Institute for Visualization and Interactive Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Compression of Multiphase Volume-of-Fluid Datasets

Fabian Baer

Course of Study:	Informatik
Examiner:	Dr. Guido Reina
Supervisor:	Moritz Heinemann, M.Sc.
Commenced:	November 24, 2023
Completed:	May 24, 2024

Abstract

To visualize Volume of Fluid (VOF) datasets, it is necessary to handle the large VOF dataset files, which is slow. This work presents a method for compressing multiphase VOF datasets using a hierarchical tree data structure and a Level of Detail (LOD) approach for visualizing the VOF data. The data structure forms a sparse representation of the VOF field. This way, it makes use of the spacial coherency of VOF datasets to achieve high compression rates. A custom file format for storing the hierarchical data structure is presented. The file format is designed in a way that allows reading the file only partially to obtain a lower resolution version of the data. The LOD approach is used to quickly render an image of the data stored in this format for visualization purposes. Piecewise Linear Interface Calculation (PLIC) is used to extract the fluid surfaces from the VOF data. The system uses parallel processing to reduce computation times. The hierarchical data structure of the compression method is used to generate the various levels of detail for the LOD approach. In addition to the implementation of the proposed methods, a large set of benchmarks are used to test both the compression algorithm and the visualization system. The compression algorithm is compared against the performance of the more common generic compression methods LZ4, zlib and the zip file format. Both the compression time and compression rate were assessed. Especially in cases where the dataset exhibits strong spatial coherency the proposed method can achieve better compression rates than the other compression methods. For the visualization system, the rendering time was measured. The calculation of the PLIC surfaces in the visualization system is tested on both the CPU and the GPU. Both implementations can achieve fast rendering times.

Contents

1 Introduction **15**

2 Fundamentals **17**
2.1 Volume of Fluid Datasets 17
2.2 PLIC Surfaces 17

3 Related Work **21**
3.1 Compression 21
3.2 Fluid Visualization 22

4 Methods **25**
4.1 Compression 25
4.2 File Format 28
4.3 Visualization 30

5 Benchmarks **39**
5.1 Setup 39
5.2 Results 41
5.3 Discussion 45

6 Conclusion and Outlook **57**

Bibliography **61**

A Appendix **65**
Zusammenfassung in deutscher Sprache 65

List of Figures

2.1 2D two-phase VOF dataset (a) and corresponding PLIC surfaces (b) 18
2.2 Three-phase PLIC example 19

4.1 Example of a 2D three-phase VOF dataset with tree nodes. Inner nodes are green, hybrid nodes are blue, leaf nodes are red. 27
4.2 Tree structure for the example dataset shown in Figure 4.1. The children of each node are ordered from left to right and from top to bottom. 27
4.3 Visualization of the PLIC algorithm 31
4.4 LOD system example 36
4.5 LOD system and multiphase datasets 37

5.1 Compression times for the files of the *drop-col-2phase* dataset 42
5.2 Compression times for the files of the *Jet-NN-DN* dataset 43
5.3 Compression rates in relation to non-zero/one cells 43
5.4 Compression rates in relation to total amount of cells 44
5.5 Total frame time of the LOD system for all datasets when using the CPU. 45
5.6 Total frame time for all datasets except *Jet-NN-DN* when using the CPU. 48
5.7 Cell search time of the LOD system for all datasets except *Jet-NN-DN* when using the CPU. 48
5.8 PLIC calculation time for all datasets except *Jet-NN-DN* when running on the CPU. 49
5.9 Time it took to copy vertices into the vertex buffer when using the CPU. 49
5.10 Triangulation time when using the CPU. 50
5.11 Total frame times when rendering on the CPU and GPU. 50
5.12 Total frame times when rendering on the CPU and GPU (zoomed in). 51
5.13 The *drop-col-2phase* dataset in time step 0, 200 and 500 51
5.14 First time step of the *drop-col-G50-SOM5* dataset 53
5.15 Composition of the frame time for the *drop-col-G50-SOM5* dataset. 54
5.16 Composition of the frame time for the *drop-col-G50-SOM5* dataset when reloading the file in every frame. 54
5.17 Composition of the frame time for the *drop-col-G50-SOM5* dataset calculating the PLIC surfaces on the GPU. 55

A.1 Compression times for different datasets using the proposed algorithm 67
A.2 Frame time composition in the CPU benchmark 68
A.3 Frame time composition when reloading the file every frame 69
A.4 Calculation times for different algorithms in the CPU benchmark when reloading the file every frame 70
A.5 Frame time composition in the GPU benchmark 71
A.6 Calculation times for different algorithms in the GPU benchmark 72

List of Tables

5.1 Overview of the datasets 40
5.2 Compression times of different algorithms 46
5.3 Compression rates of different algorithms 47

A.1 Byte layout of the root node from top to bottom. 66
A.2 Byte layout of an inner node from top to bottom. 66

List of Algorithms

4.1 PLIC Calculation 33
4.2 Polyhedron Clipping Algorithm 34
4.3 Polyhedron Clipping Algorithm: Calculate New Edges 35
4.4 Polyhedron Clipping Algorithm: Calculate New Vertices 35
4.5 Polyhedron Clipping Algorithm: Calculate New Faces 35
4.6 LOD Calculation 36

Acronyms

HLOD Hierarchical Level of Detail. 30

LOD Level of Detail. 3

PLIC Piecewise Linear Interface Calculation. 3

SLIC Simple Line Interface Calculation. 22

VOF Volume of Fluid. 3

VTK Visualization Toolkit. 21

1 Introduction

Fluid simulations continue to be an important field of research. The ever-increasing need for more accuracy results in larger datasets generated by those simulations. This is especially noticeable in the context of fluid simulations as the often used three-dimensional grid data structures grow rapidly when increasing the resolution of the simulation. They are necessary to simulate systems such as offshore oil and gas extraction [YH18]. It can be necessary to understand the flow of the fluids involved in such a system, for instance when the involved fluids pose a hazard. This is the case for oil and gas extraction, where mixtures of seawater, gas and oil need to be simulated. Other examples include nuclear power plants or the simulation of atomization of fuel in combustion engines [EV08] where it is necessary to simulate the flow of the propellant to maximize the efficiency of the engine. Moreover, fluid simulations with two or more fluid phases are becoming more important as the computational workload for them becomes manageable. The VOF method is an approach which can be used to differentiate the fluids involved in a multiphase simulation. On top of the VOF data, additional data for each grid cell is often stored. This leads to very large file sizes which become difficult to handle. As such a necessity for data compression arises. VOF data tends to be highly spatially coherent. This makes it a good candidate for efficient lossless compression.

In other fields, algorithms specific to a type of data have been developed to compress datasets if they become too large. A common example is image and video compression. For images especially, both lossy and lossless compression algorithms are in everyday use, like the PNG and JPEG file formats [KC16]. Video compression algorithms are still an active field of research, as compression algorithms such as AV1 [HLM+21] were developed relatively recently but are already heavily used. More relevant for the case of fluid simulation data is the compression of three-dimensional volumetric datasets. Those kinds of datasets are for instance found in the medical field as a result of CT and MRI scans. Work has been done to compress those datasets, both in a lossless [FY94] and lossy [CHF96] way. However, none of those methods are specifically designed to make use of the spatial coherency of VOF datasets. This means there might be room for improvements when compressing those datasets.

On top of the issue of compression, it is often necessary to visualize the state of a fluid simulation. This allows analyzing how the fluid flows or how it interacts with other fluids or solids. Commonly, this is done by rendering the surfaces of the simulated fluids. Additional data such as flow velocity may be overlaid onto those surfaces for a more detailed analysis of the behavior of the fluid.

In this thesis, an approach is presented with which Volume of Fluid (VOF) datasets can be compressed. Analogous to the image compression methods mentioned above, it is specifically designed to efficiently compress VOF datasets. Unlike other methods for compressing volumetric data the goal was to make use of the spatial coherency of VOF datasets. A custom hierarchical tree data structure is used to represent the data. This data structure forms a sparse representation of the stored data, reducing its memory footprint. Furthermore, a custom file format to store this data structure as a file is presented. A benchmark is conducted to compare this approach to previous work

in terms of compression speed and compression rate. Additionally, a method is shown which can be used to quickly generate a visualization of the fluid surfaces from VOF data that was compressed with the shown approach. This allows analyzing VOF datasets stored in the proposed file format.

2 Fundamentals

As this thesis concerns itself with the data produced by fluid simulations a small overview of this topic will be given first.

2.1 Volume of Fluid Datasets

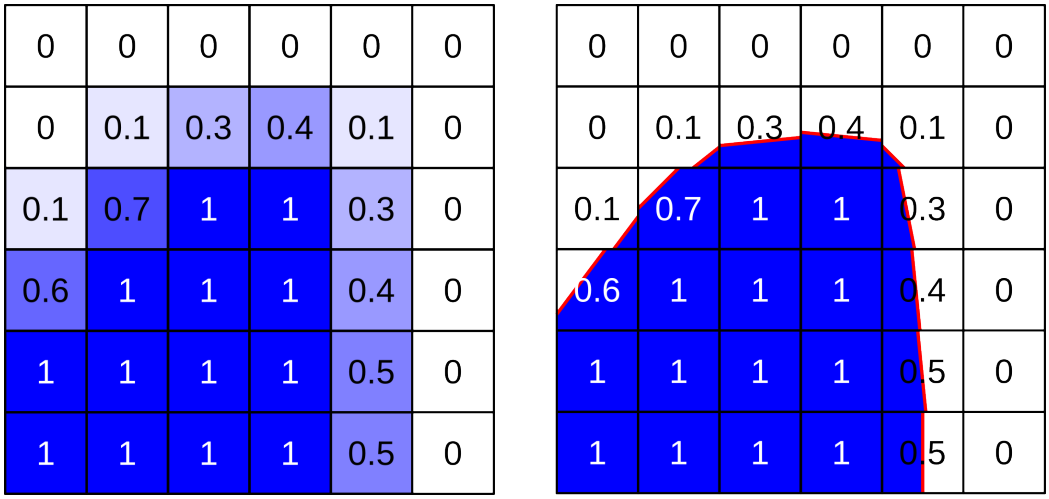
A fluid simulation requires a way to store the state of the simulated fluids on the computer. A popular way to achieve this is to discretize the spatial domain into cells and then storing information about the fluids in those cells. The different materials in a fluid simulation are called phases. This does not necessarily imply that the phases of a fluid simulation have a different physical phase. For instance oil and water would be two different phases even though physically, they are both present i liquid phase. If there is more than one phase involved, a way to differentiate between the different phases is required. Hirt and Nichols [HN81] introduced the Volume of Fluid (VOF) method as a technique to achieve this. It is based on a so-called VOF function. For two-phase flow, the VOF function is a scalar function. It maps each cell of the grid to a number between zero and one as shown in Figure 2.1a. This number describes the fractional volume of a cell that is occupied by one phase. The other phase occupies the rest of the cell.

When more than two fluid phases are involved, there are multiple scalar VOF functions. Every phase except one has a corresponding VOF function. As such, the amount of functions is one less than the amount of phases. The fractions of the volumes of each phase add up to one for each cell.

The visualization of the data requires the calculation of the surface normal of the fluid. For two-phase datasets this can be calculated as the gradient of the VOF function. When more than two phases are involved, this is not possible in some cases. Notably, it is not possible for grid cells which contain more than two fluid phases or their neighbors. This is because those are regions where two interfaces of phases intersect. Effects such as adhesion can force the interfaces to be bent. This results in sharp edges in the interface where one interface meets another. The gradient of the VOF function cannot reproduce this. That is why this gradient was stored in the VOF datasets used in this work. They were computed and stored by the simulation software when the simulations were run.

2.2 PLIC Surfaces

Piecewise Linear Interface Calculation (PLIC) [RK98] is a method to reconstruct an approximation of the interface between the fluid phases from a VOF function. For each fluid phase, it generates one flat polygon in every grid cell where one phase interfaces with another phase. This polygon is oriented in such a way that it approximates the surface of the fluid phase as shown in Figure 2.1b. This can be computed by orientating the polygon in such a way that its normal aligns with the



(a) **(b)**

Figure 2.1: 2D two-phase VOF dataset (a) and corresponding PLIC surfaces (b)

gradient of the VOF function. The polygon bisects the cell into two regions. One of the regions is an approximation of the volume occupied by the fluid phase corresponding to the VOF function. The other region is not occupied by that fluid phase. As such, the polygon needs to be positioned such that the region describing the fluid phase has the correct volume according to the value of the VOF function. This is achieved by moving it along its normal. The process of moving the plane can be done with an iterative algorithm to find the approximate position of the plane.

Those polygons combine to form a piecewise linear approximation of the surface of the fluid phase. Since the polygons generally do not line up perfectly, there are holes in the approximated surface.

In cases where more than two fluid phases are involved, a polygon is created for each VOF function as seen in Figure 2.2. The polygon for an additional VOF function does not bisect the volume of the entire grid cell. Instead, it bisects the remaining volume after the PLIC surface for the previous VOF has been calculated. This avoids different phase interfaces intersecting each other. It also means the order in which the VOF functions are processed is important as it determines which PLIC surface bisects the entire cell and which only bisect parts of the cell.

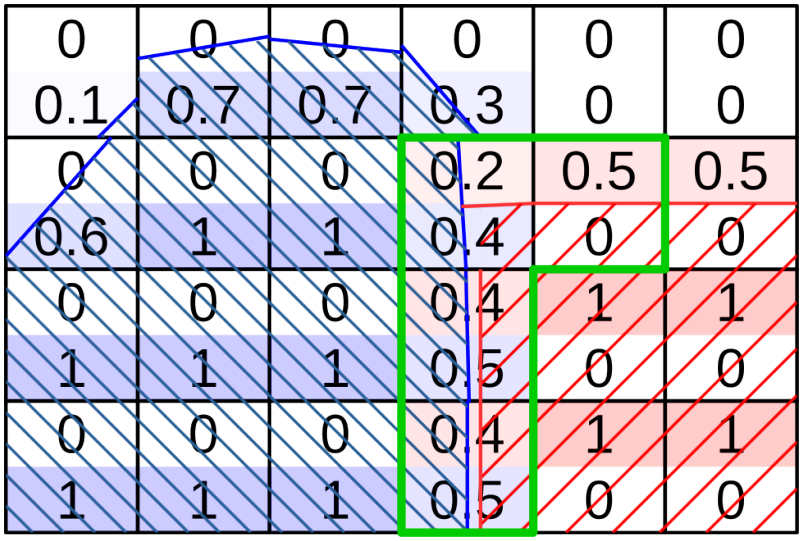


Figure 2.2: 2D three-phase PLIC example. For the cells marked green, the surface normal cannot be computed as the gradient of the VOF function.

3 Related Work

Previous work exists for both the compression of volumetric datasets and the visualization of fluid simulations. An overview over those topics will be given here.

3.1 Compression

Datasets storing three-dimensional grids tend to be large since they scale with the cube of the side length. As such, the problem of compressing them efficiently is old. However, no methods have been developed specifically for VOF datasets.

Fowler and Yagel [FY94] describe an algorithm based on combination of differential pulse-code modulation and Huffman coding to compress volume datasets. They find that this approach leads to a compression rate of roughly 2:1 for the datasets tested. Those were however datasets obtained from MRI and CT scans as well as electron-density-map data. This means that the values in the volume were more likely to be between zero and one than the values of a VOF function. As such, the data itself may not contain as much redundancy as a typical VOF dataset.

Another approach to compress arbitrary volume datasets was presented by Cochran et al. [CHF96]. It uses fractal compression to lossily compress the data. Similar to the algorithm described by Fowler and Yagel [FY94], this was used to compress volume data from CT scans. It achieves a compression rate of 60:1 with a reasonable loss in fidelity. Higher compression rates are however possible.

Saupe and Kuska [SK02] show a compression method relevant to the visualization of volumetric datasets. Instead of storing the volumetric data itself in a compressed format, they describe a way to compress isosurfaces generated from the volumetric data. Generating isosurfaces from VOF data would be possible. As such, this approach could be used to generate a way to visualize VOF data with a small memory footprint. However, the original volumetric data would be lost.

A more generic approach to volumetric data compression is provided by the Visualization Toolkit (VTK) [SML06]. VTK is a library providing functionality for visualizing various kinds of data, including volumetric data. It comes with its own file format derived from XML, which can be used to compress data. The data stored in the file is compressed as a block using the common compression algorithms LZMA, LZ4 or zlib's DEFLATE.

A file format developed specifically for storing volumetric datasets is OpenVDB which was introduced by Museth [Mus13]. It is meant for storing datasets containing volumetric effects for computer generated imagery. OpenVDB uses a hierarchical data structure to store the volumetric data. This way, the data structure of OpenVDB has an adaptive resolution. Regions which need to be detailed have a high resolution while that do not need it do not take up as much memory by having a reduced resolution. OpenVDB uses a tree with large branching factors. This results in a

relatively flat hierarchy while still benefiting from the adaptive resolution. This is similar to the approach proposed in this thesis. However, this thesis uses a fixed, relatively low branching factor for simplicity.

3.2 Fluid Visualization

To analyze the results of a fluid simulation it is helpful to be able to visualize them. There are various ways to achieve this. Different applications may necessitate different visualizations.

For visualizing the interface between fluid phases, a representation of this interface is necessary. Since this interface is only implicitly described by the volumetric data, it is necessary to reconstruct it. There are various approaches of doing this, with differing degrees of sophistication. Most of them stem from the necessity of interface reconstruction for the simulation itself. Yet, they can be used for visualizing the fluid surface. The approaches can be largely grouped into methods that produce continuous surfaces and methods that produce surfaces with discontinuities.

Bonnell et al. [BDS+03] present an approach for interface reconstruction that produces continuous surfaces. It works on regular grids. Each cell of the dual graph of the grid, which is also a regular grid, is split into six tetrahedrons. The fractional volumes of the fluid phases from the VOF function values are interpreted as barycentric coordinates. This way, it can be projected into an n-simplex where n is the number of fluid phases. This n simplex is subdivided into different regions using a voronoi decomposition with the vertices of the simplex as seed points. Each region corresponds to one of the vertices of the simplex and each vertex corresponds to a fluid phase. The surfaces between the cells of the voronoi decomposition mark the phase interfaces. These surfaces are then unprojected into the tetrahedron using the barycentric coordinates. This results in a segmentation of the tetrahedron into different parts with the parts having volumes according to the VOF functions.

Another approach to reconstruct continuous fluid interfaces is shown by Bargteil et al. [BGOS06]. It is based on a signed-distance function that describes the fluid surface. The signed distance function is stored within an octree. This octree is built in such a way that it has the highest level of detail near the implicit surface. An explicit representation of the surface is then obtained with an algorithm that is based on the marching cubes algorithm by Bloomenthal [Blo94]. The implicit surface is entirely confined to cells of the octree where the signed distance function has a zero crossing. For each of those cells, the marching cubes algorithm is run. This can be done in every time step of the simulation. Using this, it is possible to map every vertex of the generated mesh to a point on the mesh from the previous time step. That way, it is possible to track points on the fluid surface. This allows for example texturing the surface with the texture coordinates being properly warped when the fluid is moving.

A relatively simple surface reconstruction approach is Simple Line Interface Calculation (SLIC) introduced by Noh and Woodward [NW76]. It approximates the fluid interface for each grid cell separately as one or more surfaces parallel to one of the coordinate axes. It is designed to work with an alternating-direction fluid simulation, that is, a simulation running different passes for different fluid movement directions. The direction of the planes within the cell depend on the direction of the pass the simulation is running. The number of planes which are put in a cell depends on the amount of different fluid phases present in the cell and its neighbors. The position of the planes

within each cell depend on the fractional volume of the fluids within the cell. This results in a rather crude approximation of the surface, however the algorithm is simple to implement and efficient to compute.

