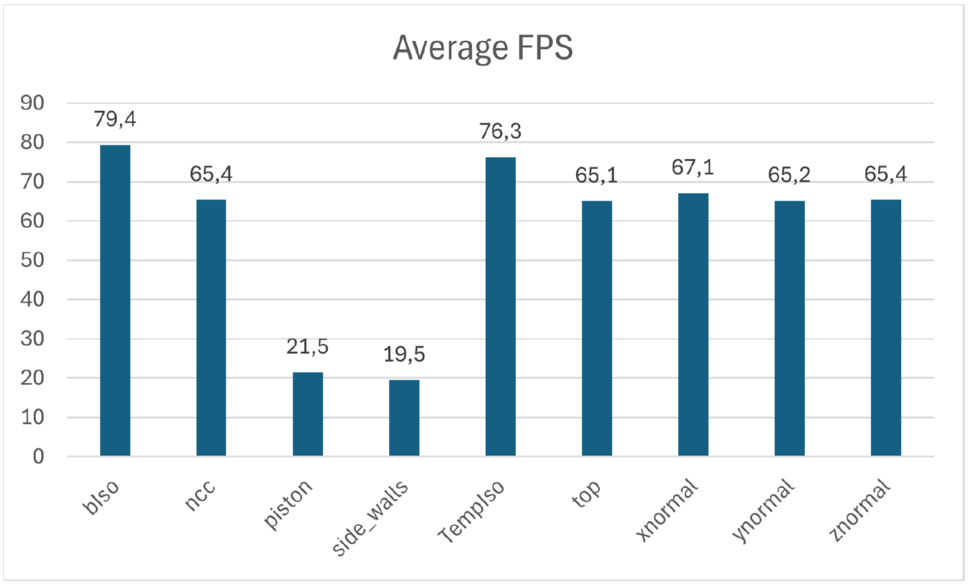


Figure 14. Averaged FPS reached during the playback of visualization sequences

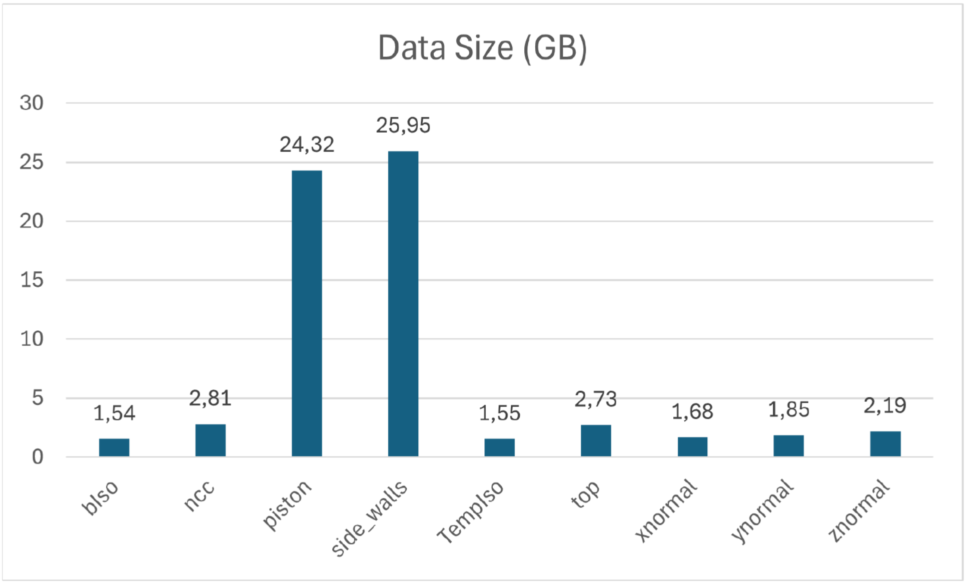


Figure 15. Data size of pre-processed data in gigabytes

Due to time constraints and the RCEM simulation data consisting only of the polygon dataset format in the “VTK simple legacy” file format, this is the only supported one out of the five described in the documentation. This bias towards the polygonal dataset format used for the development of the solution and its evaluation means that it is highly likely that the outcomes and optimizations in this thesis come from a narrow representation of the possible range of CFD simulation results.