PLIC [RK98] is another surface reconstruction method which produces discontinuous surfaces. Using this method, the surface of the fluid is approximated as a polygon per grid cell of the simulation. It produces a much smoother appearance than SLIC, however there can still be holes and steps in the surface. Yet, it still is useful for visualization purposes as shown by Karch et al. [KSM+13]. Due to its simplicity but still resonably good fidelity it was chosen as the basis for the visualization algorithm in this thesis. Additionally, PLIC was the method used for interface reconstruction by the simulations which produced the benchmark datasets [EEG+16]. A more detailed description of the PLIC method will be given in Section 2.2 and an implementation in Section 4.3.1.

After generating the surfaces, they can be rendered to visualize the surface of the fluid. This can be done using realistic shading which shows how the fluid would look like in reality or it can be rendered in a way that visualizes additional data. For example Edmunds et al. [ELC+12] present a way to visualize the fluid flow on the surface of the fluid. Obermaier et al. [OCHJ12] show how interface stability can be tracked and visualized on the fluid surface. For simplicity this was not done in the scope of this thesis.

4 Methods

The goal of this thesis was to introduce a method to compress VOF datasets and then be able to use the compressed files to quickly generate visualizations. Compressing VOF datasets and visualizing them seem initially like two unrelated tasks. However, it is of course necessary to read the data in order to be able to visualize it. This was kept in mind when designing the compression algorithm to speed up the file reading times during the visualization. Additionally, the visualization was meant to use an LOD approach for faster rendering. This requires special data structures to handle the different levels of detail. As such, this also had some impact on the design of the compression method.

4.1 Compression

Common compression algorithms such as DEFLATE used by zlib [RAG24] or LZ4 cannot exploit any special structure of the data they are compressing since they have to work for all kinds of data.

In a typical VOF dataset, there are large regions of the dataset which contain exclusively values of zero or one since only the cells which contain more than one fluid phase have a different value. This means that the data is spatially coherent. Additionally, since the data is stored as a grid, it contains a lot of redundancy.

In essence, a grid is a discrete partitioning of space which allows assigning data to a portion of space. Here, this is used to store the VOF function values. Another way to partition the space are hierarchical tree data structures. Octrees in particular are useful in this case. The root node of the octree covers a cube shaped subset of the entire space. Every inner node in the octree has eight children. Each of the eight children covers one eight of the region of its parent. The area covered by a node does not overlap with the areas of any of its siblings. This structure means that there are different levels in the tree which correspond to partitionings of the region covered by the root node with different resolutions. As such, different regions can be stored with different resolutions, which makes this data structure more flexible than a grid.

With an octree data structure like this, large regions in the data which share one VOF function value can be exploited. If the VOF function in the region of a node is constant, the VOF function value only needs to be stored once in that node. This removes redundancy from the data and therefore compresses it in a lossless way. If the VOF function is not constant over the region covered by a node, the node needs children to differentiate between the different function values. This allows the data structure to provide high resolution data where necessary, but not everywhere to save space. It was chosen to use octrees for the data structure in this work because of their simplicity and effectiveness. Due to how octrees work, it is best when the underlying data has a cube shaped domain with the side length being a power of two. This way, the area covered by the octree is not too large. Since not all datasets have the same, power of two grid size along all three dimensions

however, a slight modification of the octree data structure was used in this thesis. The root node was allowed to have more than eight children. This way, the hierarchical data structure more efficiently covers the domain of the dataset.

For multiphase VOF datasets, more than one VOF function needs to be represented. The octrees for each VOF function could have a different structure because the different VOF functions may require more resolution in different areas. As such, storing them in the same octree without any further considerations would not work. It would be possible to create an octree which provides the maximum necessary resolution everywhere, however this would be wasteful as in some areas at least one of the VOF functions would be stored at a higher resolution than necessary.

A way around this would be to simply store one tree per VOF function. However, this inflates the file size significantly, because the actual VOF function values only make up a rather small part of the file size as described in Section 4.2.

A solution to this is to create a hybrid between the trees of the different VOF functions. This means that the tree has enough levels everywhere to represent every VOF function. In areas where one VOF function requires a high resolution and another does not, the value of the VOF function which requires less resolution is stored in an inner node. To sample the value of the VOF function which requires higher resolution, it is necessary to descend the tree further after encountering this inner node. This way, file size can be reduced since all nodes below such an inner node do not need to store the value of that VOF function.

Using this approach, there are four types of nodes in the hierarchical data structure. A flag byte indicates the type of the node.

The root node is special since it can have more than eight children. It also stores some information about the dataset. This includes the amount of grid cells and their size as well as the amount of stored VOF functions.

Hybrid nodes stem from the approach of combining two different trees into one. They store the values of one or more of the VOF functions, but not all of them. To access the values for the VOF functions not stored in a hybrid node, it is necessary to descend the tree further.

Leaf nodes store the values of the VOF functions. They may only store the values of some VOF functions if the values for the other VOF functions is stored further up in the tree in a hybrid node. Additionally, leaf nodes may store the gradient associated with the first VOF function. If this is the case, it is indicated in the flag byte. Both hybrid and leaf nodes need to indicate which VOF functions they store values for. A flag byte is used for this purpose. This limits the amount of VOF functions that can be stored to eight. As such the maximum amount of fluid phases that this data structure supports is nine.

Inner nodes store the VOF function values computed from the VOF function values of all of their children as proposed in Section 4.3. This allows extracting a lower resolution version of the stored data. Both inner and hybrid nodes need to indicate in their flag byte if at least one of their direct or indirect children stores a gradient as this is necessary for the visualization.

An example of this kind of data structure can be seen in Figure 4.1 and Figure 4.2. Figure 4.1 shows the underlying grid with two VOF function values per cell. The nodes of the tree structure are indicated with colored squares. They show which part of the grid is covered by a node. Child nodes are the smaller squares inside bigger ones. Figure 4.2 shows the resulting tree.

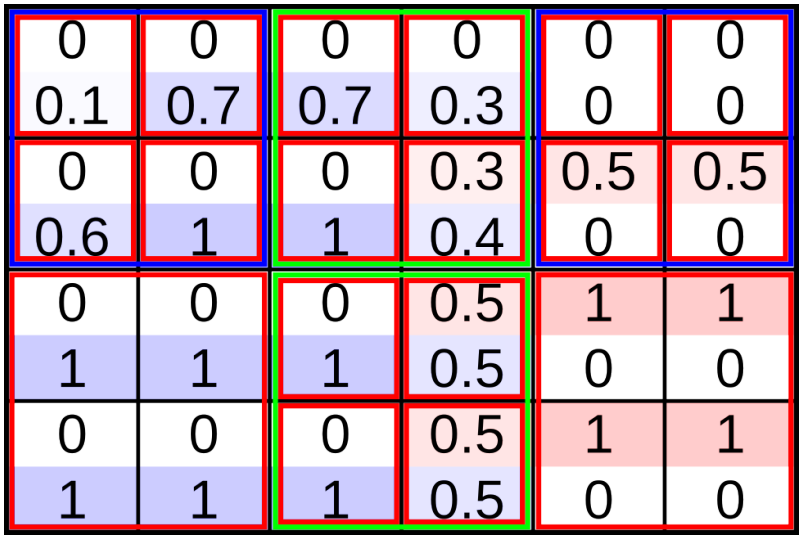


Figure 4.1: Example of a 2D three-phase VOF dataset with tree nodes. Inner nodes are green, hybrid nodes are blue, leaf nodes are red.

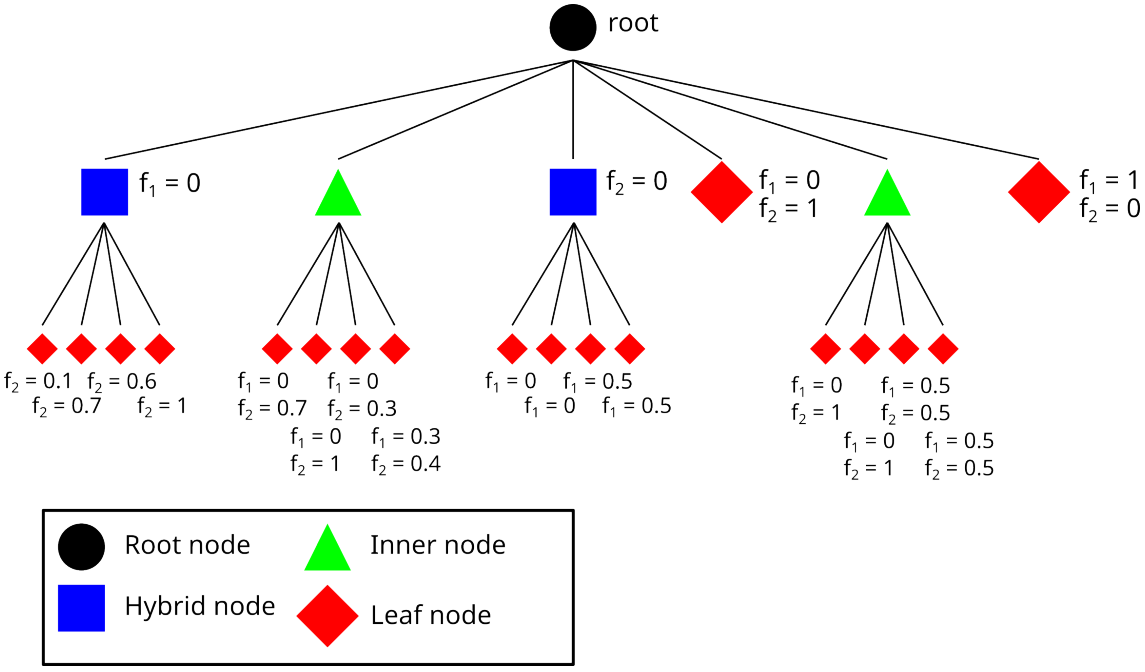


Figure 4.2: Tree structure for the example dataset shown in Figure 4.1. The children of each node are ordered from left to right and from top to bottom.

4.1.1 Creating a Tree from a Grid

An algorithm was implemented which can generate a hierarchical data structure from a data set given as a rectilinear grid from the VTK library.

First, a root node is created and initialized with the parameters from the grid. Building the tree structure from the top down would be inefficient because for every new node it has to be decided whether it will be a leaf node or not. This depends on whether all grid cells covered by the node share a single VOF function value for each VOF function. To check this, it would be required to iterate over all those cells. If differences in the VOF function values are found, the same cells have to be iterated over again for the next level of the tree. In the worst case this means that every cell of the grid has to be checked for every level of the tree.

Because of this, the tree is built from the bottom up. First, a full tree is created which is then simplified by pruning branches. Allocating a full tree is computationally expensive and takes up a lot of memory due to the amount of nodes. As such, the last level of the tree is left out when it is allocated. An octree with one level less has only one eighth of the nodes. As such, this approach can save the creation of up to seven eighths of the nodes compared with creating a full tree.

Because the node in the lowest level in the octree were not allocated at the start, they are created on demand. Each of the nodes in the second to last level covers eight grid cells. The VOF values in those cells are compared. If they differ, eight child nodes are created. If they do not, the node is considered a leaf node and the VOF values are stored in it.

The pruning algorithm traverses the tree in depth-first post-order. This means for every node visited, the children are simplified first.

If, after simplifying the child branches every child is a leaf with the same VOF function value for every VOF function, the child branches are removed and the visited node is turned into a leaf node. If only some of the VOF functions have the same value for every child branch, it is turned into a hybrid node instead. The VOF function values that are constant across the child branches are stored in the visited node, the rest are stored in the child branches.

4.2 File Format

Because disk I/O would slow down the rendering process, one goal was to minimize the amount of data read from disk. To achieve this, the file format was designed such that files can be read only partially. This way, the LOD approach for the visualization does not need to load the full resolution version of the data in areas which are not going to be rendered at full resolution. Because of this, the tree is stored in breadth-first order and each node contains pointers to its children. This allows loading individual nodes and whole subtrees without reading the entire file.

Files in this format are binary files. This reduces the file size, and it is easier to work with since most data in the file are floating point numbers. Using the same binary representation for those numbers in the file and in system memory means no conversion or parsing is necessary. This also speeds up the file reading process.

The file starts with a header containing some information about the file and the saved data. A bit flag describes whether the file contains double precision floating point numbers or single precision numbers. There is a second bit flag which enables additional compression for the whole file. With it enabled, everything except the header is compressed as a single block using the zlib compression library [RAG24]. This has the side effect that it is not possible to only read the file partially in this case since the entire file has to be read in order to decompress it.

After the header, the file contains all nodes of the hierarchical data structure in breadth-first order. Each node starts with the flag byte of the node. This way, the flag byte can be read first to decide how many more bytes need to be read. Then, information specific to that node follows. The size and kind of that information depends on the node. At the end, a list of pointers to the children of the node follows. The pointers are simply the number of the byte at which the data describing the respective child starts. Since the pointers are four byte numbers, they make up a significant amount of the file size. This is especially the case when using single precision floating point numbers.

The first node after the header is the root node of the tree. The information specific to the root node contain some information about the dataset. First are the amount of children of the root node in x, y and z direction since those can vary as discussed earlier. Then, the dataset size and the size in cells of the nodes on the second level of the tree follow. After that, the number of VOF functions stored follows. At the end before the pointers to the children, three arrays containing the cell coordinates follow.

Inner nodes and leaf nodes contain information about the VOF function. First is another flag byte describing which VOF functions are stored in the node. This is used for hybrid nodes and any children below those, since these do not need to store the values of all VOF functions. After the flag byte, the values of the VOF functions follow. Optionally, the gradient for one VOF function is stored at the end which is indicated in the flag byte of the node. A table showing the byte layout of the nodes in the file can be seen in the appendix in Table A.1 and Table A.2.

4.2.1 Reading Files

As described earlier, a lazy loading approach was used to minimize the amount of nodes read from storage. When reading a file, only the header and root node are read first. The nodes of the second level in the tree are left uninitialized. Now, when an algorithm operates on the tree, any uninitialized nodes that are encountered are initialized by loading them from the file. Whenever this is done, children for the previously uninitialized node are created and left uninitialized. This way, the tree is incrementally built over time. It results in only the required nodes being read from file and initialized.

Whenever a node needs to be loaded, the flag byte is read first. This allows computing the size of the data stored in the node. With this information it is possible to read only the bytes necessary for the node.

4.3 Visualization

Multiphase fluid simulations based on the VOF method often use PLIC for interface reconstruction. In order to accurately portray the state of the simulation, it is possible to visualize the surfaces created by the PLIC algorithm directly. The result is an image of the simulated fluid, albeit with some holes in the surface where the cells of the simulation meet.

The main requirement of visualization of the VOF data was to quickly render a frame from a compressed dataset. This can be used to automatically generate an image sequence from a list of datasets containing subsequent time steps of a fluid simulation. It is not suitable for an interactive application which requires fast render times for every frame.

There are several ways to speed up the rendering of one single frame. A lot of time is spent computing the PLIC surfaces. As such, one way to reduce the workload is to introduce a system which reduces the amount of PLIC surfaces rendered. One way to reduce the amount of surfaces rendered is to render regions far away from the camera with lower resolution. This is similar to a LOD system. Since the data is already in a hierarchical structure, it lends itself to a special kind of LOD system, namely a Hierarchical Level of Detail (HLOD) system as described by Clark [Cla76]. The HLOD system by Clark [Cla76] uses a tree of objects to be rendered. In this tree the relation between a parent and its children describes that the parent object can be rendered instead of all its children if the required level of detail is low enough. This is usually the case when the children are far away from the camera. In that case, they take up only a few pixels on the screen and can be replaced by a lower fidelity model, which is the one stored in the parent. The tree structure allows the definition of an arbitrary amount of intermediary levels of detail which are displayed at different distances.

This approach can be transferred to the hierarchical VOF data structure. Instead of computing and rendering every PLIC surface for every cell with a VOF function value between zero and one, cells which are far away from the camera are not rendered. They are instead replaced by computing a single PLIC surface for their parent node and rendering that. This requires that a PLIC surface can be created for the parent node. As such a VOF function value for the parent node needs to be known. For the PLIC surface to have the correct position, this value needs to represent the portion of the parent node that is filled with the corresponding fluid phase. This can be calculated from the children's volume and VOF function values:

$$vof(parent) = \frac{\sum_{child} vof(child) \cdot volume(child)}{volume(parent)}$$

Since the rendering time for the first frame was supposed to be as low as possible, special care was taken to optimize that. Unlike in subsequent frames, for the first frame the data needs to be loaded from the disk. This takes some time and can be optimized. With the introduction of a LOD system, not all PLIC surfaces are rendered. As such, it is not necessary to load the cells from disk which are not going to be rendered. To achieve this, a lazy loading algorithm can be used to only load parts of the compressed file when they are necessary. This means no cells are loaded at first. When an algorithm needs to access data in a cell, it is then loaded on demand.

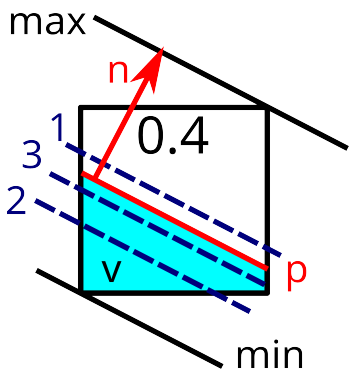


Figure 4.3: Visualization of the PLIC algorithm: In the first iteration the plane is in the middle between the minimum and maximum offsets (*min* and *max*). This is position 1. It is then iteratively moved along its normal n (positions 2 and 3) until eventually the volume v below the plane fills a fraction of the cell approximately equal to the value of the VOF function (here 0.4) and ends up in its final position p.

Another way to speed up the rendering is to execute the PLIC calculation in parallel. Since each PLIC surface is completely independent of the others, this is relatively simple to achieve. As such, it is also possible to do the PLIC calculations on a GPU to leverage their massive amount of parallel computing power.

4.3.1 PLIC Surface Creation

To create a visualization of the PLIC surfaces, they have to be available as polygons such that modern computer hardware can render them as an image. As such, the PLIC surfaces have to be recreated from the data.

PLIC Calculation

Algorithm 4.1 shows the PLIC algorithm which does this. It cuts the cell for which to calculate the PLIC surface with a plane and then iteratively moves that plane along its normal. Figure 4.3 visualizes this process.

First a minimum and maximum offset are calculated. These describe the range in which the plane can moved along its normal before no intersection with the cell happens anymore. These offsets are marked in Figure 5.8. The initial offset of the plane can be chosen such that it starts in the middle between the minimum and maximum offset. An alternative which converges to the final plane position slightly faster is to set the offset of the plane for the first iteration to an interpolated value between the minimum and maximum according to the target volume:

$$offset = min_offset + (max_offset - min_offset) \cdot \frac{target_volume}{cell_volume}$$

This results in the plane being offset in the correct direction before the first iteration is run. It leads to good results using less iterations.

Next, a fix number of iterations of the algorithm are run. In each iteration, the plane is calculated first. The normal of the plane is given, just a point on the plane is required. This point is chosen according to the offset. The polyhedron describing the cell is then clipped using this plane and the volume of the resulting polyhedron is calculated. To calculate the offset for the next iteration an approach similar to a binary search is used: If the resulting volume is too large, the offset becomes the new maximum offset and the new offset is chosen to be in the middle of the minimum offset and the new maximum offset. Otherwise, when the volume is too small, the offset becomes the new minimum offset. Again, the new offset is the middle between minimum and maximum.

The resulting polyhedron after clipping is an approximation of the fluid in the cell. With each iteration the volume of that polyhedron gets closer to the volume of the fluid, and as such the approximation becomes better. Testing has shown that around ten iterations are enough to get a smooth looking approximation of the fluid surface. The PLIC surface is the cut surface of this clipped polyhedron. After those iterations, this face is extracted from this polyhedron and returned.

Since the algorithm is effectively a binary search through the value range of the offset variable, the optimal value is found after at most n iterations where n is the amount of bits used to store the offset variable. In practice this means, when using 32-bit numbers it is not necessary to use more than 32 iterations as it would not have any effect.

For two-phase PLIC Algorithm 4.1 only needs to be run once for each cell with the cuboid describing the cell as input. This is not sufficient for multiphase PLIC. Here, the algorithm needs to be run for every VOF function. Subsequent runs of the algorithm need to use the resulting clipped polyhedron of the previous run. This ensures correct placement of the phase interfaces in cells where more than two fluid phases are present. On top of that, it ensures that different phase interface surfaces do not intersect each other. As mentioned earlier the VOF functions may have sharp edges in areas where two interfaces intersect. This means only the normals of the PLIC surfaces of one VOF function can be computed by using the gradient of the VOF function. The gradients for other VOF functions need to be provided by the dataset. Since the file format described in Section 4.2 only allows storing the gradients for on a VOF function, this limits the amount of VOF functions that can be visualized to two. That means only three-phase VOF datasets can be visualized.

Polyhedron Clipping

As described above in Section 4.3.1 a polyhedron clipping algorithm is required to compute the PLIC surfaces. It is necessary that this algorithm returns not only the intersection points between the edges of the polyhedron and the clipping plane but instead a complete description of the resulting polyhedron including its faces. Since an additional goal was to provide an implementation of the PLIC algorithm for the GPU, the PLIC algorithm must not use any data structures which are not available on the GPU.

Therefore, a custom polyhedron data structure with a custom clipping algorithm were chosen which fulfill these criteria while still being relatively simple. The custom polyhedron data structure describes a polyhedron as an array of three-dimensional vectors containing the vertices of the polyhedron. Another array of index pairs contains the edges. Each pair of indices describes an edge,

Algorithm 4.1 PLIC Calculation

```
function PLIC(cell, normal, target_volume)
    min_offset ← minimum offset along normal in cell
    max_offset ← maximum offset along normal in cell
    offset ← (min_offset + max_offset)/2
    for i = 0, iterations do
        position ← position according to offset
        clipped ← POLYHEDRON_CLIP(cell, {position, normal})
        volume ←POLYHEDRON_VOLUME(clipped)
        if volume < target_volume then
            min_offset ← offset
            offset ← (min_offset + max_offset)/2
        else
            max_offset ← offset
            offset ← (min_offset + max_offset)/2
        end if
    end for
    return cut surface of clipped polyhedron
end function
```

and the indices point into the vertex array. A third array contains the faces. Each face is an array of indices pointing to the vertices that the face is made of. These indices appear in counter-clockwise order.

Algorithm 4.2 to Algorithm 4.5 show the custom algorithm used to clip convex polyhedrons using this data structure. It clips a polyhedron with a plane described by a point on the plane and its normal. The polyhedron it returns lies on the back side of the plane and is closed.

It starts by first marking each vertex in the input polyhedron as either *keep* or *delete* depending on whether the vertex lies on the back or in front of the plane respectively. Then, the edges of the output polyhedron are calculated as shown in Algorithm 4.3. All edges of the input polyhedron which lie completely on the backside of the plane are kept. Edges in front of the plane are discarded. For each edge intersecting the plane, the intersection point is calculated.

Since the polyhedron is convex, each face only has at most two edges which intersect the plane. If the face intersects the plane, the intersections of those edges with the plane form a new edge of the output polyhedron. To keep track of this, each face has a corresponding pair of potentially new vertices. Each newly calculated vertex is inserted into the pairs corresponding to the two faces bordering the intersecting edge. After all edges are processed, the pairs for each face are either still empty or contain exactly two vertices, assuming a well-formed input polyhedron.

Then, the vertex array for the output polyhedron is assembled from the vertices marked as *keep* at the beginning and the intersection points between the edges and the plane. This is shown in Algorithm 4.4. Afterward, the faces of the output polyhedron are constructed by taking the faces of the input polyhedron and removing all vertices marked as *delete* at the beginning as shown in Algorithm 4.5. At the same time, if the pair of new vertices corresponding to the face is not empty,

Algorithm 4.2 Polyhedron Clipping Algorithm

```
function POLYHEDRON_CLIP(polyhedron, plane)
    for all vertices in polyhedron do
        if vertex on front side of plane then
            mark vertex delete
        else
            mark vertex as keep
        end if
    end for
    new_edges, intersections ← CALCULATE_NEW_EDGES(polyhedron)
    new_vertices ← CALCULATE_NEW_VERTICES(polyhedron, intersections)
    new_faces ← CALCULATE_NEW_FACES(polyhedron)
    cut_edges ← []
    for all faces in polyhedron do
        if face has been intersected then
            put {first new vertex of face, second new vertex of face} into new_edges
            put {first new vertex of face, second new vertex of face} into cut_edges
        end if
    end for
    if some edges have been intersected then
        new_face ← []
        stitch together edges from cut_edges into loop of vertices, put into new_face
        if normal of new face is wrong way then
            reverse new_face
        end if
        put new_face into new_faces
    end if
    return new polyhedron with new_vertices, new_edges, new_faces
end function
```

those two vertices are inserted at the correct location into the vertex array describing the new face. Any faces which have less than three vertices left are discarded. This automatically deletes any faces in front of the clipping plane.

Since the output polyhedron does not yet contain the edges which form the cut surface, the pairs of new vertices of each face that are not empty are inserted into the output edge array as new edges. Additionally, the pairs are stitched together in the correct order to form the cut face. This face is inserted into the output face array.

4.3.2 Level Of Detail System

To reduce the amount of computation needed to create the final image, it may not be necessary to compute the PLIC surfaces of every data cell. In real-time graphics, where similar performance concerns are common, an often used approach is a LOD system. By rendering objects that are far away from the camera with lower fidelity, it reduces the computational load.

Algorithm 4.3 Polyhedron Clipping Algorithm: Calculate New Edges

```
function CALCULATE__NEW_EDGES(polyhedron)
    new_edges ← []
    intersections ← []
    for all edges in polyhedron do
        if exactly one of the edge's vertices is marked as delete then
            calculate intersection point between edge and plane
            mark intersection as new vertex
            put intersection into intersections
            for all faces containing the edge do
                put intersection into pair of new vertices for the face
            end for
        else if both vertices of the edge are marked as keep then
            put old edge into new_edges
        end if
    end for
    return new_edges, intersections
end function
```

Algorithm 4.4 Polyhedron Clipping Algorithm: Calculate New Vertices

```
function CALCULATE_NEW_VERTICES(polyhedron, intersections)
    new_vertices ← []
    for all vertices including intersections do
        if vertex is marked keep or new then
            put vertex into new_vertices
        end if
    end for
    return new_vertices
end function
```

Algorithm 4.5 Polyhedron Clipping Algorithm: Calculate New Faces

```
function CALCULATE_NEW_FACES(polyhedron)
    new_faces ← []
    for all faces in polyhedron do
        remove vertices marked as delete from face
        insert pair of new vertices for face into face at correct location
        if face has at least 3 vertices then
            put face into new_faces
        end if
    end for
    return new_faces
end function
```

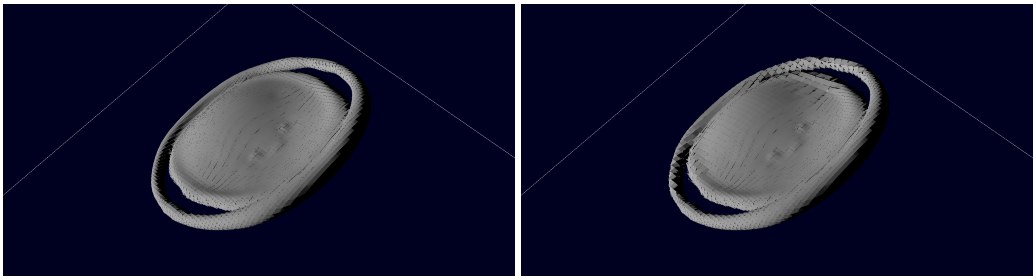


Figure 4.4: The same image without (left) and with (right) the LOD system active. The part further away from the camera has a lower resolution on the right.

Algorithm 4.6 LOD Calculation

```
function FIND_NODES_FOR_RENDERING(cell, camera_position)
    if vof value is zero or one then
        return
    end if
    lod_value ← (distance_to_camera / node_diagonal) · lod_bias
    if (lod_value > 1 ∧ ¬cell contains gradient) ∨cell is leaf then
        add node to list of nodes to be rendered
    else
        for all children of cell do
            FIND_NODES_FOR_RENDERING(child, camera_position)
        end for
    end if
end function
```

The hierarchical data structure can be used to achieve this with the PLIC surface visualization, similar to a hierarchical level of detail system as suggested by Clark [Cla76]. The system proposed by Clark [Cla76] uses a hierarchy of different objects where those further up in the hierarchy are a lower resolution representation of their children. The approach used here adapts this system to replace multiple PLIC surfaces by one with less detail when needed. A comparison between an image with and without the LOD system can be seen in Figure 4.4.

Every Frame, the LOD system needs to decide for which cells to generate PLIC surfaces. It does this by traversing the tree in depth-first order. For every node passed, it is decided whether the node provides sufficient detail. This is shown in Algorithm 4.6. If a node does not provide sufficient detail, the algorithm descends the tree further. The decision whether a node provides sufficient detail depends on the distance from the center of the node to the camera and the length of the diagonal of the node. Additionally, a bias term is used to allow customizing the amount of detail of the final visualization. This way, the size of each cell that is rendered is roughly the same on the screen. This means that all the PLIC surfaces will be roughly the same size on screen as well. The bias term can then be used to maximize performance without reducing visual fidelity by setting it to a value which results in each PLIC surface to roughly take up one pixel on the screen.

The data structure cannot provide a gradient for a lower resolution version of the data in areas where phase interfaces meet. This is because the gradients in those areas are provided by the dataset. Since the datasets were originally given as a rectilinear grid, there is no lower resolution variant of those

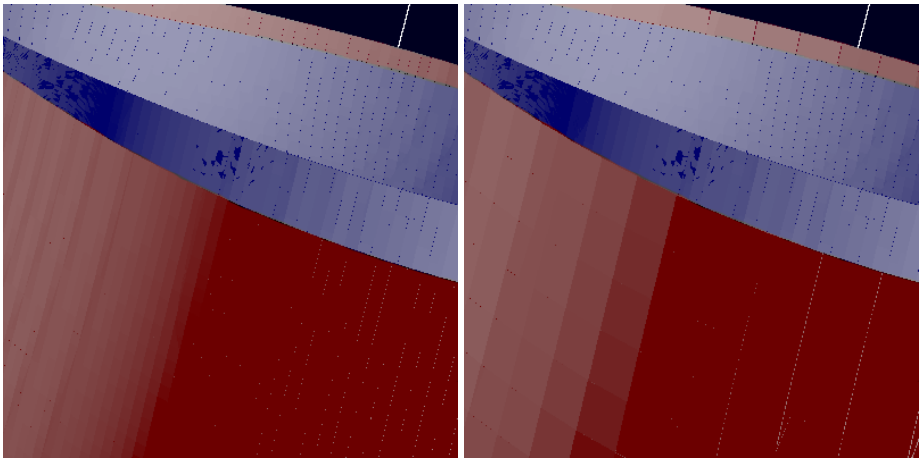


Figure 4.5: The LOD system cannot reduce the resolution in areas where three phases meet: An image of a droplet of one phase within a droplet of another without LOD (left) and with LOD (right). The interface between the outer droplet and the surrounding fluid (red) can be rendered at lower resolution. Due to an air film between the droplets there are three fluid phases present in those areas. Thus, the surface of the inner droplet (blue) cannot be rendered at lower resolution.

gradients available. As such, it is necessary to render those areas at full resolution. Therefore, in case a direct or indirect child of the node contains a gradient, it is necessary for the LOD algorithm to descend to the leaf nodes of the tree. However, this reduces the effectiveness of the LOD system since it increases the number of PLIC surfaces calculated as can be seen in Figure 4.5

4.3.3 Rendering Architecture

MegaMol

MegaMol is a visualization framework developed at the Visualization Research Center University of Stuttgart [GBB+19]. Originally intended for the visualization of particle datasets, it now supports various algorithms and techniques to visualize different kinds of datasets. It is composed of different modules providing different rendering functionality. Each module can be used as a node in a graph which describes the rendering architecture. This graph can be created both via scripting in Lua and manually with the MegaMol graphical user interface.

PLIC computation on the CPU

A MegaMol rendering module was implemented for visualization of the PLIC surfaces of a VOF dataset stored in the file format as described in Section 4.2. This module generates the polygons required to render the PLIC surfaces on the CPU. It integrates with the deferred rendering system provided by MegaMol.

Each frame, it starts by invoking the LOD system. This results in a list of nodes in the hierarchical data structure for which PLIC surfaces are to be created and rendered. Then, the PLIC algorithm calculates the PLIC surfaces for each of those nodes. This is done in parallel to speed up the computation. After the PLIC computation is complete, the vertices of all PLIC surfaces are copied into the vertex buffer which will be sent to the GPU. Then the PLIC surface is triangulated, creating the triangles that the GPU will render. Once all triangles are created, the data is sent to the GPU for rendering.

GPU Accelerated Rendering

A lot of the computational work required for the visualization of PLIC surfaces can be done in parallel. GPUs are known to excel in parallel computing tasks, especially when the computations are similar. Since the same PLIC algorithm has to be performed for every cell that is rendered, this task lends itself to be transferred to the GPU. Theoretically both the calculation of the gradients of the VOF function and the calculation of the PLIC surface polygons can be done on the GPU. However, the calculation of the gradients requires data from neighboring cells. Because the file reader reads only nodes of the hierarchy which need to be accessed it is possible that these cells have not been loaded from the hard drive yet and need to be loaded in while the calculation is running. Doing this on the GPU would require loading the entire file at once before sending any data to the GPU. Alternatively, it may be possible to use technologies such as Nvidia GPUDirect Storage to load files partially directly onto the GPU. However, doing so was not deemed feasible in the scope of this thesis. As such only the calculation of the PLIC surfaces has been ported to run on the GPU.

This has been implemented as another MegaMol rendering module. This second module also integrates with the deferred renderer of MegaMol. For simplicity, the GPU implementation was done in the geometry shader in GLSL, since existing rendering architecture in MegaMol uses a rendering pipeline with vertex, geometry and fragment shaders.

The dynamic data structures required to represent the polyhedrons in the polyhedron clipping algorithm were implemented using static arrays, which limits the complexity of the polyhedrons. This is however not a problem for this use case, as the polyhedrons involved are limited to no more than eight faces. That is because each run of the PLIC algorithm potentially increases the number of faces of the polyhedron describing the volume in which the PLIC surface can lie by one. The PLIC algorithm is run once for each VOF function. The first run starts with a cuboid, which has six faces. When it intersects this cuboid with a plane, a seven-sided polyhedron may be produced. As such, the second run needs to place a PLIC surface in a polyhedron with seven faces or fewer. During the computation of the second run, an eight-sided polyhedron may be produced. Since no more than two VOF functions are supported by the algorithm, this is the maximum amount of faces.

However, this data structure using static arrays led to problems with the Nvidia shader compiler as it would not compile the shader. The exact reason is unclear, but it is likely that the arrays require too much memory. Other than that it works similarly to the CPU implementation described in Section 4.3.3, all computations from the PLIC computation onward happen on the GPU in the geometry shader.

5 Benchmarks

Benchmarks were conducted to estimate the speed and compression rate of the compression algorithm. This allows comparisons with other common compression methods. The visualization system was tested separately to assess its rendering performance.

5.1 Setup

To measure the performance and efficiency of the compression algorithm, files from eight VOF datasets were compressed. All datasets were created with simulations using FS3D by Eisenschmidt et al. [EEG+16]. The datasets *drop-col-G50-SOM5* and *drop-col-G50-SOM5-10* were created by Potyka and Schulte [PS24]. The *drop-col-G50-Perfluo*, *drop-col-G50-SOM3* and *drop-col-G50-SOM20* datasets were introduced by Potyka et al. [PSP23]. All five of those datasets are different variations of two droplets of different fluids colliding. In all cases one droplet ends up inside the other with some smaller droplets being scattered around them. They contain $768 \times 384 \times 192$ (*drop-col-G50-Perfluo*: $1024 \times 512 \times 256$) cells. The *jet-collision* dataset was introduced by Potyka et al. [PSW+24]. It is a simulation of droplets colliding with a jet of a different fluid and the has the largest resolution of all tested datasets in this benchmark with $2048 \times 1024 \times 256$ cells. Ertl (Ertl and Weigand [EW17] and Ertl [Ert19]) created the *Jet-NN-DN* dataset which contains a fluid jet breaking up. The *drop-col-2phase* dataset contains two droplets of water colliding with each other. The collision happens slightly off-center at high velocity. It is the smallest dataset tested with a resolution of $256 \times 256 \times 256$ cells. Table 5.1 shows an overview of these datasets. Each file contained one time step of a fluid simulation. In total 39 files were processed. The *drop-col-2phase* and *Jet-NN-DN* datasets do not contain any hybrid nodes or gradients in compressed form because they contain only two-phase VOF data. Some of the datasets contained additional data such as flow velocity. These were stripped from the files before the benchmarks since they cannot be stored in the proposed file format.

The performance of the compression algorithm was compared to that of the 'zip' file format and the built-in compression in VTK. This way, it is possible to assess whether the proposed method is an improvement over existing solutions. For the zip compression the commonly used Info-ZIP [Roe08] tool set with default settings was used. This means the compression level was six on a range from zero (no compression) to nine (optimal compression). VTK provides optional compression via zlib [RAG24], LZ4 and LZMA. For the benchmark, LZ4 and zlib were chosen for comparison. LZMA was not considered as it was found to be much slower than zlib [AKSV15]. For both LZ4 and zlib the compression level was kept at the default value, which is five on a scale from zero (no compression) to nine (best compression). The time it took to read the original file, compress it and then write the data was measured. Before running the benchmark, the disk cache was cleared to avoid side effects.

Dataset	Dimensions (Cells)	Time Steps	Phases	Size per Step
drop-col-2phase	$256 \times 256 \times 256$	6	2	65 MB
drop-col-G50-Perfluo [PSP23]	$1024 \times 512 \times 256$	6	2	2.6 GB
drop-col-G50-SOM20 [PSP23]	$768 \times 384 \times 192$	6	3	1.1 GB
drop-col-G50-SOM3 [PSP23]	$768 \times 384 \times 192$	5	3	1.1 GB
drop-col-G50-SOM5 [PS24]	$768 \times 384 \times 192$	5	3	1.1 GB
drop-col-G50-SOM5-10 [PS24]	$768 \times 384 \times 192$	5	3	1.1 GB
jet-collision [PSW+24]	$2048 \times 1024 \times 256$	4	3	11 GB
Jet-NN-DN [EW17][Ert19]	$1536 \times 512 \times 512$	2	2	1.6 GB

Table 5.1: Overview of the datasets. The size per step is the size of one file containing one time step of the simulation in the uncompressed VTK rectilinear grid file format without unnecessary data such as the velocity field.

To benchmark the visualization, the same datasets were visualized in three different benchmarks. The first benchmark was meant to test the performance of the rendering algorithms. This includes the LOD system and the PLIC surface creation. Since file loading and potentially other initialization related delays from MegaMol itself may impact the results, the first ten frames after starting MegaMol or after loading a new file were discarded.

For every file from the dataset, visualizations were created from twenty different camera perspectives. Ten of those perspectives were laid out on a circle around the origin, looking at the dataset. This simulates an animation of the camera moving around the dataset. The other ten perspectives were random perspectives inside the unit cube with one corner at the origin and another at $(1, 1, 1)$. For each perspective ten frames were rendered. This means for every file a total of 200 frames were rendered. The total frame time was measured as well as the time the LOD algorithm took to find the cells for rendering. Additionally, the PLIC calculation time and the time it took to triangulate and upload the PLIC surfaces to the GPU was measured.

Since one goal was to optimize the time to render a single frame, the second benchmark did the opposite and reloaded the file after every frame rendered. The system file cache was cleared after every frame as well to make sure that the files are actually being loaded from system storage. This simulates the rendering of an animation of different time steps of a simulation where a new file has to be loaded for every frame.