6 Conclusion

The Unity game engine shows promise for fast development of immersive CFD post-processing applications but is still lacking the vast number of post-processing features found in conventional post-processing applications. Building similar capabilities through an open-source repository would require a massive community effort from a group of developers specialized in CFD.

This study successfully demonstrates that Unity exposes sufficient low-level access to rendering features for handling massive-scale polygonal CFD visualization datasets with a high-resolution XR headset. The game engine also exposes CPU parallelization and file IO methods through .NET to optimize the time efficiency of data streaming and data conversions between CFD visualization formats and rendering-ready geometry. Geometrical data streaming speeds of over a gigabyte per second were measured with a randomized sample dataset, meaning transient multi-gigabyte dataset playback is possible. However, to determine this more precisely further testing is needed with more diverse datasets.

Future research could investigate ways of time-efficiently downsampling the data during pre-processing to enable visualizing datasets of almost any size. Even closer to the focus of the current thesis, methods could be explored for directly streaming geometry and attribute data from the VTK file sequences. In the bigger picture of CFD post-processing in Unity, further research could focus on determining how CFD visualization data coming in various formats could be parsed into a more universal format which could then be efficiently rendered and post-processed.

References

1 Wärtsilä. Green engine simulation tool developed by VTT speeds up commercialization of new technology. [Online]. 2023 [cited 2024 Dec 8]. Available from: https://www.zemecosystem.com/green-engine-simulation- tool-developed-by-vtt-speeds-up-commercialization-of-new-technology/.

2 Resolved Analytics. An Introduction to CFD Post-Processing. [Online]. 2020 [cited 2024 Dec 7]. Available from: https://www.resolvedanalytics.com/cfd/intro-to-cfd-post-processing.

3 Berger M, Cristie V. CFD Post-processing in Unity3D. Procedia Computer Science. 2015 May 1; 51: p. 2913-2922.

4 Solmaz S, Van Gerven T. Acrossim: A toolkit for cross-platform integration of CFD simulation data in computer graphics. SoftwareX. 2023 December; 24(101585).

5 Wheeler G, et. al. Virtual interaction and visualisation of 3D medical imaging data with VTK and Unity. Healthcare Technology Letters. 2018 Aug.

6 Unity Technologies. Unity Asset Store. [Online]. 2024 [cited 2024 Dec 5]. Available from: https://assetstore.unity.com/packages/essentials/tutorial-projects/vtkunity-activiz-163686.

7 Solmaz S, Van Gerven T. Interactive CFD simulations with virtual reality to support learning in mixing. Computers & Chemical Engineering. 2022 January; 156(107570).

8 Wang R, et. al. Portable interactive visualization of large-scale simulations in geotechnical engineering using Unity3D. Advances in Engineering Software. 2020 June; 148.

9 Github Inc. Github. [Online]. 2024 [cited 2024 Dec 5]. Available from: https://github.com/Ron-Wang/GIV.

10 OpenCFD Ltd. OpenFOAM - 7.1 paraFoam. [Online]. 2024 [cited 2024 Dec 5]. Available from: https://www.openfoam.com/documentation/user-guide/7-post- processing/7.1-parafoam

11 Kodman J. B. et. al. A Comprehensive Survey of Open-Source Tools for Computational Fluid Dynamics Analyses. Journal of Advanced Research in Fluid Mechanics and Thermal Sciences. 2024 July; 119(123-148).

12 Jakub S. ParaView Discourse. [Online]. 2021 [cited 2024 Dec 5]. Available from: https://discourse.paraview.org/t/openvr-plugin-in-paraview- 5-8-0-pretty-much-just-works/3677/13.