For the second benchmark, ten random perspectives as described above were rendered for each file. Only one frame per perspective was rendered due to the additional workload of reloading the files every frame taking a longer time.

A third benchmark of the visualization was used to assess the performance of the GPU implementation. It was similar to the first one in that the files were only loaded once and the first ten frames after loading a file were ignored. Both orbit and random camera transforms were tested here. Here, the time the LOD algorithm took was measured. Additionally, the time it took to calculate and upload the data required for the PLIC calculations was measured. This includes the computation of the gradients of the VOF function as they are calculated on the CPU. Additionally, the time of the draw call that the visualization system makes to the GPU was measured. This is the amount of time that the GPU needed to calculate and render the PLIC surfaces.

The compression benchmarks and the visualization benchmarks using the CPU were done on a computer with an AMD Ryzen Threadripper 3995 WX with 256GB of RAM and an Nvidia Quadro RTX A6000 GPU. For the CPU variant of the visualization algorithm, the same hardware was used. Due to the issues with the shader code not compiling with the Nvidia GPU driver on the benchmark hardware, a computer with different hardware was used to benchmark the GPU variant of the visualization system. This computer is running on an AMD Ryzen 5 5600X with 32GB of RAM and an AMD RX6600XT GPU. Unfortunately, the performance difference between the two computers makes the results hard to compare.

5.2 Results

5.2.1 Compression Algorithm

Table 5.2 shows the time it took the tested compression methods to compress each file. VTK with LZ4 was the fastest compression algorithm in every case. Depending on the file it was between 68% and 82% faster than the algorithm introduced in this thesis. On average, it was roughly 74% faster than the new algorithm. Zip was the second fastest algorithm on average. It was between 63% and 76% faster than the proposed algorithm. On average it was 70% faster than the proposed algorithm. VTK with zlib was slower than VTK with LZ4 but is on average still 45% faster than the proposed algorithm. It was in every case at least 20% faster than the new algorithm. This was the best case for the newly introduced algorithm. Overall, the introduced algorithm was the slowest for every file by a large margin.

Figure 5.1 shows how much time was spent in different stages while compressing the files from the *drop-col-2phase* dataset. It is immediately visible that the most time is spent during the conversion from the rectilinear grid to the hierarchical data structure. The file reading time from VTK which creates the rectilinear grid is relatively short. The serialization of the hierarchical data structure into the proposed file format and subsequent writing of the file take even less time. Additionally, it seems the compression time is very similar for the different files in this dataset. An exception to this can be seen in Figure 5.2. For the *Jet-NN-DN* dataset the writing process took roughly half as long as the conversion to the hierarchical data structure. However, this dataset was the only one where this behavior occurred.

The compression rate of each tested algorithm is shown in Table 5.3. Here, LZ4 performs the worst for almost every file. Only for three of the files from the *drop-col-2phase* dataset and from both of the files from the *Jet-NN-DN* dataset the algorithm proposed in this thesis performed worse. Especially the files from the *Jet-NN-DN* dataset are outliers, however. When not considering those files, the average compression rate of the proposed method drops from 0.88% to 0.44%. This puts it at third place after zip and VTK with zlib. With the outliers removed, zip achieves a roughly 33% better compression rate on average than the proposed method. Meanwhile, VTK with zlib is roughly 11% better and VTK with LZ4 is about 50% worse.

Figure 5.3 shows the relationship between the amount of non-zero/one cells and the file size after compressing with the proposed algorithm. It can clearly be seen that the file size grows with the amount of non-zero/one cells. Figure 5.4 shows that this is not related to the total amount of cells in the dataset.

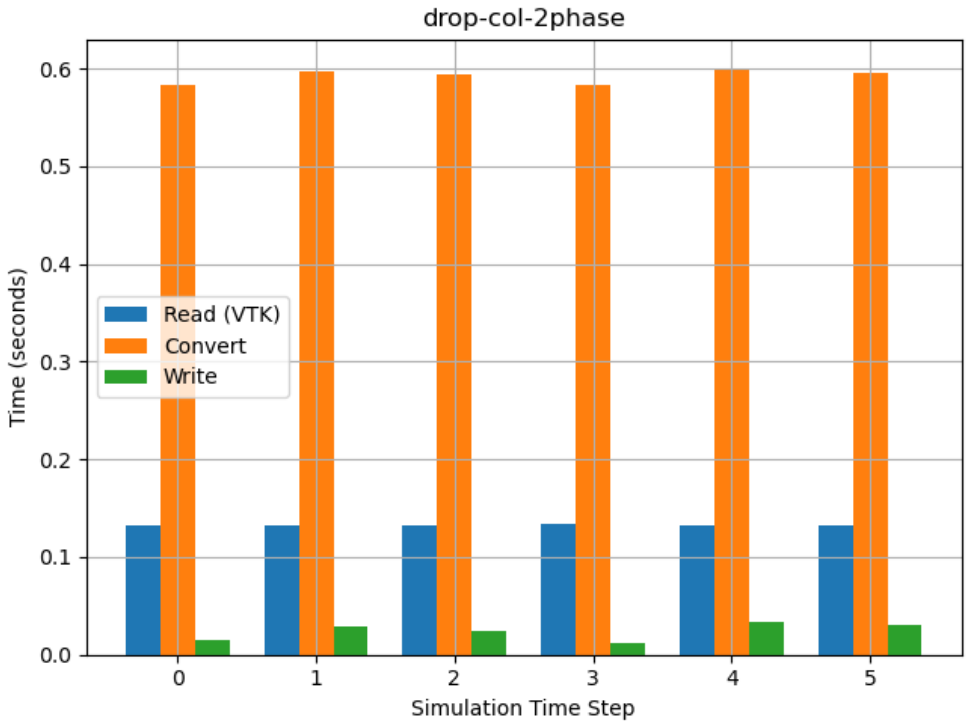


Figure 5.1: Compression times for the files of the *drop-col-2phase* dataset. Read (VTK) refers to the entire process of creating a rectilinear grid from a VTK file. Convert is the process of creating the hierarchical data structure from the grid. Write is the process of serializing the data structure and writing the file to disk.

5.2.2 Visualization

Figure 5.5 shows the rendering performance during the first benchmark of the visualization system for every dataset with both random camera transforms and orbit camera transforms. Here, the PLIC calculation happens on the CPU and files are not reloaded during every frame. The non-zero/one cell count on the x-axis is the number of cells in each dataset where one of the VOF functions is not zero or one. Those are the cells where it is possible to calculate a PLIC surface. Since all the dataset *Jet-NN-DN* has a relatively large number of those cells, Figure 5.6 shows the same graph without it for better readability. It can be seen that the time the frame time has an upper bound that is proportional to the amount of non-zero/one cells. There are some benchmarks where the frame time was lower than this upper bound with a few outliers very close the x-axis where rendering took very little time.

Figure 5.7 shows the performance of the LOD system during the same benchmark. It behaves similar to the total frame time with respect to the amount of non-zero/one cells. However, there are more cases which lie between the upper bound and the x-axis.

The performance of the PLIC computation with respect to the amount of non-zero/one cells behaves similar to the total frame time. Unlike the LOD system, there are few data points between the upper bound and the outliers near the x-axis. This can be seen in Figure 5.8. The non-zero/one cell count on the x-axis is still the total number of cells in the dataset where one of the VOF functions is not

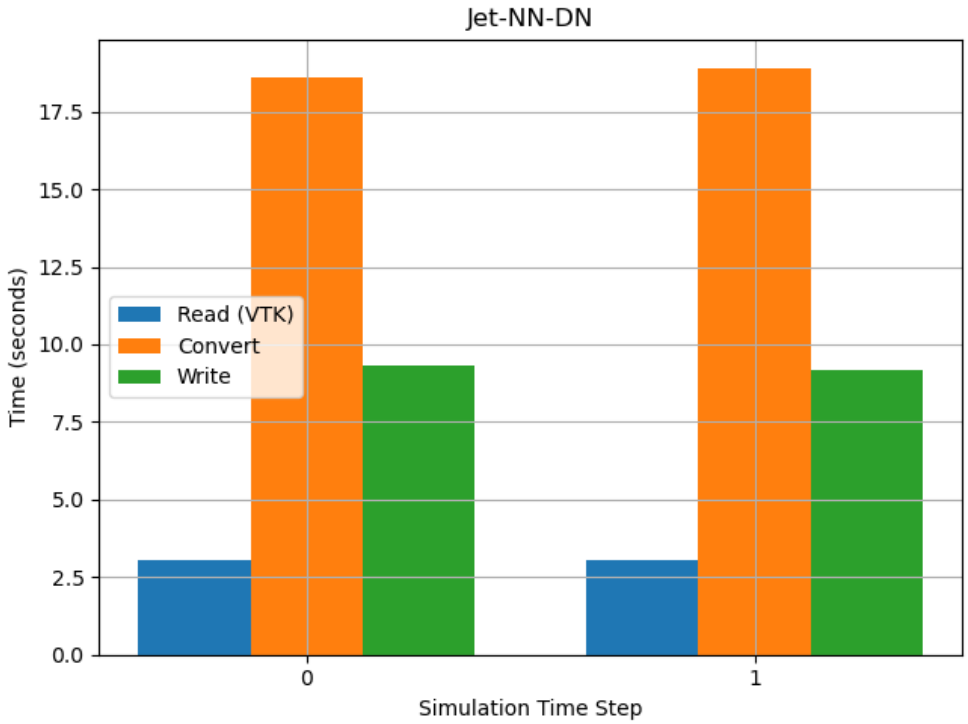


Figure 5.2: Compression times for the files of the *Jet-NN-DN* dataset.

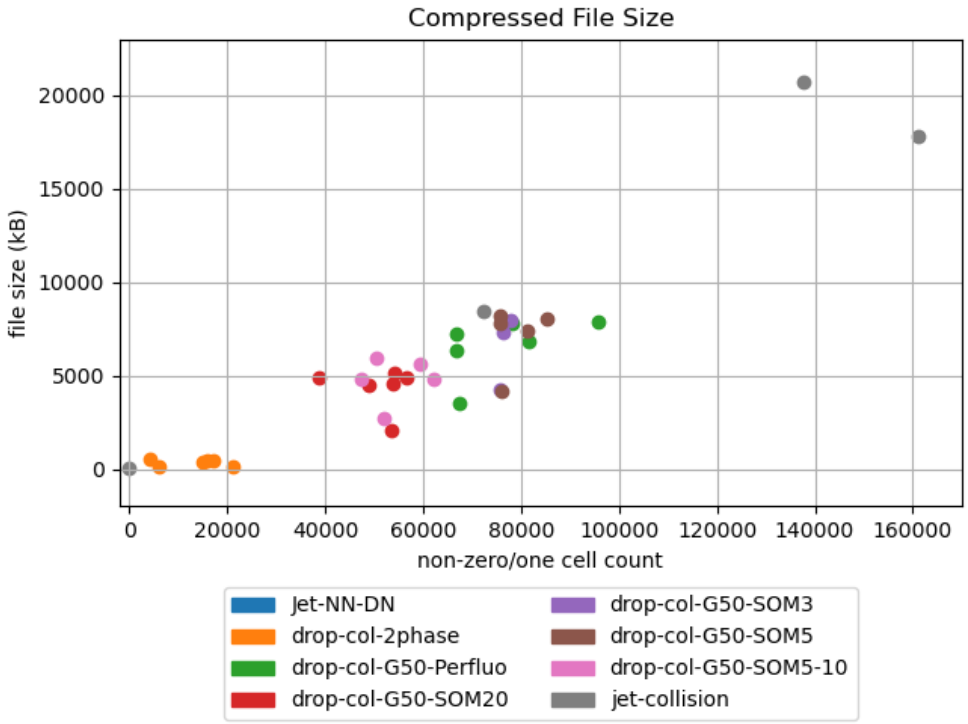


Figure 5.3: Compression rates of the proposed algorithm in relation to the amount of non-zero/one cells in the dataset.

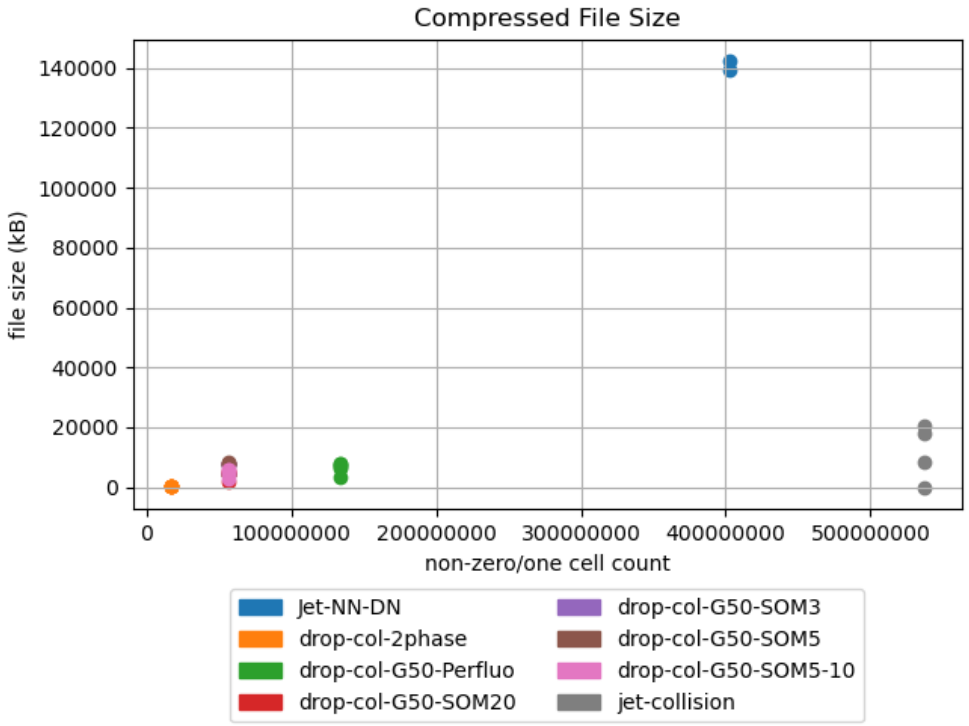


Figure 5.4: Compression rates of the proposed algorithm in relation to the total amount of grid cells in the dataset.

zero or one. However, the PLIC system only calculates a PLIC surface for the cells specified by the LOD system. The PLIC computation takes roughly ten times more time than the LOD system. An example of this can be seen in Figure 5.15 where the PLIC calculation makes up approximately half of the frame time and the time of the LOD system is very small. Figure 5.9 and Figure 5.10 show that the behavior of the time it takes to copy the vertices to the vertex buffer and triangulate the PLIC surfaces is similar to that of the PLIC calculation time. Similar to the LOD system, however, those processes take relatively little time.

Figure 5.15 also shows that copying the vertices into the vertex buffer and triangulating the PLIC surfaces take relatively little time. The time labeled as Overhead is comparatively large. Additionally, there is very little difference between the benchmark with random camera transforms and the one with orbit camera transforms.

In comparison, Figure 5.16 shows the benchmark for the same dataset in the case that the file is reloaded for every frame. The camera perspectives used here were random. One can see, that the LOD system takes much longer here than the benchmark without reloading the file. Additionally, for the first file of the dataset the time labeled as Overhead is very large compared to the other files.

The frame times for the GPU benchmark of the same dataset can be seen in Figure 5.17. Here, the length of the draw call makes up the majority of the frame time. This is the time that the GPU required to calculate and render the PLIC surfaces. Both the LOD system and the upload to the GPU required very little time in comparison. The calculation of the data required for the PLIC calculation and the overhead from MegaMol are relatively small.

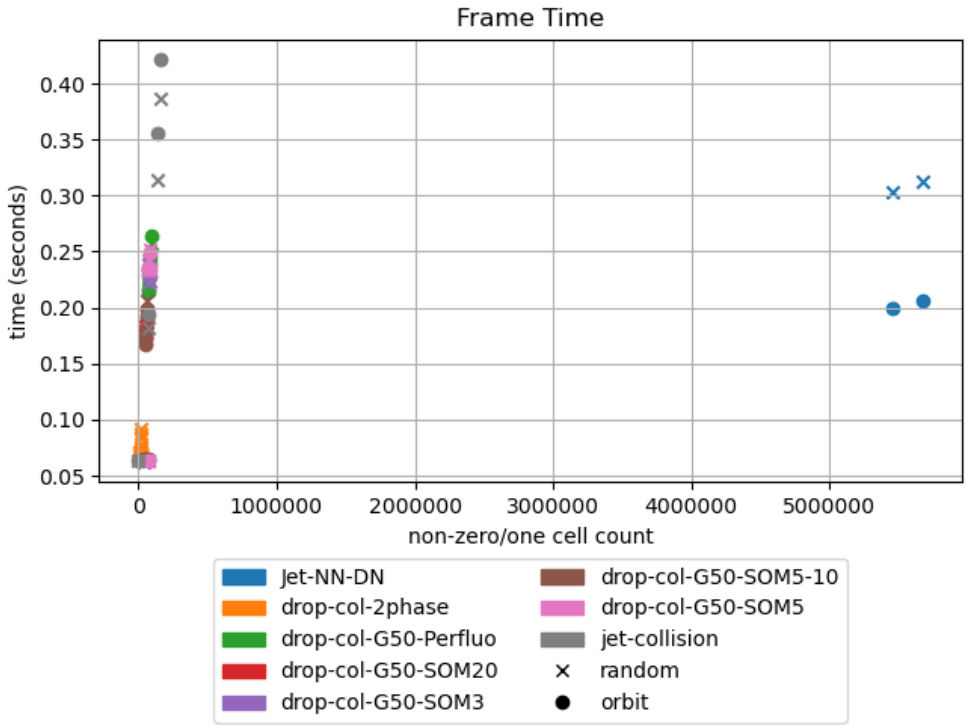


Figure 5.5: Total frame time of the LOD system for all datasets when using the CPU.

A comparison between the frame times of the CPU and GPU benchmarks can be seen in Figure 5.11. It can be seen that the GPU implementation is about two to three times slower. However, Figure 5.12 shows that this is only true when the rendering takes a long time. When the rendering overall takes little time, then the GPU implementation is faster. The point at which the GPU overtakes seems to be around 0.06 to 0.07 seconds.

5.3 Discussion

5.3.1 Compression

One of the reasons why the compression of the datasets took so long using the proposed method is the creation of the hierarchical data structure. Due to the way it was implemented, many memory allocations have to be made. This may slow down the algorithm considerably. Additionally, since it is single-threaded it cannot make use of multi core CPUs.

For the *drop-col-2phase* dataset, the VOF function of the later simulation time steps become increasingly less spatially coherent which can be seen in Figure 5.13. This can also be seen in Table 5.3 where the later time steps have a worse compression rate with the proposed compression algorithm. Yet, results show that the compression process takes a similar amount of time for each of the files. This means that the spatial coherency has little impact on the compression times. Similar behavior can be seen in the appendix in Figure A.1 which shows the compression times for other datasets.

File	zip	VTK (zlib)	VTK (LZ4)	New Approach
drop-col-2phase.0000.vtr	0.27s	0.58s	0.23s	0.73s
drop-col-2phase.0100.vtr	0.27s	0.56s	0.23s	0.73s
drop-col-2phase.0200.vtr	0.27s	0.57s	0.23s	0.75s
drop-col-2phase.0300.vtr	0.28s	0.58s	0.23s	0.77s
drop-col-2phase.0400.vtr	0.27s	0.57s	0.23s	0.76s
drop-col-2phase.0500.vtr	0.28s	0.57s	0.23s	0.76s
drop-col-G50-Perfluo fs3d.0000.vtr	10.71s	21.54s	9.91s	33.99s
drop-col-G50-Perfluo fs3d.0020.vtr	10.71s	22.21s	8.70s	34.31s
drop-col-G50-Perfluo fs3d.0040.vtr	10.79s	20.62s	9.23s	34.43s
drop-col-G50-Perfluo fs3d.0060.vtr	10.65s	22.09s	9.02s	34.69s
drop-col-G50-Perfluo fs3d.0080.vtr	10.70s	21.69s	9.21s	34.48s
drop-col-G50-Perfluo fs3d.0100.vtr	10.66s	22.13s	8.85s	34.72s
drop-col-G50-SOM20 fs3d.0000.vtr	4.50s	8.92s	3.76s	15.21s
drop-col-G50-SOM20 fs3d.0020.vtr	4.55s	9.35s	3.69s	15.35s
drop-col-G50-SOM20 fs3d.0040.vtr	4.52s	9.22s	3.89s	15.40s
drop-col-G50-SOM20 fs3d.0060.vtr	4.55s	9.10s	3.72s	15.54s
drop-col-G50-SOM20 fs3d.0080.vtr	4.63s	9.23s	3.76s	15.39s
drop-col-G50-SOM20 fs3d.0100.vtr	4.54s	9.31s	3.79s	15.37s
drop-col-G50-SOM3 fs3d.0000.vtr	4.57s	9.28s	3.93s	15.49s
drop-col-G50-SOM3 fs3d.0040.vtr	4.60s	9.20s	3.75s	15.57s
drop-col-G50-SOM3 fs3d.0080.vtr	4.61s	9.37s	3.79s	15.76s
drop-col-G50-SOM3 fs3d.0120.vtr	4.63s	9.16s	3.71s	15.86s
drop-col-G50-SOM3 fs3d.0160.vtr	4.72s	9.27s	4.02s	15.68s
drop-col-G50-SOM5 fs3d.0000.vtr	4.54s	8.88s	3.78s	15.27s
drop-col-G50-SOM5 fs3d.0020.vtr	4.75s	9.10s	3.85s	15.34s
drop-col-G50-SOM5 fs3d.0040.vtr	4.61s	9.32s	3.98s	15.58s
drop-col-G50-SOM5 fs3d.0060.vtr	4.66s	9.33s	3.71s	15.57s
drop-col-G50-SOM5 fs3d.0080.vtr	4.63s	9.10s	3.98s	15.33s
drop-col-G50-SOM5-10 fs3d.0000.vtr	4.48s	9.26s	3.77s	15.51s
drop-col-G50-SOM5-10 fs3d.0030.vtr	4.59s	9.46s	3.93s	15.48s
drop-col-G50-SOM5-10 fs3d.0060.vtr	4.58s	9.37s	3.95s	15.74s
drop-col-G50-SOM5-10 fs3d.0090.vtr	4.57s	9.09s	3.91s	15.75s
drop-col-G50-SOM5-10 fs3d.0120.vtr	4.55s	9.67s	3.74s	15.74s
jet-collision.0000.vtr	42.17s	68.87s	36.21s	135.70s
jet-collision.0050.vtr	42.52s	68.99s	36.34s	140.50s
jet-collision.0100.vtr	42.83s	68.81s	36.22s	140.99s
jet-collision.0150.vtr	42.95s	69.03s	38.19s	143.13s
Jet-NN-DN fs3d.0400.vtr	7.58s	14.49s	7.05s	31.15s
Jet-NN-DN fs3d.0600.vtr	7.71s	14.64s	5.66s	31.00s
Average	8.92s	16.22s	7.60s	29.73s

Table 5.2: Compression times of zip, VTK and the proposed file format in seconds, rounded to two decimal places.

File	Original	zip	VTK (zlib)	VTK (LZ4)	New
drop-col-2phase.0000.vtr	65MiB	0.13%	0.21%	0.49%	0.23%
drop-col-2phase.0100.vtr	65MiB	0.15%	0.23%	0.49%	0.24%
drop-col-2phase.0200.vtr	65MiB	0.23%	0.31%	0.59%	0.59%
drop-col-2phase.0300.vtr	65MiB	0.27%	0.37%	0.63%	0.84%
drop-col-2phase.0400.vtr	65MiB	0.24%	0.34%	0.60%	0.74%
drop-col-2phase.0500.vtr	65MiB	0.23%	0.33%	0.59%	0.68%
drop-col-G50-Perfluo fs3d.0000.vtr	2.6GiB	0.16%	0.24%	0.50%	0.13%
drop-col-G50-Perfluo fs3d.0020.vtr	2.6GiB	0.26%	0.37%	0.63%	0.3%
drop-col-G50-Perfluo fs3d.0040.vtr	2.6GiB	0.23%	0.31%	0.58%	0.27%
drop-col-G50-Perfluo fs3d.0060.vtr	2.6GiB	0.24%	0.32%	0.60%	0.3%
drop-col-G50-Perfluo fs3d.0080.vtr	2.6GiB	0.24%	0.33%	0.60%	0.26%
drop-col-G50-Perfluo fs3d.0100.vtr	2.6GiB	0.23%	0.32%	0.58%	0.24%
drop-col-G50-SOM20 fs3d.0000.vtr	1.1GiB	0.18%	0.27%	0.53%	0.19%
drop-col-G50-SOM20 fs3d.0020.vtr	1.1GiB	0.32%	0.43%	0.70%	0.42%
drop-col-G50-SOM20 fs3d.0040.vtr	1.1GiB	0.30%	0.39%	0.67%	0.44%
drop-col-G50-SOM20 fs3d.0060.vtr	1.1GiB	0.32%	0.42%	0.69%	0.46%
drop-col-G50-SOM20 fs3d.0080.vtr	1.1GiB	0.32%	0.41%	0.69%	0.44%
drop-col-G50-SOM20 fs3d.0100.vtr	1.1GiB	0.31%	0.40%	0.68%	0.41%
drop-col-G50-SOM3 fs3d.0000.vtr	1.1GiB	0.27%	0.37%	0.63%	0.39%
drop-col-G50-SOM3 fs3d.0040.vtr	1.1GiB	0.43%	0.55%	0.82%	0.66%
drop-col-G50-SOM3 fs3d.0080.vtr	1.1GiB	0.47%	0.58%	0.86%	0.72%
drop-col-G50-SOM3 fs3d.0120.vtr	1.1GiB	0.48%	0.59%	0.87%	0.72%
drop-col-G50-SOM3 fs3d.0160.vtr	1.1GiB	0.48%	0.59%	0.87%	0.72%
drop-col-G50-SOM5 fs3d.0000.vtr	1.1GiB	0.27%	0.36%	0.62%	0.38%
drop-col-G50-SOM5 fs3d.0020.vtr	1.1GiB	0.45%	0.58%	0.85%	0.67%
drop-col-G50-SOM5 fs3d.0040.vtr	1.1GiB	0.46%	0.56%	0.84%	0.74%
drop-col-G50-SOM5 fs3d.0060.vtr	1.1GiB	0.48%	0.60%	0.87%	0.72%
drop-col-G50-SOM5 fs3d.0080.vtr	1.1GiB	0.47%	0.58%	0.86%	0.71%
drop-col-G50-SOM5-10 fs3d.0000.vtr	1.1GiB	0.21%	0.30%	0.56%	0.24%
drop-col-G50-SOM5-10 fs3d.0030.vtr	1.1GiB	0.31%	0.40%	0.68%	0.43%
drop-col-G50-SOM5-10 fs3d.0060.vtr	1.1GiB	0.36%	0.45%	0.73%	0.54%
drop-col-G50-SOM5-10 fs3d.0090.vtr	1.1GiB	0.36%	0.45%	0.73%	0.51%
drop-col-G50-SOM5-10 fs3d.0120.vtr	1.1GiB	0.32%	0.42%	0.69%	0.43%
jet-collision.0000.vtr	11GiB	0.10%	0.17%	0.44%	0%
jet-collision.0050.vtr	11GiB	0.14%	0.22%	0.49%	0.08%
jet-collision.0100.vtr	11GiB	0.19%	0.28%	0.55%	0.17%
jet-collision.0150.vtr	11GiB	0.21%	0.30%	0.57%	0.2%
Jet-NN-DN fs3d.0400.vtr	1.6GiB	1.71%	1.86%	2.36%	8.85%
Jet-NN-DN fs3d.0600.vtr	1.6GiB	1.77%	1.93%	2.42%	9.06%
Average		0.37%	0.47%	0.75%	0.88%
Average (without Jet-NN-DN)		0.29%	0.39%	0.66%	0.44%

Table 5.3: Compression rates in percent, rounded to two decimal places for readability. Original file sizes are the sizes of uncompressed but stripped VTK rectilinear grid files.

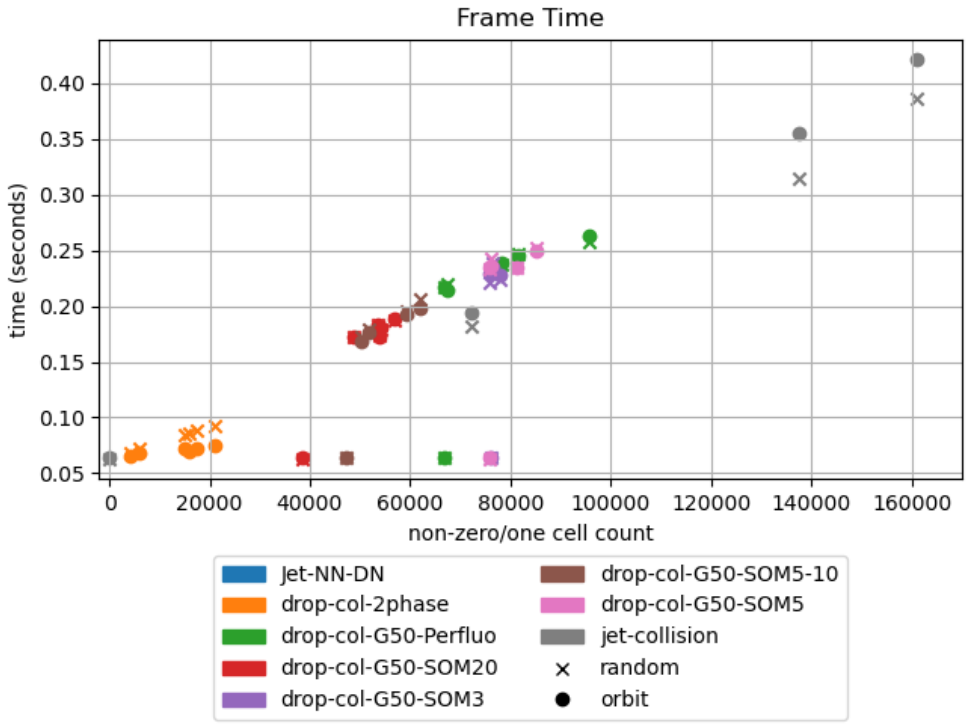


Figure 5.6: Total frame time for all datasets except *Jet-NN-DN* when using the CPU.

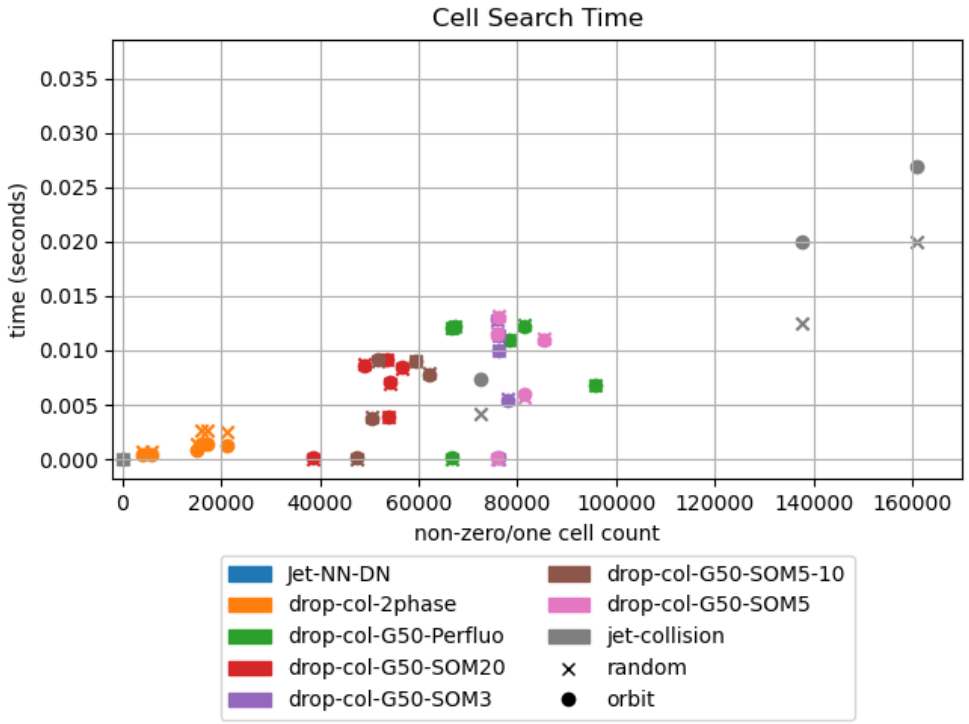


Figure 5.7: Cell search time of the LOD system for all datasets except *Jet-NN-DN* when using the CPU.

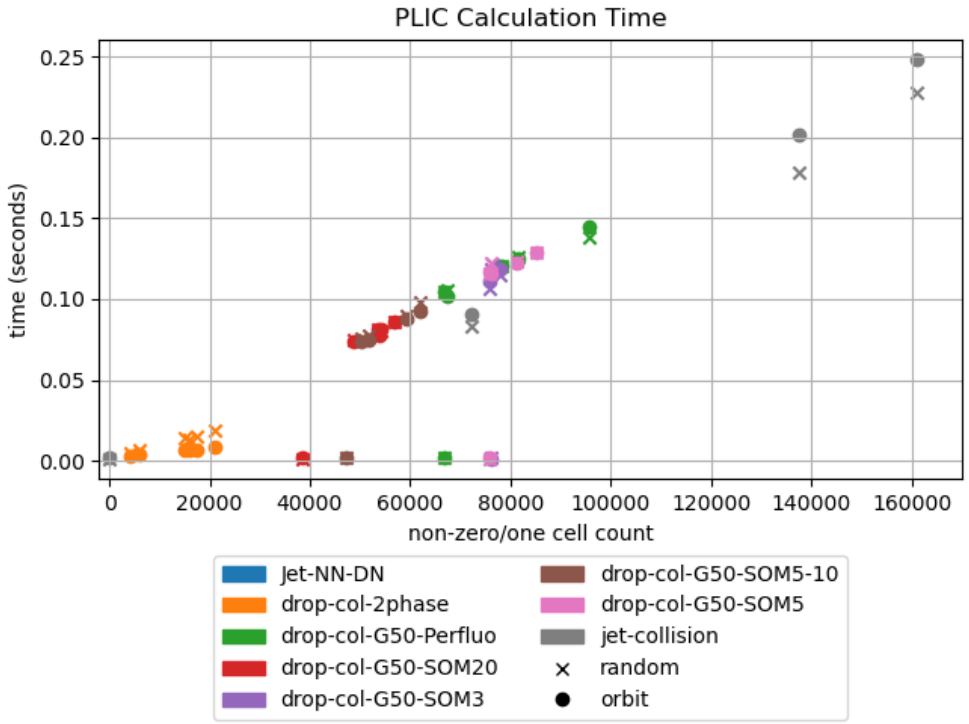


Figure 5.8: PLIC calculation time for all datasets except *Jet-NN-DN* when running on the CPU.

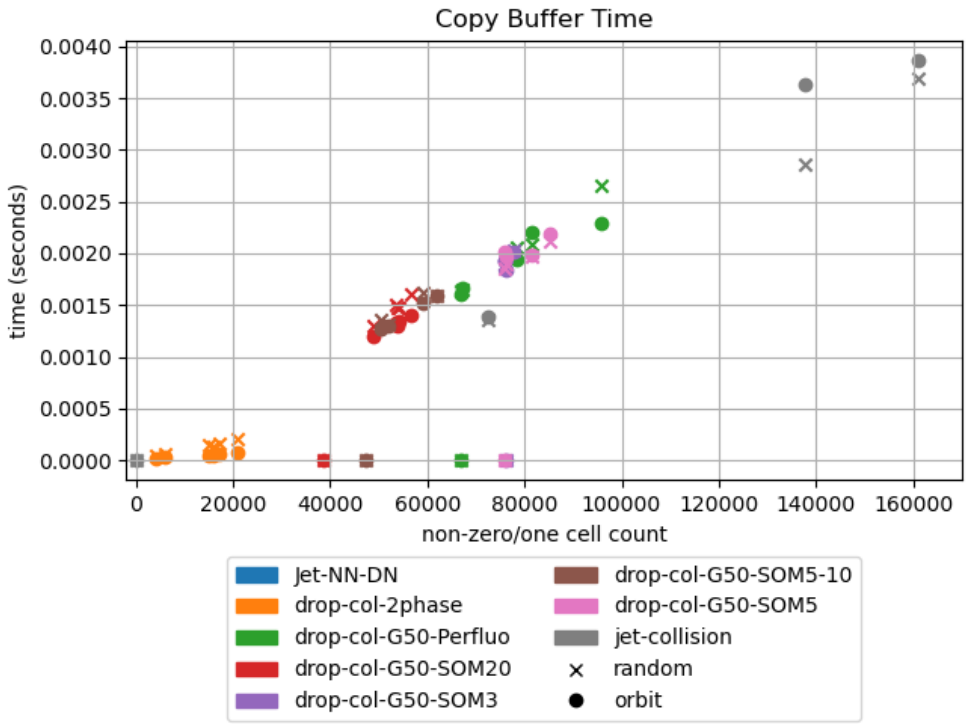


Figure 5.9: Time it took to copy vertices into the vertex buffer when using the CPU.

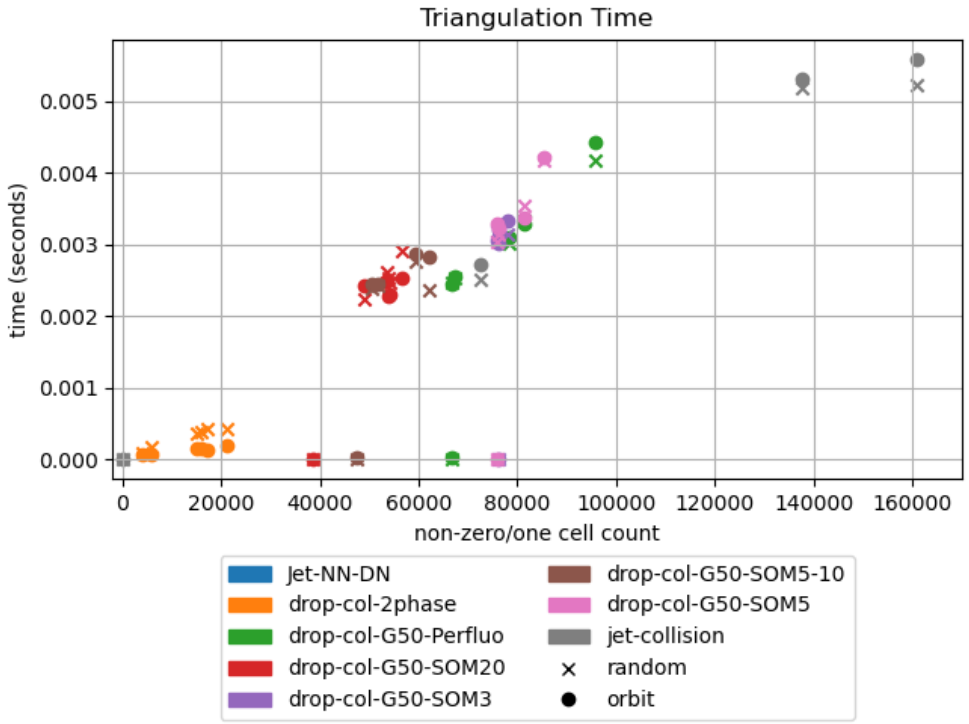


Figure 5.10: Triangulation time when using the CPU.