13 Unity Technologies. Unity Solutions - XR. [Online]. 2024 [cited 2024 Dec 5]. Available from: https://unity.com/solutions/xr.

14 Epic Games, Inc. Unreal Engine Communities Visualization. [Online]. 2024 [cited 2024 Dec 5]. Available from: https://dev.epicgames.com/community/unreal-engine/getting-started/visualization.

15 Unity Technologies. Unity Topics - Visualization. [Online]. 2024 [cited 2024 Dec 5]. Available from: https://unity.com/topics/3d- visualization-explained.

16 Unity Technologies. Unity Documentation - Graphics. [Online]. 2024 [cited 2024 Dec 5]. Available from: https://docs.unity3d.com/ScriptReference/Graphics.html.

17 Unity Technologies, Davis N,. Unity Blog - Toyota makes mixed reality magic with Unity and Microsoft HoloLens 2. [Online]. 2020 [cited 2024 Dec 5]. Available from: https://unity.com/blog/industry/toyota-makes-mixed-reality-magic-with-unity-and-microsoft-hololens-2.

18 Zhang Y, et al. A survey of immersive visualization: Focus on perception and interaction. Visual Informatics. 2023 September; 7(4).

19 Werler M, et al. A Rapid Compression Expansion Machine (RCEM) for studying chemical kinetics: Experimental principle and first applications. [Online]. 2016 [cited 2024 Dec 5]. Available from: https://doi.org/10.48550/arXiv.1606.06095.

20 OpenCFD Ltd. OpenFOAM Documentation - foamToVTK. [Online]. 2022 [cited 2024 Dec 5]. Available from: https://www.openfoam.com/documentation/guides/latest/man/foamToVTK.html.

21 VTK Developers. VTK Docs - VTK File Formats. [Online]. 2024 [cited 2024 Dec 5]. Available from: https://docs.vtk.org/en/latest/design_documents/VTKFileFormats.html.

22 Varjo. Varjo Developer Docs - Performance. [Online]. 2024 [cited 2024 Dec 5]. Available from: https://developer.varjo.com/docs/get-started/Performance.

23 Microsoft. Learn.NET - Memory-mapped files. [Online]. 2022 [cited 2024 Dec 5]. Available from: https://learn.microsoft.com/en -us/dotnet/standard/io/memory-mapped-files.

24 Unity Technologies. Unity Documentation - GraphicsBuffer. [Online]. 2024 [cited 2024 Dec 5]. Available from: https://docs.unity3d.com/ScriptReference/GraphicsBuffer.html.

25 Unity Technologies. Unity Documentation - RenderParams. [Online]. 2024 [cited 2024 Dec 5]. Available from: https://docs.unity3d.com/ScriptReference/RenderParams.html.

26 Unity Technologies. Unity Docs - Sample Texture 2D LOD Node. [Online]. 2024 [cited 2024 Dec 5]. Available from: https://docs.unity3d.com/Packages/com.unity.shadergraph@6.9/manual/Sample-Texture-2D-LOD-Node.html.

27 OpenCFD Ltd. OpenFOAM. [Online]. 2024 [cited 2024 Dec 5]. Available from: https://www.openfoam.com/.

28 Epic Games, Inc. Unreal Engine Documentation - ParallelFor. [Online]. 2024 [cited 2024 Dec 5]. Available from: https://dev.epicgames.com/documentation/en-us/unreal-engine/API/Runtime/Core/Async/ParallelForImpl__ParallelForInte-.

Appendices

Appendix 1. List of abbreviations

CFD	Computational Fluid Dynamics
VTK	Visualization Toolkit
GECFD	Green engine computational fluid dynamics
VTT	Technical Research Centre of Finland
MR	Mixed Reality
AR	Augmented Reality
SOTA	State-of-the-art
CPU	Central Processing Unit
GPU	Graphics Processing Unit

Appendix 2. Demonstrative video of the application

Recorded on 09.12.2024. Duration of 1 minute and 57 seconds.