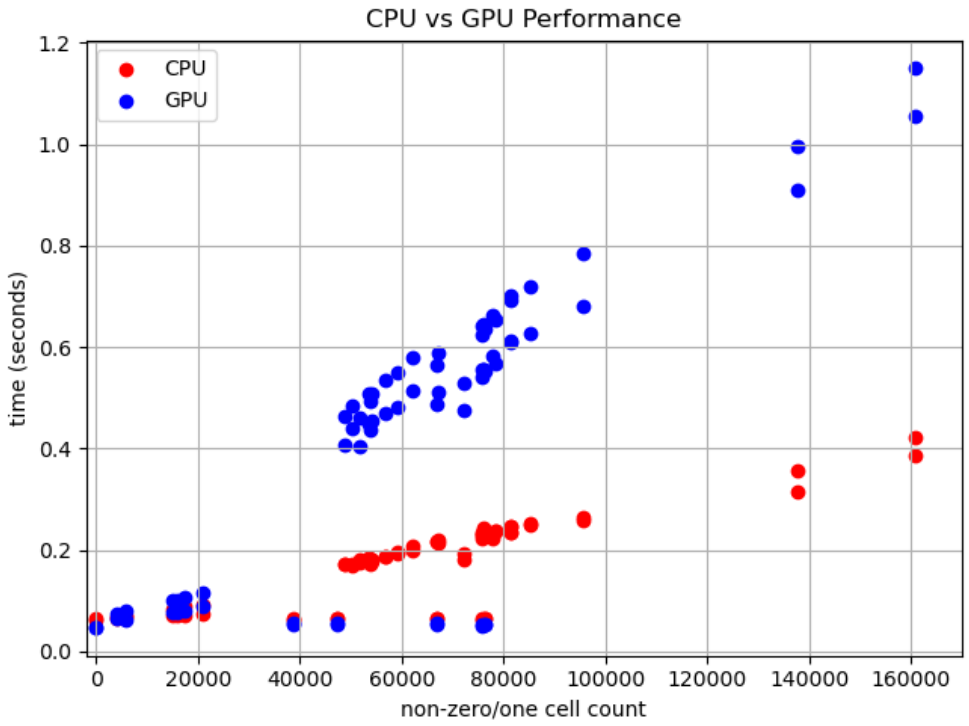


Figure 5.11: Total frame times when rendering on the CPU and GPU.

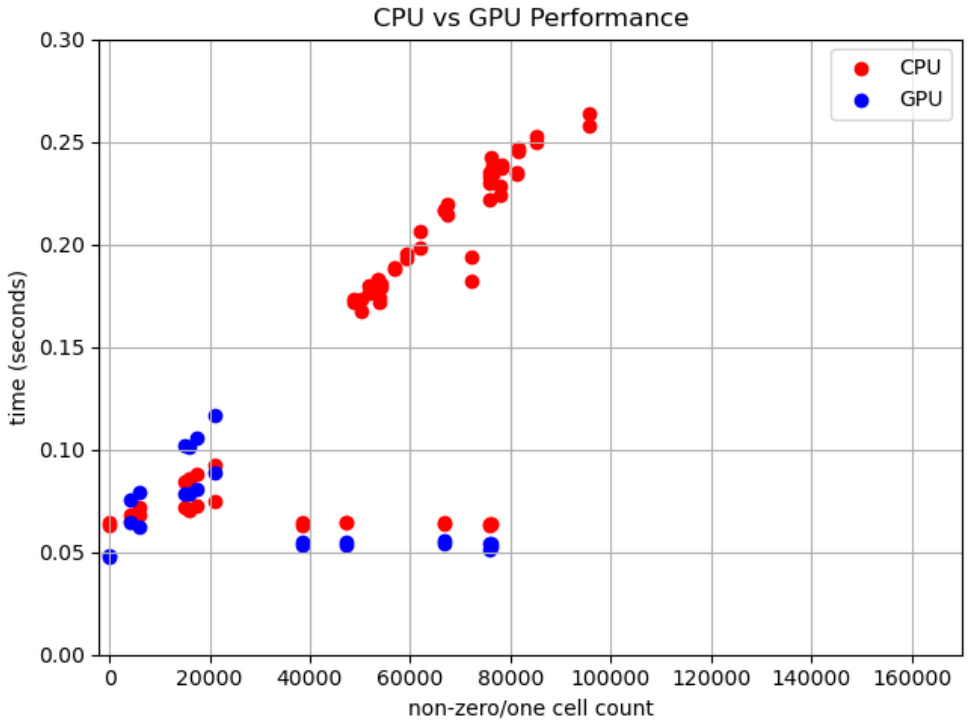


Figure 5.12: Total frame times when rendering on the CPU and GPU (zoomed in).

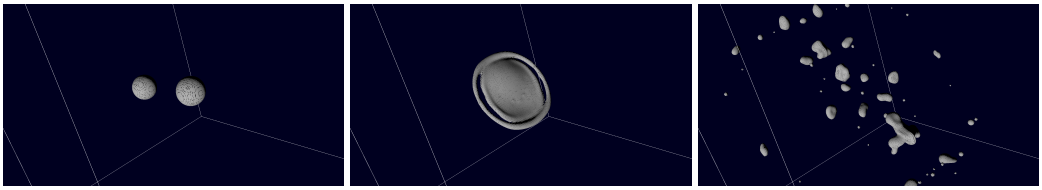


Figure 5.13: The *drop-col-2phase* dataset in time step 0, 200 and 500

A reason why the *Jet-NN-DN* dataset requires more time than the other datasets to serialize and write to disk could be that theVOF function in the original dataset is less spatially coherent. This is because the compression rate of this dataset was bad. The large file size of the compressed dataset would be explained by little spatial coherence in the data because this forces the hierarchical data structure to have a lot of nodes. This large file size then explains the long file writing process: Creating and writing a larger file takes more time.

The reason why the proposed compression algorithm achieves worse compression rates than zip and zlib may be related to the hierarchical data structure. While the data structure can theoretically exploit the spatial coherence of VOF datasets, it may not be able to do so efficiently in practice. This may be because the different regions in the dataset containing different VOF values may not line up with the borders of the nodes in the hierarchical data structure. This leads to the data structure having more nodes, which requires more space. Additionally, the requirement for the files to be partially readable inflates the file size. This is because this requirement necessitates the use of pointers in the file such that it is possible to find the children of a node without reading the entire file. Those pointers take up a significant amount of space: For every node except the root node a

pointer is present in the file which points from the parent of the node to the node. Each pointer is 32 bits large, which is the same as a VOF function value, provided single precision is used. Considering additional data such as flag bytes this means that in a two-phase dataset, less than half of the file size is used for storing the actual VOF function values. This is a bit better for three-phase datasets because each node can store up to two VOF function values. However, even here at least a third of the total file size is taken up by the pointers and other data required to reconstruct the data structure. When dropping the requirement to be able to read files partially, it would be possible to drop the pointers and instead store the data structure in depth-first order. This way, it can still be reconstructed but the overhead from the pointers would be gone.

The fact that the compressed file size grows with a growing amount of non-zero/one cells is most likely related to the spatial coherency of the data. A large number of non-zero/one cells indicates a low spatial coherency. Thus, the algorithm cannot compress the data as efficiently and the resulting files are larger.

5.3.2 Visualization

The time rendering time tends to have an upper bound that depends approximately linearly on the amount of non-zero/one cells because all involved algorithms visits each of those cells at most once per VOF function. This is also why the calculation times of the LOD system, the PLIC calculation, the copying of the data and the triangulation of the PLIC faces behave similarly. The outliers near the x-axis are cases where the rendering took very little time. This happens when there are very few or even no cells at all which contain more than two fluid phases. In those cases, the LOD algorithm is very rarely or not at all forced to descend through the full hierarchical data structure. This effect can be seen when comparing various time steps of the *drop-col-G50-SOM5* dataset as shown in Figure 5.15. The first time step has no cells containing three fluid phases as seen in Figure 5.14. This results in a short rendering time. There are few cases where rendering time lies between the upper bound and the ouliers near the bottom due to the nature of the datasets and the way the LOD algorithm works. As mentioned earlier, the LOD algorithm cannot simplify areas where three phases meet. The datasets tested contained either a lot of those areas or none at all which leads to the observed behavior. The reason why the graph showing the calculation time for the LOD system has more data points between the upper bound and the outliers near the x-axis compared to the PLIC calculation, data copying and triangulation may be related to the way the LOD system works. It descends through the tree towards every non-zero/one cell. However, it does not start at the root node for each of those cells. Instead, it iterates through the entire tree in depth-first order. Many non-zero/one cells are usually in proximity to each other because together they form the phase interface. This leads to the LOD algorithm usually finding multiple non-zero/one cells in quick succession without restarting the search from a node far up in the tree which speeds up the calculation time per cell. The PLIC calculation, data copying and triangulation need to perform the same calculations for every cell and thus do not experience such a spped up.

The reason why the PLIC calculation time as well as the time to copy the vertices to the vertex buffer and the triangulation times behave similar to the time of the LOD system is that those times depend on the number of PLIC cells which need to be calculated. When the LOD system takes little time, it means that it processed a relatively small amount of node which in turn means relatively few PLIC surfaces need to be calculated.

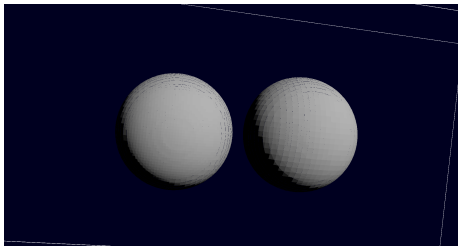


Figure 5.14: First time step of the *drop-col-G50-SOM5* dataset. The two droplets are different fluid phases, but they are completely separate so there are no grid cells where all three phases meet.

The overhead time shown in Figure 5.15 could be larger than expected since it includes the draw call which actually draws the calculated triangles. Additionally, MegaMol may copy the vertex buffer internally, which would also increase this time. The reason why the times for the LOD system are a lot longer in the benchmark that reloads the file every time is that reloading the file causes the hierarchical data structure to be deleted. As such it needs to be rebuilt. Since this is done only when an algorithm requires access to the data and the LOD algorithm is the first algorithm to run, this means that the time for the LOD algorithm is inflated by file reading operations. This behavior was expected.

Overall the behavior of the performance in the GPU benchmark was similar to that of the CPU benchmark. The PLIC calculation was responsible for a large part of the total frame time. The reason why the overhead with the GPU in Figure 5.17 is much lower than with the CPU in Figure 5.15 is because in the CPU benchmark the time of the draw call is part of the overhead time. This is not the case for the GPU implementation where the draw call is its own category. However, there may still be some calculations caused by the PLIC rendering module in MegaMol which are under the overhead time, since the overhead time is still relatively large at around 0.05 seconds. The calculation and upload of the data required for the PLIC calculation does not take much time since this is mostly the computation of the PLIC surface normals from the VOF data.

The GPU implementation is possibly slower than the CPU implementation because the hardware used for the GPU benchmark was worse. Additionally, the PLIC surface calculation may be an algorithm that does not run well on modern GPUs. It contains a lot of loops with a variable amount of iterations which may cause issues with the parallel processing nature of the GPU. The fact that it outperforms the CPU when the entire algorithm takes very little time could be an indication of this. This is the case when there are very few PLIC surfaces to calculate. Then the GPU still has some cores available, and the issue with running different amounts of loop iterations in parallel is less noticeable.

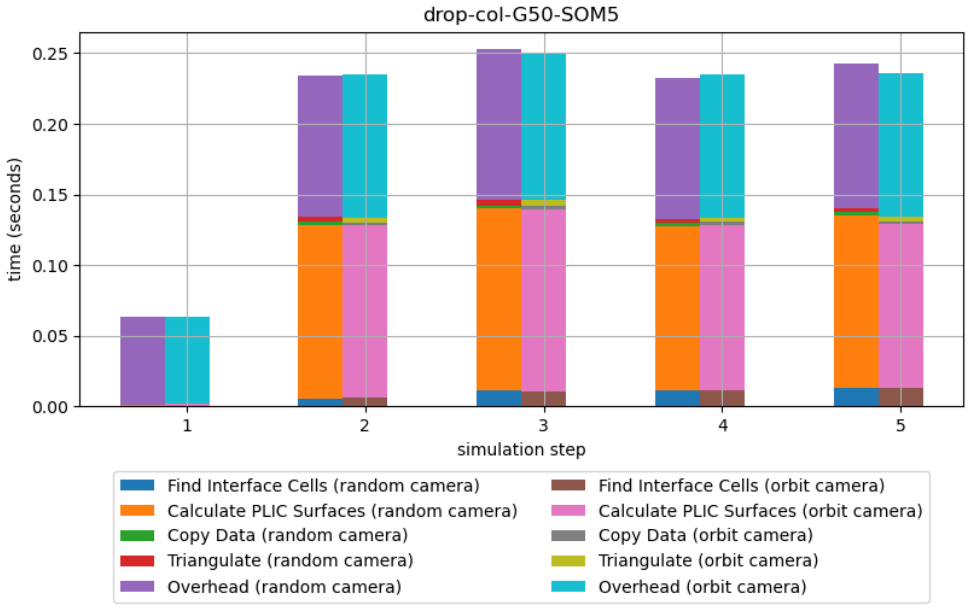


Figure 5.15: Composition of the frame time for the *drop-col-G50-SOM5* dataset.

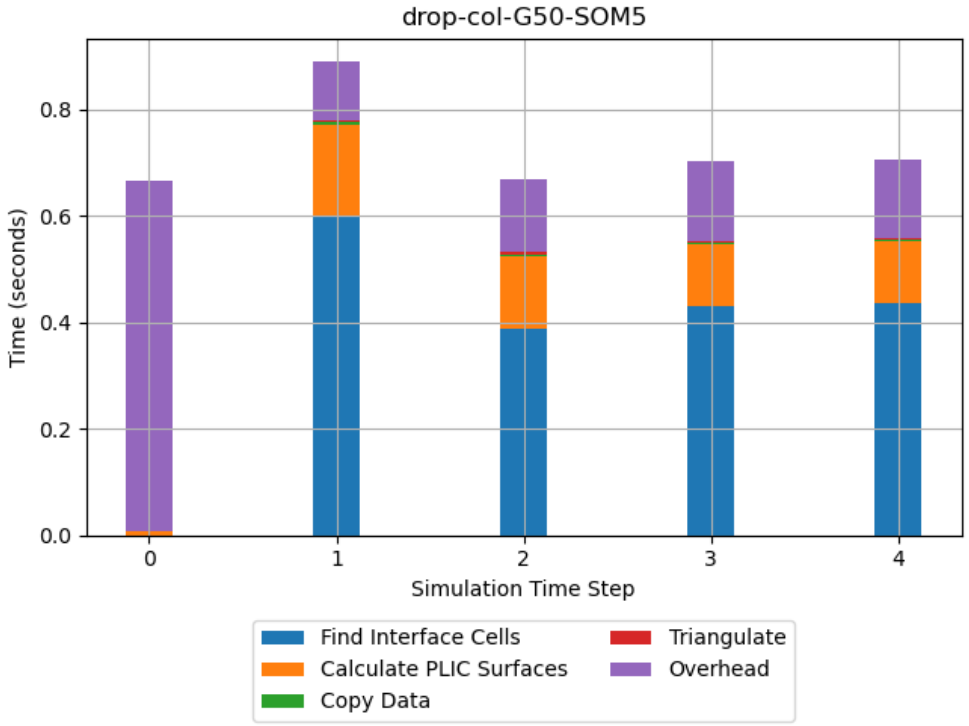


Figure 5.16: Composition of the frame time for the *drop-col-G50-SOM5* dataset when reloading the file in every frame.

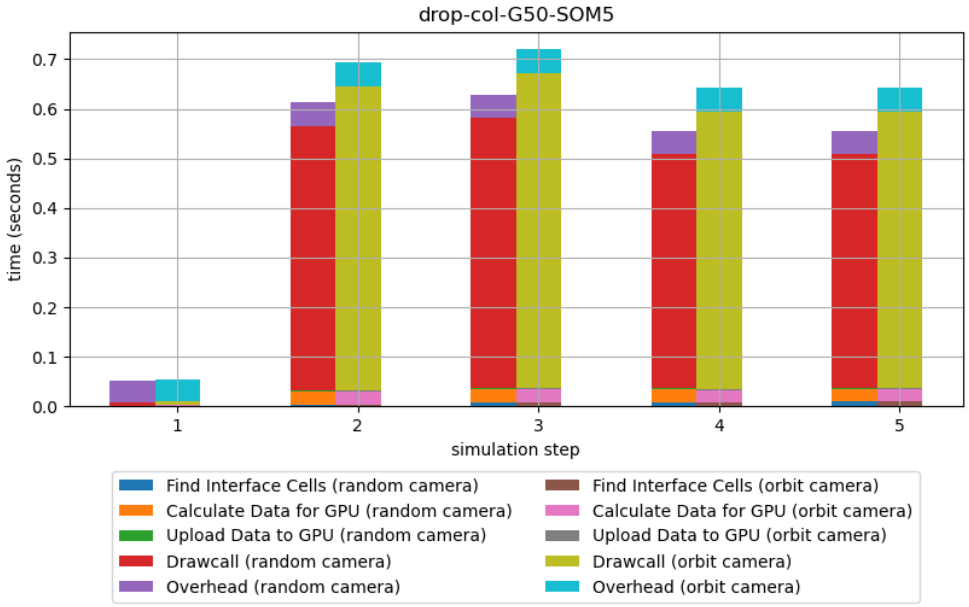


Figure 5.17: Composition of the frame time for the *drop-col-G50-SOM5* dataset calculating the PLIC surfaces on the GPU.

6 Conclusion and Outlook

The goal of this thesis was to design and implement a framework to compress multiphase VOF datasets. A visualization system should then be able to read the compressed files produced by the compression framework and quickly generate a visualization of the VOF data. This would allow generating visualizations of fluid simulations more quickly and efficiently. Thus, analyzing the behavior of the fluid would become easier.

For its simplicity and ability to exploit spatially coherent data, a hierarchical data structure was chosen to store the VOF data. It is able to store multiphase VOF datasets with up to nine fluid phases, which is enough for most current applications. The framework can be used to generate such a hierarchical data structure from a rectilinear grid representation. A custom file format was introduced to be able to write this data structure into a file. It allows partial reading of the file to extract the data at a lower resolution. Reading of the file is done on-demand to minimize the amount of data read.

The partial and on-demand file reading allow quickly generating a visualization from a file in local storage using the visualization system. This system uses the PLIC method to produce an approximation of the fluid surface. Due to the nature of the PLIC method, this surface is discontinuous. However, it is still usable for visualization purposes as described by Karch et al. [KSM+13]. A LOD system is used to accelerate the visualization process at a small cost of visual fidelity. This LOD system directly makes use of the hierarchical data structure, which makes it memory efficient as it does not require any further data structures.

The PLIC calculation is easily implemented for parallel computation. Because of this it was not only implemented for the CPU, but also for the GPU. However, it led to some issues with the Nvidia graphics drivers as they would refuse to compile the shader code.

Both the compression framework and the visualization system were tested to assess their performance. To compare the performance of the compression framework the zip file format and the zlib and LZ4 compression libraries were used. The compression framework was the slowest in all tested cases. It also produced the largest files on average. However, when removing two outliers where the tested datasets had low spatial coherency, the compression rate is around 33% worse than zip and only 11% worse than zlib. The proposed method achieves a 33% better compression rate than LZ4. On top of that, the custom file format allows partial reading of a file which can speed up applications where it is not necessary to access the data at full resolution.

As a result, it is most likely easier to use common data compression methods such as the tested zip file format to compress VOF datasets for simple applications such as file transfers. In most cases, they provide similar or even better compression rates. However, with more sophisticated problems such as the visualization of the data the partial file reading can be an advantage which makes the additional effort and slightly larger file sizes worth it.

Overall, the visualization system is able to quickly produce visualizations of compressed datasets. Both the CPU and GPU implementation of the PLIC calculation algorithm were tested. The benchmark system for the GPU algorithm was a different one with worse performance due to the issues with the Nvidia graphics drivers. This may be the reason why the GPU implementation performed overall worse in the benchmarks. Alternatively, the task of computing PLIC surfaces may not be a good fit for modern GPU architectures.

The entire system can be used to write compressed VOF datasets during the simulation. This then allows quickly rendering a visualization of the simulated fluids. The fast rendering times allow the creation of an entire series of images or even video. This way, fluid simulations can be visualized and analyzed quickly.

Outlook

Due to time constraints the functionalities covered had to be limited. As such, both the compression and the visualization frameworks have room for improvement. In the following some suggestions for enhancements and optimizations of the proposed approach will be discussed.

More options can be explored to optimize the compression rate of the files. Some improvements could be made by introducing new types of nodes in the tree, such as nodes which store an entire region as a compressed grid. This might improve compression performance by reducing the amount of nodes and therefore the overhead from storing many nodes. Furthermore, alternative compression algorithms besides zlib could be explored for compressing the file as a whole.

The compression algorithm could also be improved from a performance standpoint. The current implementation is single threaded. The creation of the tree data structure could potentially be multithreaded by running the creation and simplification of several branches of the tree in parallel. This would significantly cut down the time it takes to generate the data structure.

The visualization framework could also be adapted into an interactive application. This would require improving the performance further, for example by caching some results of the visualization algorithms such as the offset for the plane in each cell or potentially even the entire resulting mesh.

Further work is required to make the GPU implementation work on more platforms. Since the shader code presumably fails to compile due to insufficient memory for local variables it may be required to supply more memory through storage buffers or images. On top of that it may be possible to improve the performance by running the calculations in a compute shader instead of a geometry shader. Additionally, the work required to compute which cells to render could be moved to the GPU. This would require reading the entire file and uploading it onto the GPU. This way, potential benefits from reading the file only partially could be traded for faster computation times when computing the cells to render. It may also be possible to use technologies such as Nvidia GPUDirect Storage to achieve even faster file loading times by bypassing the CPU entirely and loading the file directly from storage onto the GPU.

The framework also only supports visualization of three phases. Theoretically up to nine fluid phases can be represented by the described file format. However, this only applies to the VOF functions themselves. The additional gradients needed for more than three phases cannot be stored

using the shown implementation. As such another future improvement is to support a larger number of phases. This would also necessitate changes in the visualization algorithm as it expects one or two VOF functions, limiting the amount of phases to three.

The framework could also be expanded to allow for additional data, such as flow velocity or pressure. This would possibly require different compression algorithms for the additional data as they are not as spatially coherent as the VOF function. It would however allow for more sophisticated visualizations showing flow direction, speed or pressure.

Bibliography

[AKSV15] J. Alakuijala, E. Kliuchnikov, Z. Szabadka, L. Vandevenne. “Comparison of brotli, deflate, zopfli, lzma, lzham and bzip2 compression algorithms”. In: *Google Inc* (2015), pp. 1–6 (cit. on p. 39).

[BDS+03] K. Bonnell, M. Duchaineau, D. Schikore, B. Hamann, K. Joy. “Material interface reconstruction”. In: *IEEE Transactions on Visualization and Computer Graphics* 9.4 (2003), pp. 500–511. DOI: 10.1109/TVCG.2003.1260744 (cit. on p. 22).

[BGOS06] A. W. Bargteil, T. G. Goktekin, J. F. O’Brien, J. A. Strain. “A semi-Lagrangian contouring method for fluid simulation”. In: *ACM Trans. Graph.* 25.1 (Jan. 2006), pp. 19–38. ISSN: 0730-0301. DOI: 10.1145/1122501.1122503. URL: https://doi.org/10.1145/1122501.1122503 (cit. on p. 22).

[Blo94] J. Bloomenthal. “An Implicit Surface Polygonizer”. In: *Graphics gems*. 1994. URL: https://api.semanticscholar.org/CorpusID:117145464 (cit. on p. 22).

[CHF96] W. Cochran, J. Hart, P. Flynn. “Fractal volume compression”. In: *IEEE Transactions on Visualization and Computer Graphics* 2.4 (1996), pp. 313–322. DOI: 10.1109/2945.556500 (cit. on pp. 15, 21).

[Cla76] J. H. Clark. “Hierarchical geometric models for visible surface algorithms”. In: *Commun. ACM* 19.10 (Oct. 1976), pp. 547–554. ISSN: 0001-0782. DOI: 10.1145/360349.360354. URL: https://doi.org/10.1145/360349.360354 (cit. on pp. 30, 36).

[EEG+16] K. Eisenschmidt, M. Ertl, H. Gomaa, C. Kieffer-Roth, C. Meister, P. Rauschenberger, M. Reitzle, K. Schlottke, B. Weigand. “Direct numerical simulations for multiphase flows: An overview of the multiphase code FS3D”. In: *Applied Mathematics and Computation* 272 (2016). Recent Advances in Numerical Methods for Hyperbolic Partial Differential Equations, pp. 508–517. ISSN: 0096-3003. DOI: https://doi.org/10.1016/j.amc.2015.05.095. URL: https://www.sciencedirect.com/science/article/pii/S0096300315007195 (cit. on pp. 23, 39).

[ELC+12] M. Edmunds, R. S. Laramee, G. Chen, N. Max, E. Zhang, C. Ware. “Surface-based flow visualization”. In: *Computers & Graphics* 36.8 (2012). Graphics Interaction Virtual Environments and Applications 2012, pp. 974–990. ISSN: 0097-8493. DOI: https://doi.org/10.1016/j.cag.2012.07.006. URL: https://www.sciencedirect.com/science/article/pii/S0097849312001355 (cit. on p. 23).

[Ert19] M. Ertl. “Direct numerical investigations of non-newtonian drop oscillations and jet breakup”. In: (2019). DOI: 10.18419/opus-10764. URL: http://dx.doi.org/10.18419/opus-10764 (cit. on pp. 39, 40).

[EV08] J. Eggers, E. Villermaux. “Physics of liquid jets”. In: *Reports on Progress in Physics* 71.3 (Feb. 2008), p. 036601. DOI: 10.1088/0034-4885/71/3/036601. URL: https://dx.doi.org/10.1088/0034-4885/71/3/036601 (cit. on p. 15).

[EW17] M. Ertl, B. Weigand. "ANALYSIS METHODS FOR DIRECT NUMERICAL SIMULATIONS OF PRIMARY BREAKUP OF SHEAR-THINNING LIQUID JETS". In: *Atomization and Sprays* 27.4 (2017), pp. 303–317. ISSN: 1044-5110. DOI: 10.1615/AtomizSpr.2017017448. URL: https://doi.org/10.1615/AtomizSpr.2017017448 (cit. on pp. 39, 40).

[FY94] J. E. Fowler, R. Yagel. "Lossless compression of volume data". In: *Proceedings of the 1994 Symposium on Volume Visualization*. VVS '94. Tysons Corner, Virginia, USA: Association for Computing Machinery, 1994, pp. 43–50. ISBN: 0897917413. DOI: 10.1145/197938.197961. URL: https://doi.org/10.1145/197938.197961 (cit. on pp. 15, 21).

[GBB+19] P. Gralka, M. Becher, M. Braun, F. Frieß, C. Müller, T. Rau, K. Schatz, C. Schulz, M. Krone, G. Reina, et al. "MegaMol–a comprehensive prototyping framework for visualizations". In: *The European Physical Journal Special Topics* 227 (2019), pp. 1817–1829. DOI: https://doi.org/10.1140/epjst/e2019-800167-5 (cit. on p. 37).

[HLM+21] J. Han, B. Li, D. Mukherjee, C.-H. Chiang, A. Grange, C. Chen, H. Su, S. Parker, S. Deng, U. Joshi, Y. Chen, Y. Wang, P. Wilkins, Y. Xu, J. Bankoski. "A Technical Overview of AV1". In: *Proceedings of the IEEE* 109.9 (2021), pp. 1435–1462. DOI: 10.1109/JPROC.2021.3058584 (cit. on p. 15).

[HN81] C. Hirt, B. Nichols. "Volume of fluid (VOF) method for the dynamics of free boundaries". In: *Journal of Computational Physics* 39.1 (1981), pp. 201–225. ISSN: 0021-9991. DOI: https://doi.org/10.1016/0021-9991(81)90145-5. URL: https://www.sciencedirect.com/science/article/pii/0021999181901455 (cit. on p. 17).

[KC16] R. Kaur, P. Choudhary. "A review of image compression techniques". In: *Int. J. Comput. Appl* 142.1 (May 2016), pp. 8–11. DOI: 10.5120/ijca2016909658 (cit. on p. 15).

[KSM+13] G. K. Karch, F. Sadlo, C. Meister, P. Rauschenberger, K. Eisenschmidt, B. Weigand, T. Ertl. "Visualization of piecewise linear interface calculation". In: *2013 IEEE Pacific Visualization Symposium (PacificVis)*. 2013, pp. 121–128. DOI: 10.1109/PacificVis.2013.6596136 (cit. on pp. 23, 57).

[Mus13] K. Museth. "VDB: High-resolution sparse volumes with dynamic topology". In: *ACM Trans. Graph.* 32.3 (July 2013). ISSN: 0730-0301. DOI: 10.1145/2487228.2487235. URL: https://doi.org/10.1145/2487228.2487235 (cit. on p. 21).

[NW76] W. F. Noh, P. Woodward. "SLIC (Simple Line Interface Calculation)". In: *Proceedings of the Fifth International Conference on Numerical Methods in Fluid Dynamics June 28 – July 2, 1976 Twente University, Enschede*. Ed. by A. I. van de Vooren, P. J. Zandbergen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1976, pp. 330–340. ISBN: 978-3-540-37548-7. DOI: 10.1007/3-540-08004-X_336 (cit. on p. 22).

[OCHJ12] H. Obermaier, F. Chen, H. Hagen, K. I. Joy. "Visualization of material interface stability". In: *2012 IEEE Pacific Visualization Symposium*. 2012, pp. 225–232. DOI: 10.1109/PacificVis.2012.6183595 (cit. on p. 23).

[PS24] J. Potyka, K. Schulte. “A volume of fluid method for three dimensional direct numerical simulations of immiscible droplet collisions”. In: *International Journal of Multiphase Flow* 170 (2024), p. 104654. ISSN: 0301-9322. DOI: https://doi.org/10.1016/j.ijmultiphaseflow.2023.104654. URL: https://www.sciencedirect.com/science/article/pii/S0301932223002744 (cit. on pp. 39, 40).

[PSP23] J. Potyka, K. Schulte, C. Planchette. “Liquid distribution after head-on separation of two colliding immiscible liquid droplets”. In: *Physics of Fluids* 35.10 (Oct. 2023), p. 102125. ISSN: 1070-6631. DOI: 10.1063/5.0168080. URL: https://doi.org/10.1063/5.0168080 (cit. on pp. 39, 40).

[PSW+24] J. Potyka, J. Stober, J. Wurst, M. Ibach, J. Steigerwald, B. Weigand, K. Schulte. “Towards DNS of Droplet-Jet Collisions of Immiscible Liquids with FS3D”. In: *High Performance Computing in Science and Engineering ’22*. Ed. by W. E. Nagel, D. H. Kröner, M. M. Resch. Cham: Springer Nature Switzerland, 2024, pp. 197–212. ISBN: 978-3-031-46870-4. DOI: 10.1007/978-3-031-46870-4_14. URL: https://doi.org/10.1007/978-3-031-46870-4_14 (cit. on pp. 39, 40).

[RAG24] G. Roelofs, M. Adler, J.-l. Gailly. *zlib Home Site*. http://zlib.net/. [Online; accessed 15-05-2024]. 1996 – 2024 (cit. on pp. 25, 29, 39).

[RK98] W. J. Rider, D. B. Kothe. “Reconstructing Volume Tracking”. In: *Journal of Computational Physics* 141.2 (1998), pp. 112–152. ISSN: 0021-9991. DOI: https://doi.org/10.1006/jcph.1998.5906. URL: https://www.sciencedirect.com/science/article/pii/S002199919895906X (cit. on pp. 17, 23).

[Roe08] G. Roelofs. *Info-ZIP Home Page*. https://infozip.sourceforge.net/. [Online; accessed 15-05-2024]. 1995 – 2008 (cit. on p. 39).

[SK02] D. Saupe, J.-P. Kuska. “Compression of Isosurfaces for Structured Volumes”. In: *Proceedings. First International Symposium on 3D Data Processing Visualization and Transmission*. IEEE Comput. Soc, 2002, pp. 384–390. ISBN: 0-7695-1521-5. DOI: 10.1109/TDPVT.2002.1024088 (cit. on p. 21).

[SML06] W. Schroeder, K. Martin, B. Lorensen. *The Visualization Toolkit (4th ed.)* Kitware, 2006. ISBN: 978-1-930934-19-1 (cit. on p. 21).

[YH18] G. Yadigaroglu, G. F. Hewitt. *Introduction to Multiphase Flow: Basic Concepts, Applications and Modelling*. Springer, Oct. 2018. ISBN: 978-3-319-58717-2. DOI: 10.1007/978-3-319-58718-9 (cit. on p. 15).

All links were last followed on May 23, 2024.

A Appendix

Zusammenfassung in deutscher Sprache

Um Volume of Fluid (VOF) Datensätze zu visualisieren ist es notwendig, mit den großen Dateien der VOF Datensätze umzugehen, was langsam ist. Diese Arbeit zeigt eine Methode für die Komprimierung von VOF Datensätzen mittels einer hierarchischen Datenstruktur und einen Level of Detail (LOD) Ansatz um die VOF Daten zu visualisieren. Diese Datenstruktur ist eine zerstreute Repräsentation des VOF Feldes. Damit nutzt sie die räumliche Kohärenz der VOF Datensätze um hohe Kompressionsraten zu erzielen. Ein eigenes Dateiformat wird vorgestellt, um diese Datenstruktur abzuspeichern. Dieses Dateiformat ist so entworfen, dass eine Datei auch nur teilweise gelesen werden kann, um eine niedriger aufgelöste Version der Daten zu bekommen. Der LOD Ansatz wird benutzt, um schnell Bilder von Daten zu erstellen, die mit diesem Dateiformat komprimiert wurden. Piecewise Linear Interface Calculation (PLIC) wird benutzt, um die Oberflächen der Flüssigkeiten aus den VOF Daten zu extrahieren. Dieses System nutzt Parallelisierung, um die Rechenzeit zu verringern. Die hierarchische Datenstruktur der Komprimierungsmethode wird außerdem benutzt, um die verschiedenen Detailstufen des LOD Ansatzes zu erzeugen. Zusätzlich zur Implementierung dieser Methoden wird eine große Menge an Benchmarks durchgeführt, um sowohl den Kompressionsalgorithmus als auch das Visualisierungssystem zu testen. Die Performance des Kompressionsalgorithmus wird mit der von den häufig anzutreffenden, generischen Kompressionsalgorithmen LZ4, zlib und dem zip Dateiformat verglichen. Sowohl die Rechenzeit der Kompressionsalgorithmen als auch deren Kompressionsraten werden betrachtet. Besonders in Fällen, in denen ein Datensatz hohe räumliche Kohärenz aufweist, kann die vorgestellte Methode eine bessere Kompressionsrate als die verglichenen Kompressionsalgorithmen erreichen. Bei dem Visualisierungssystem wurde die Rechenzeit für das Rendering gemessen. Die Berechnung der PLIC Oberflächen im Visualisierungssystem wurde sowohl auf der CPU als auch auf der GPU getestet. Beide Implementierungen können schnelle Renderzeiten erreichen.

Amount of Bytes	Use
1 byte	flag byte
4 bytes	amount of children along x-axis c_x
4 bytes	amount of children along y-axis c_y
4 bytes	amount of children along z-axis c_z
4 bytes	amount of cells along x-axis d_x
4 bytes	amount of cells along y-axis d_y
4 bytes	amount of cells along z-axis d_z
4 bytes	size of the children of the root node, in cells
4 bytes	amount of VOF functions
$4d_x$ bytes	cell x-coordinates
$4d_y$ bytes	cell y-coordinates
$4d_z$ bytes	cell z-coordinates
$4(c_x \cdot c_y \cdot c_z)$ bytes	pointers to children

Table A.1: Byte layout of the root node from top to bottom.

Amount of Bytes	Use
1 byte	flag byte
1 byte	VOF function flag byte (which functions are stored)
$4 \cdot n$ bytes	VOF function values (only for functions indicated by flag byte)
12 bytes	gradient for PLIC interface reconstruction (only if indicated by flag byte)
32 bytes	pointers to children

Table A.2: Byte layout of an inner node from top to bottom.

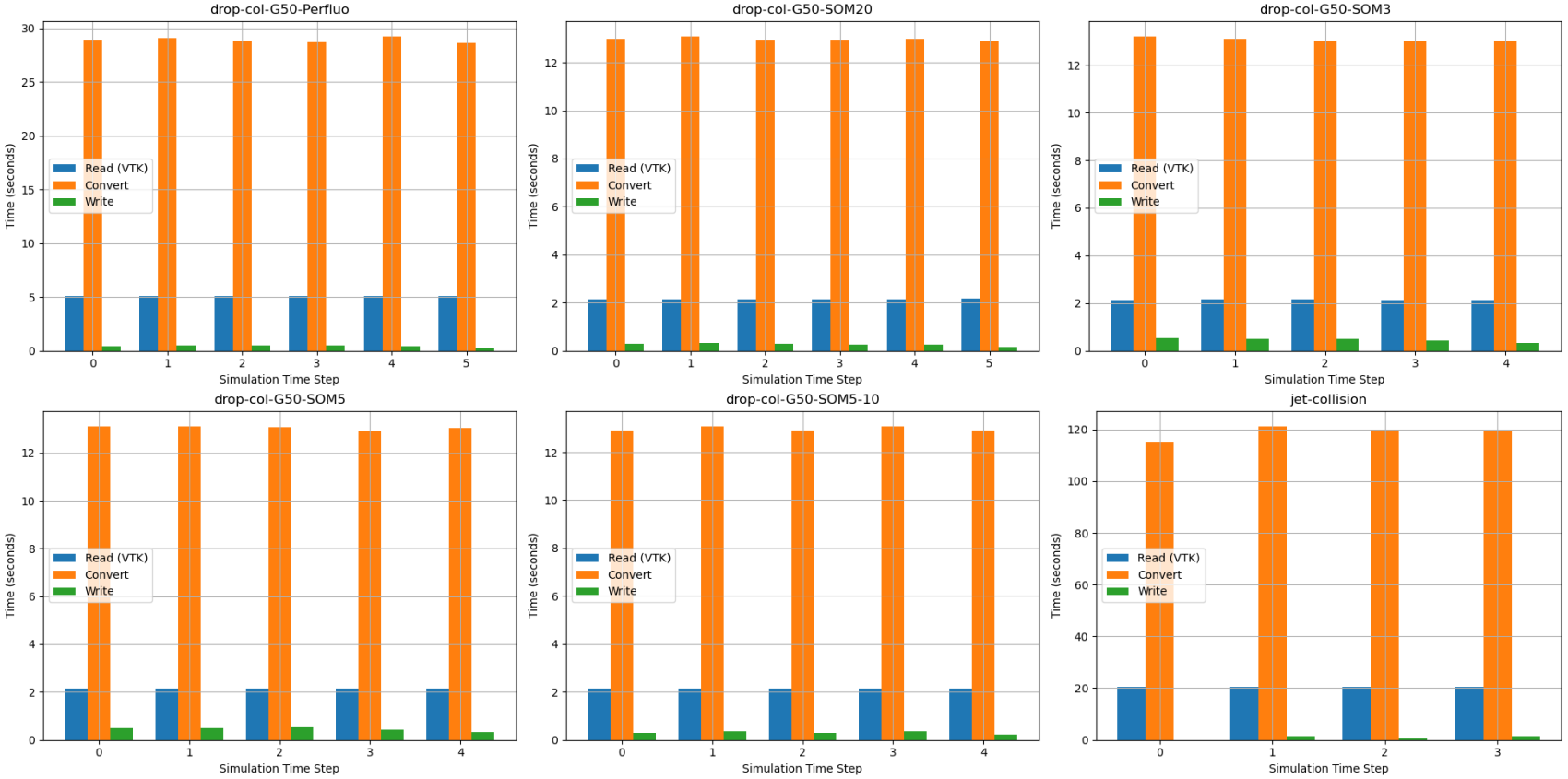


Figure A.1: Compression times for different datasets using the proposed algorithm

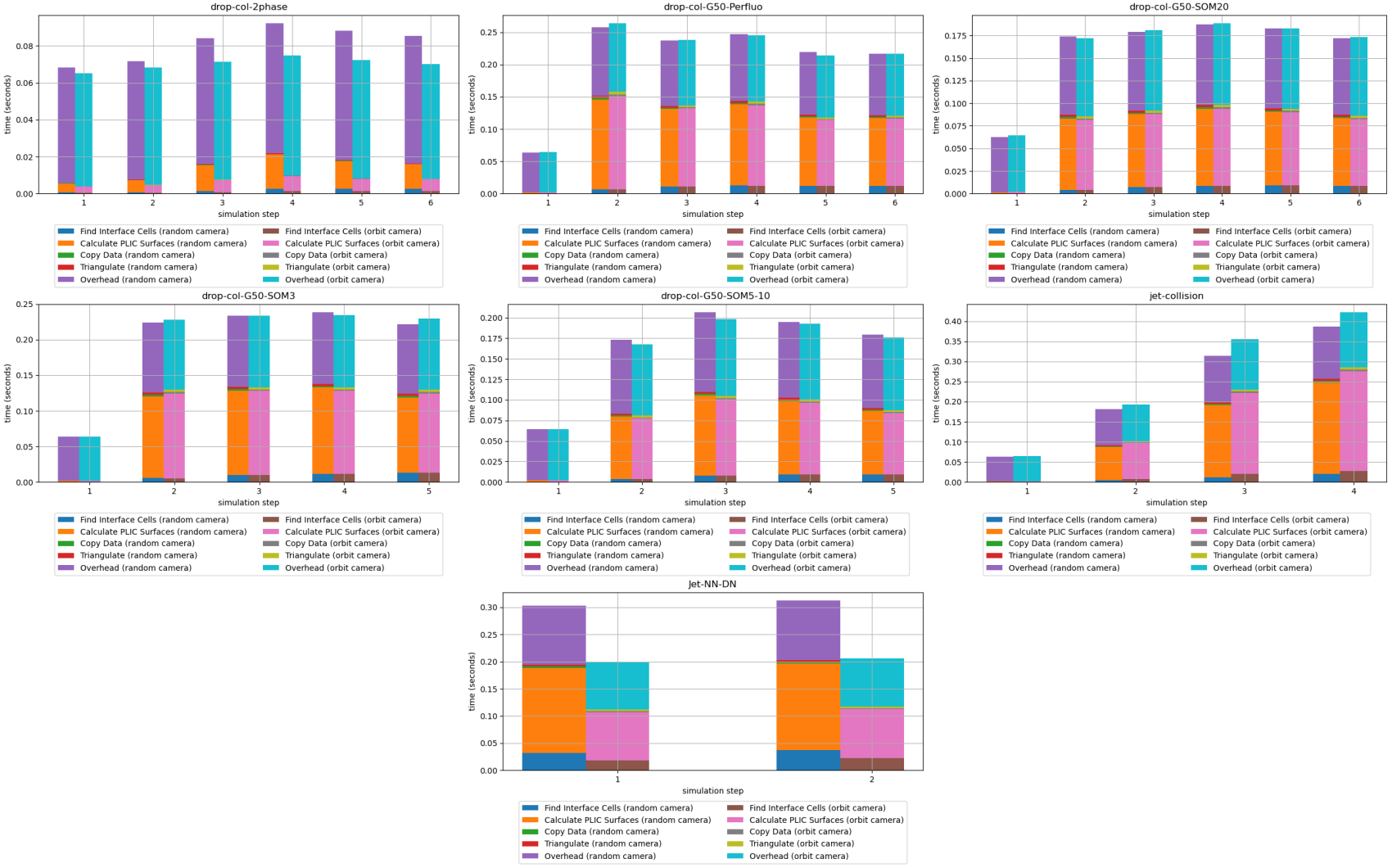


Figure A.2: Frame time composition in the CPU benchmark

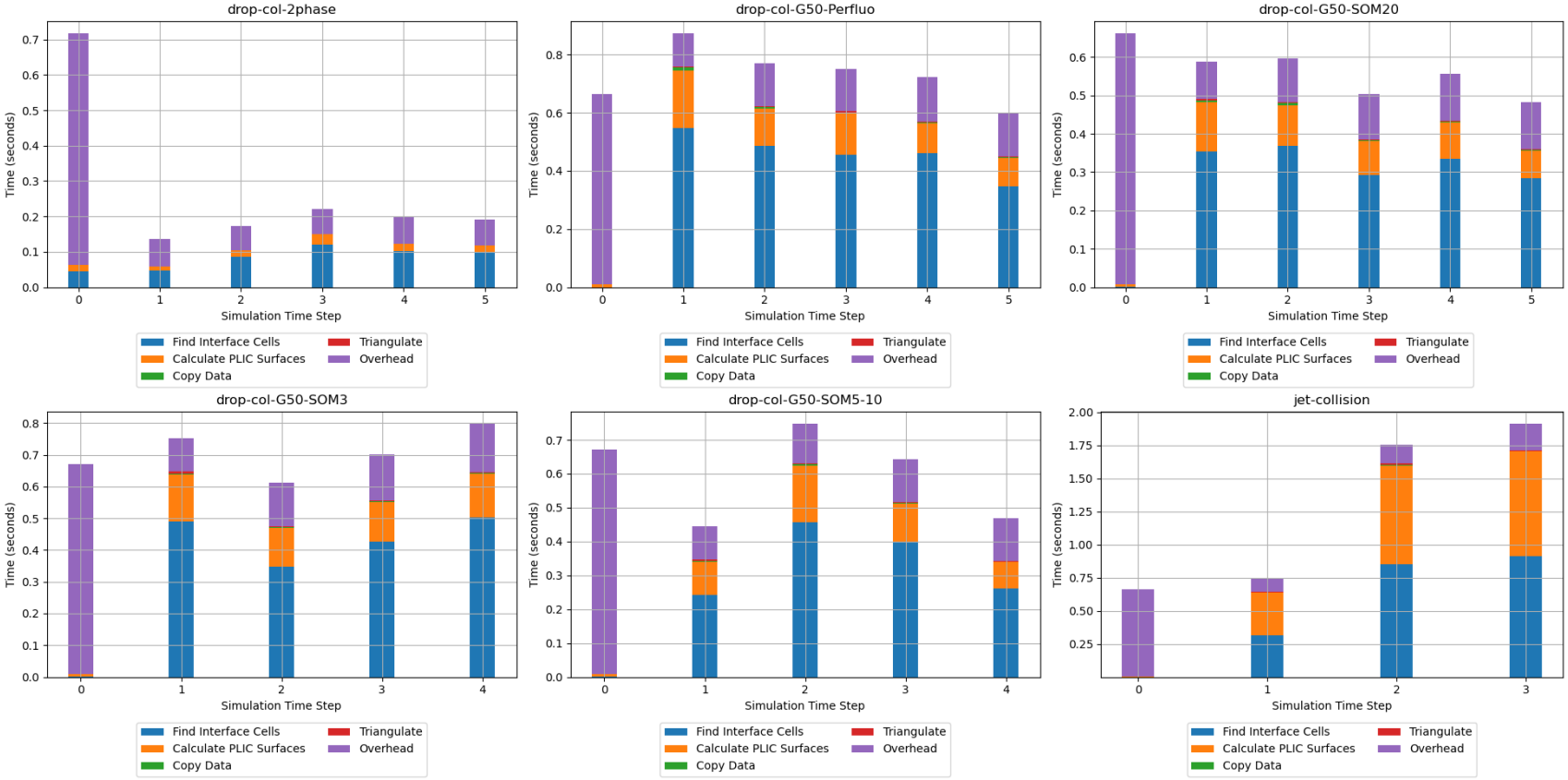


Figure A.3: Frame time composition when reloading the file every frame

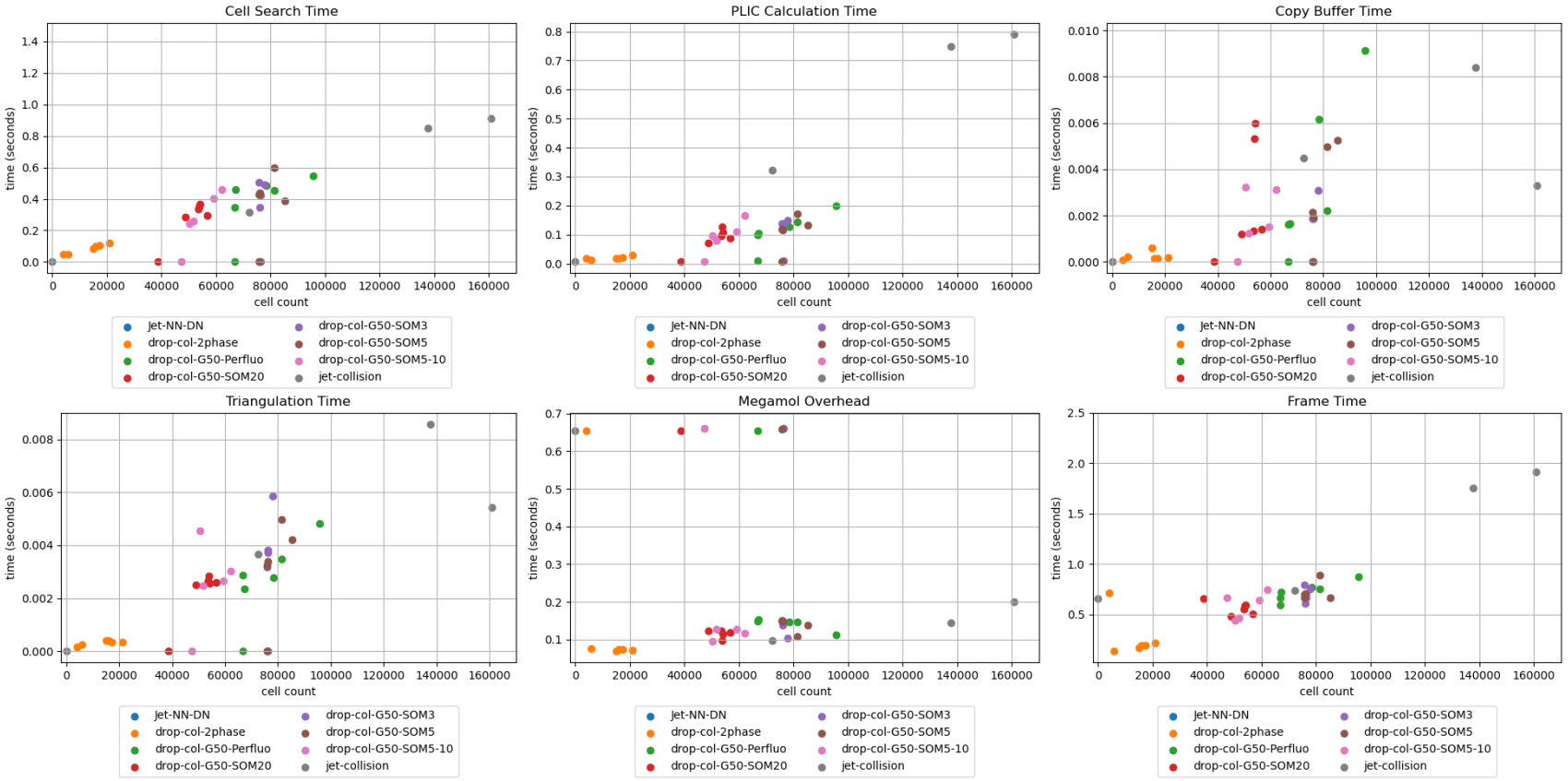


Figure A.4: Calculation times for different algorithms in the CPU benchmark when reloading the file every frame

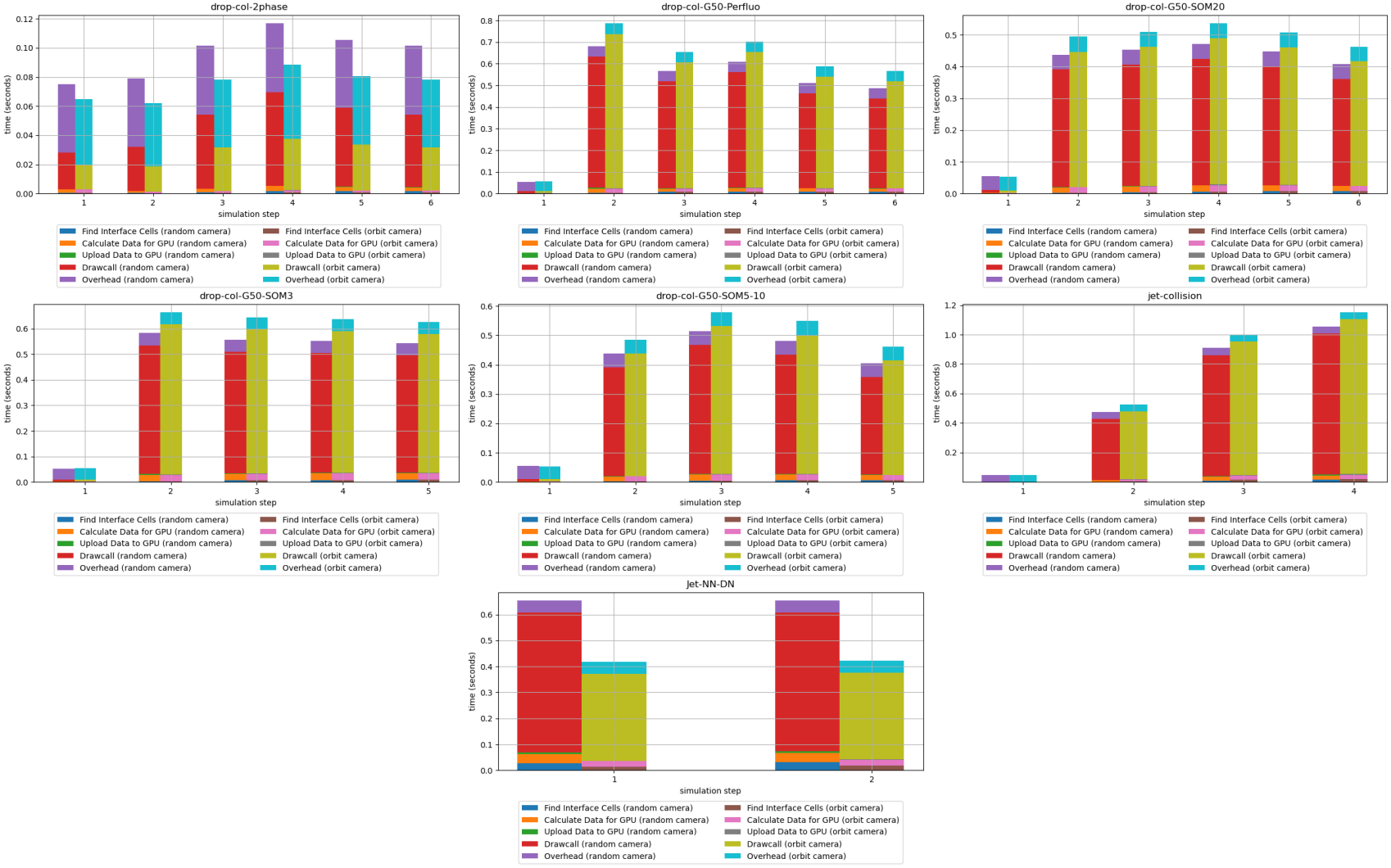


Figure A.5: Frame time composition in the GPU benchmark

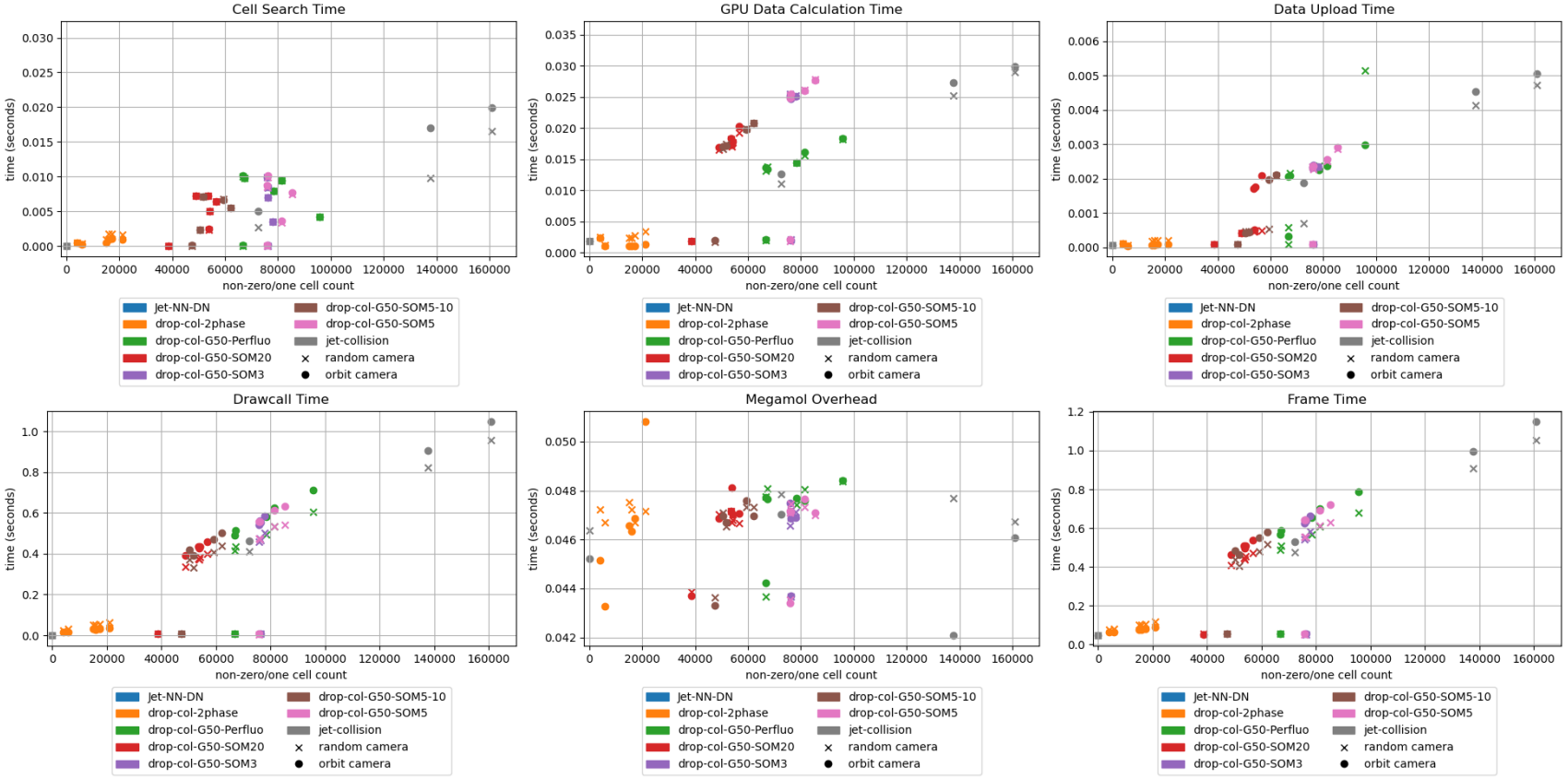


Figure A.6: Calculation times for different algorithms in the GPU benchmark

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature