UNIVERSITY OF BERGEN
DEPARTMENT OF INFORMATICS

Liquid Conductor: Animated Transitions Across Dimensions

Author: Tines Valen
Supervisors: Stefan Bruckner

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

June, 2023

Abstract

This thesis explores the power of visualization in conveying complex data through intuitive visual representations. It investigates the effectiveness of animated transitions in understanding correlations between different visualizations of the same data. While volume is generally considered a less effective encoding of magnitude in visualization, 3D becomes crucial when shape recognition is required. However, when comparing the volume of 3D objects, it is necessary to incorporate the objects themselves in the visualization. The Liquid Conductor program is introduced to address this challenge, which seamlessly transforms 3D objects into bar charts using animated transitions. Leveraging fluid simulation, the program effectively communicates volume differences by filling each object with liquid and pouring it into cylinders, acting as bars within the bar chart visualization.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Prof. Stefan Bruckner, and also Dr. Thomas Trautner, for their invaluable guidance and unwavering support throughout this research. Their expertise and insightful feedback have played a crucial role in shaping this work. My heartfelt thanks go to my family, friends, and my girlfriend for their constant encouragement and belief in my abilities.

Tines Valen
Monday 19th June, 2023

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Creating informative and engaging data visualizations often requires complex techniques to effectively communicate data trends and patterns. One of the challenges in developing effective data visualizations is ensuring that users can easily understand the relationships between different representations of the same data. Animated transitions between visualizations can be a powerful tool in this regard.

Animated transitions between two visualizations can effectively convey complex information in an easy-to-understand format. Animated transitions in 2D are well researched and shown to be helpful and expressive [20]. They can help users retain their sense of context and understand how the data has changed. By smoothly transitioning between two visualizations, users can more easily identify changes and patterns in the data and better understand how different data points are related. Additionally, animated transitions can highlight specific data features, such as changes in magnitude or position, which can help users interpret the presented information more effectively.

Overall, animated transitions can be a valuable technique for enhancing the usability and effectiveness of data visualizations by improving context retention and supporting more accurate and insightful data analysis. Several reasonable solutions exist to create animated transitions, such as D3.js [6], Tableau [48], etc. However, these products are intended to be used by professionals and are often quite limited in their 3D visualization options.

Animated transitions in 2D already suffer from problems such as occlusion [20] but can be prevented by techniques such as staggering [13, 25]. This problem becomes even more prevalent when we introduce the third dimension, this problem becomes even more prevalent, as 3D

visualizations introduce several inherent issues in 3D environments, such as occlusion, navigation, and illumination [14, 11]. Due to this, creating such transitions in 3D quickly becomes a complex task and is often time-consuming, and is generally made on a per-dataset basis.

Traditional approaches to creating 3D animated transitions often require careful manipulation of keyframes to achieve a smooth and convincing transition [45, 37]. This requires expertise in 3D computer graphics software with animation features, such as Blender [15] or Maya [23], in addition to knowledge of the subject matter of the data set.

1.1 Problem Description

Comparing volumes of different objects is not easy. With physical objects, one can use Archimedes' principle [27], which states that when an object is immersed in a fluid, it experiences an upward buoyant force equal to the weight of the fluid it displaces. This principle allows us to easily compare volumes of irregular objects because the buoyant force is directly related to the volume of the fluid displaced. By measuring the buoyant force and knowing the density of the liquid, we can determine the object's volume without needing complex calculations or measurements of its shape.

There are several ways to measure the volume of a 3D virtual object [53], with Computer Assisted Drawing (CAD) and 3D modeling software often having features that calculate a mesh's volume and output the result. These are accurate and quick methods, but if one wants to compare the relative volume differences of two or more 3D objects, being shown pure numbers is not an engaging way to communicate this.

We can not simply look at the 3D objects and intuit their volume relative to each other, as visual volume estimation is quite difficult for human eyes. This is further pointed out in that 3D size (volume) is often considered the least effective way to encode magnitude in data visualizations [36]. Although volume is a physical attribute that can be easily quantified and visualized, it is difficult for human eyes to read accurate values out of just volume. In addition, depth perception can also be a contributing factor, as the perceived size of an object can change depending on its position relative to the viewer. Therefore, other visual cues, such as color, position, or shape, are often preferred for encoding magnitude in data visualizations.

This thesis uses visualization techniques such as animated transitions to better communicate the relative volume differences in 3D objects. by transitioning from the mesh representation to a representation using a more effective channel for encoding magnitude, such as a bar chart,

we can communicate the volume differences intuitively. As well as convey the relationship between the representations. Bar charts use the *Position on a common scale* channel, considered the best channel for encoding magnitude [36]. However, bar charts are boring and forgettable [28].

To combat this, we experimented with fluid simulation as the transition method. By converting the mesh volume into fluid particles in a fluid solver and pouring the fluid into a bar in a bar chart, we make the entire visualization more memorable and engaging and give users an intuitive and easily understood transition type.

With this thesis, we aim to find solutions to the following questions:

- How can we communicate relative volume differences expressively and engagingly using animated transitions?
- How effective is a fluid simulation as a transition means?
- How can we transition from a 3D to a 2D representation without losing context?

In the next chapters, we first discuss the background work on visualization and some more specific topics, such as animation, communicative visualization, and fluid simulation. In chapter 3, we discuss the most relevant literature in more detail. Later, our solution is explained in chapter 4. Key components of our solution are presented, with explanations of their role in the entire system and a high-level description of how they work. Chapter 5 goes over our specific implementation of the components presented in chapter 4. In chapter 6, we first present use cases where the Liquid Conductor is applied. The last two chapters are the discussion and conclusion.

Chapter 2

Background

The book *Visualization Analysis & Design* by Munzner [36], defines visualization the following way:

> Computer-based visualization systems provide visual representations of datasets designed to help people carry out tasks more effectively. Visualization is suitable when there is a need to augment human capabilities rather than replace people with computational decision-making methods. The design space of possible vis idioms is huge, and includes the considerations of both how to create and how to interact with visual representations. Vis design is full of trade-offs, and most possibilities in the design space are ineffective for a particular task, so validating the effectiveness of a design is both necessary and difficult. Vis designers must take into account three very different kinds of resource limitations: those of computers, of humans, and of displays. Vis usage can be analyzed in terms of why the user needs it, what data is shown, and how the idiom is designed.

Visualization solutions are typically designed to address the specific needs of different data types, user groups, and tasks. These require tailored visualizations to convey information effectively. Visualization solutions aim to optimize the understanding, analysis, and interpretation of data for users in their respective domains by targeting these specific aspects.

Researchers have developed abstractions that categorize visualization tasks based on their goals and purposes to abstract from domain-specific tasks to more generalized visualization tasks. Shneiderman [42] proposed one widely referenced taxonomy that categorizes visualization tasks based on the data types involved, such as 1D, 2D, 3D, temporal, and multi-dimensional

data. This taxonomy emphasizes the importance of understanding the specific characteristics of the visualized data and tailoring the visualization techniques accordingly.

In her book, Munzner [36] defined a task abstraction that introduces three levels of actions that define user goals: Analyze, Search, and Query. The analysis level is separated into goals of data *consumption* and *production*, meaning whether the user wants to use existing data or create new data. This thesis seeks to focus on the presentation and enjoyment parts of visualization. Within the data consumption goal, Munzner identified three different goals: *Discovering* new knowledge within the data, *Presenting* the data to communicate the information within, and visualizing the data for *Enjoyment* purposes.

2.1 Presentational and Communicative Visualization

Presentational visualization tasks emphasize the visual design and aesthetics of the visualizations to create visually compelling and engaging representations. Communicative visualization tasks focus on conveying complex information clearly and understandably, making it accessible to a broader audience.

Kosara [28] states the importance of memorability and engagement when presenting data. The goal when presenting data is to create a visualization that viewers remember. He states that simple charts, such as bar charts, scatter plots, etc., while good for analysis and exploratory settings, are generally not memorable enough for presentation.

Moere and Purchase [32] state that, in visualization, the three important aspects are utility, soundness, and aesthetics. Utility refers to the efficacy and efficiency of a visualization technique. Soundness refers to how generalizable the technique is, meaning how easily it is adapted to different data sets and tasks. Moere and Purchase emphasize the significance of the third aspect, which is traditionally focused less on. They argue that more focus on aesthetically pleasing visualizations can increase engagement, leading to better communication.

In the context of this thesis, we aim to enhance bar charts by utilizing fluid simulation to hopefully create a memorable visualization of volume differences in 3D objects.

2.2 Narrative Visualization and Storytelling with Data

Narrative visualization is a specialized approach within the visualization field that uses storytelling techniques to convey insights and meaning through data. It combines the power of visual representations with narrative structures to create compelling and engaging data-driven stories.

Segel et al. [40] conducted a design-space analysis of storytelling narrative visualizations from online journalism, comics, art, etc. They differentiate between author-driven and reader-driven narratives and state that most visualizations lie on a spectrum with these two extremes. An author-driven narrative refers to a linear path through the story. This approach focuses on communication and storytelling rather than interactivity and exploration. On the other hand, reader-driven narratives focus on interactivity and exploration on the reader's part, with less focus on linearity and storytelling.

Like author-driven narratives, Hullman et al. [22] conducted a user study to deepen understanding of the various forms and reactions associated with sequencing in narrative visualization. Their approach identifies potential transitions between visualizations based on an objective function that minimizes transition costs from the audience's perspective. The cost refers to what they call *transformation cost*, which amounts to the visual difference between each visualization. A high transformation cost leads to a high visual difference, which puts a further cognitive load on the user to understand the connection between the two visualizations. They question whether animated transitions can aid in overcoming the effects of transitions with a high cost.

2.3 Animation in Visualization and Animated Transitions

Animation has been widely employed in visualization to enhance information understanding, engagement, and retention. Animated transitions, in particular, offer a powerful tool for conveying changes, comparisons, and relationships between visualizations. By smoothly animating from one representation to another, animated transitions provide a visual continuity that facilitates the tracking of data changes and supports the viewer's mental model of the data.

Heer and Robertson [20] conducted a study on animated transitions in visualization, highlighting their effectiveness in aiding perception, supporting smooth transitions between visual representations, and improving user understanding. Various approaches and techniques for creating

animated transitions have been explored, ranging from keyframe manipulation to physics-based simulations. These techniques offer opportunities to dynamically convey data transformations and facilitate the storytelling aspects of visualization by guiding the viewer through a series of visual states.

In their user study, Kim et al. [26] explored the use of animation to help subjects understand aggregate operations applied on different graphs. They compare subject reactions to staged animation, interpolation, and static transitions that convey an aggregation of some data. The staged animations split in two, elaborate and basic staged animation. The elaborate staging is intended to show the operation performed entirely, focusing on giving the users a complete impression of the operation. The basic staging is similar but omits some stages from the elaborate staging to focus on simplicity. Interpolation animation refers to simply interpolating between the start and end states linearly. Static transitions refer to not animating the transition at all. Their results show that the subjects greatly preferred animation over static transitions across all aggregate operations. Subjects also tend to prefer the staged animations over interpolation, and preference over elaborate vs. basic staging seemed to vary based on the operation performed.

The speed at which an animation does not have to be constant. A non-constant speed transition can help make the transition easier to follow, as pointed out by Dragicevic et al. [13]. They evaluated object-tracking accuracy on animated transitions on point-based graphs, such as scatterplots. They found that a slow-in/slow-out approach resulted in the highest accuracy, meaning slowing down the animation speed at the start and end of the transition.

2.4 The Use of 3D in Visualization

When to use 3D and the benefits of 3D in visualization is a widely discussed topic [43, 31, 29]. In the early days of 3D visualization, the benefits of 3D over 2D visualization were overestimated, but in later years many of these perceived benefits were overturned [9, 10, 11].

There are, however, situations where 3D can perform better than 2D. Brath [7] documented several applications where the 3D interfaces showed benefits beyond 2D. He states that 3D visualization can benefit from several of the intrinsic properties of 3D vis, such as: using lighting models to reveal shapes in the data and utilizing the extra dimension to separate marks which normally overlap if rendered in 2D.

Stasko et al. [47] presented a categorization of 3D visualization tasks:

1. **Augmented 2D views** This category includes visualization techniques typically rendered in two dimensions, adding a third dimension for presentation or aesthetic purposes.
2. **Inherent 3D application domain views** Includes visualizations that deal with data inherently in three dimensions: volume rendering and 3D flow visualization.
3. **Adapted 2D views** Includes visualizations, usually rendered in 2D, using the third dimension to encode additional features or information.

In their user study, St. John et al. [46] exposed subjects to tasks that required shape recognition and tasks that required judging objects' relative positions. They found that 3D visualizations outperformed 2D in tasks requiring shape recognition but performed worse in tasks requiring relative position judging.

2.5 Fluid Simulation

This section will discuss some key work on fluid simulation, focusing primarily on particle-based methods. Firstly, two popular techniques for simulating fluids are presented, followed by a discussion of how fluid-solid interactions are represented computationally in these fluid solvers.

2.5.1 Smoothed Particle Hydrodynamics (SPH)

Smoothed Particle Hydrodynamics (SPH) [33] is a Lagrangian particle-based method commonly used for simulating fluid flows. SPH represents the fluid as a set of particles that interact with each other through pairwise forces. Each particle carries various attributes, such as position, velocity, density, and pressure. The key idea behind SPH is to estimate fluid properties at each particle location by interpolating values from neighboring particles using a smoothing kernel. This interpolation allows SPH to capture complex fluid behaviors, including surface tension, viscosity, and turbulence. SPH has been widely adopted in computer graphics and fluid simulation due to its versatility and ability to handle free surface flows. However, SPH can be computationally expensive, especially when dealing with large-scale simulations or complex fluid interactions.

Incompressibility refers to the characteristic property of some fluids that exhibit minimal volume changes when subjected to external pressures. For instance, water is a prominent example of an incompressible fluid, as its volume remains virtually unchanged even when subjected to

substantial pressure. In contrast, compressible fluids like air and other gases can be significantly compressed or expanded under varying pressures.

Enforcing incompressibility in sph is the most computationally costly part of the simulation process. with methods that enforce incompressibility, such as Weakly-compressible sph [4], being too expensive for real-time purposes on larger fluid simulations. Later work, such as by Solenthaler et al. [44], improves this computational cost by an order of magnitude by creating a prediction-correction scheme to determine particle pressures.

2.5.2 Position-Based Fluids (PBF)

Position-Based Fluid (PBF) by Macklin et al. [30] is another popular method for simulating fluid behavior. PBF is a particle-based method representing the fluid as a set of particles, where each particle carries attributes such as position, velocity, and density. PBF is an extension to the Position-Based Dynamics (PBD) system [35], which is used for cloth simulation. PBF employs a position-based constraint framework to enforce incompressibility, boundary conditions, and other fluid dynamics principles. The simulation proceeds by iteratively satisfying these constraints to achieve stable and physically plausible fluid behavior. PBF is known for its computational efficiency and stability, making it suitable for real-time simulations and interactive applications. It handles complex fluid phenomena such as splashing, swirling, and surface tension while controlling fluid viscosity and other material properties.

2.5.3 Fluid-Solid Interaction

Fluid-solid interaction in fluid dynamics refers to the constraints applied to the fluid by forces from solid objects. In a fluid simulation, proper fluid-solid interactions, in addition to other boundary conditions, are essential for accurately modeling the behavior of the fluid, as they help to ensure that the simulation is consistent with physical reality.

There are two different approaches to fluid-solid collisions: *no-slip* and *no-stick* [8], which dictate the behavior of fluid particles near solid boundaries. No-slip conditions assume the fluid particles stick to the solid surface and have zero velocity. In contrast, free-slip conditions allow fluid particles to slide along the boundary with a velocity that matches the tangential component of the fluid flow. In the context of the Liquid Conductor, it is most beneficial to use a free-slip approach. A no-slip approach would result in fluid sticking to the bar chart's sides, likely confusing users. The following subsections will introduce different methods to represent solid objects.

Signed Distance Fields

Signed distance fields are a volumetric approach that can concisely and efficiently represent solid geometry [17]. A distance function assigns a signed distance value to each point in space, indicating the distance from that point to the nearest surface of the solid. Positive distances indicate points outside the solid, while negative distances indicate points inside the solid. This representation allows for efficient collision detection and boundary handling in fluid simulations.

Boundary Particles

Boundary particles [34] are a common technique used in fluid simulation to enforce boundary conditions at the surface of an object. The basic idea is to define a set of particles that sit on the object's surface and interact with the fluid to mimic the behavior of a solid boundary. These particles are typically initialized with a high density and a strong repulsive force that prevents fluid particles from penetrating the boundary. Additionally, boundary particles may have a fixed velocity or be constrained to move along with the object they represent. Including boundary particles in a fluid simulation makes it possible to accurately model fluid behavior around complex shapes and provide a more realistic representation of fluid-solid interactions.

Ghost Particles

Ghost particles [12] are a common technique used in fluid simulation to enforce boundary conditions at the surface of an object. Similar to boundary particles, ghost particles are used to define a virtual boundary that interacts with the fluid in a way that mimics the behavior of a solid boundary. However, unlike boundary particles, ghost particles are not explicitly present in the simulation and do not take up physical space. Instead, they are used to compute an effective boundary force applied to the fluid particles near the object's surface. Ghost particles are typically placed a small distance from the object's surface and initialized with a low density and a weak repulsive force. By using ghost particles, it is possible to avoid the computational overhead of explicitly simulating the boundary particles while still accurately modeling fluid behavior around complex shapes and providing a more realistic representation of fluid-solid interactions.

Chapter 3

Related Work

This chapter will discuss two solutions that solve similar problems or use similar techniques to solve different problems.

Sorger et al. [45] created Metamorphers, a solution for creating reusable animated transitions for molecular data sets. Their method allows users to make these transitions using and combining pre-made templates, which they call *Metamorphers*. Figure 3.1 shows two examples of different combinations of metamorphers that can result in widely visually differing transitions. This gives users high levels of control to create their desired transitions. Similarly to our goal, their work seeks to automate the creation of animated transitions in three-dimensional space. However, their solution still requires some amount of work from the user. Their work is similar to our goal, but their method is limited to molecular data only.

In Basch's thesis [2], he delved into the realm of animated transitions by utilizing it to transition between a volumetric representation and a histogram representation, as seen in figure 3.2. This was done so users could examine the correlation between volumetric representations and histograms as an alternative to Linking and Brushing [5], which is commonly used. By leveraging

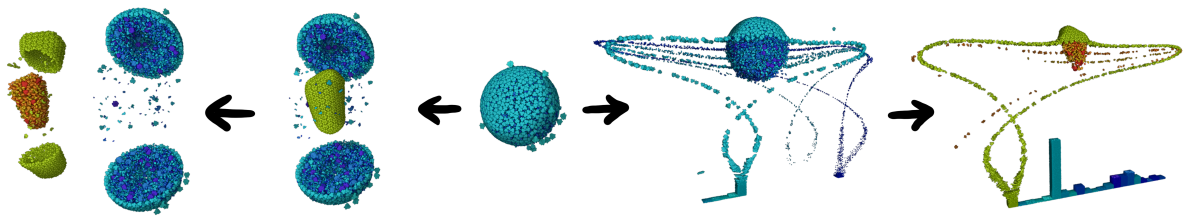

Figure 3.1: Example of where different combinations of metamorphers can lead to two different transitions

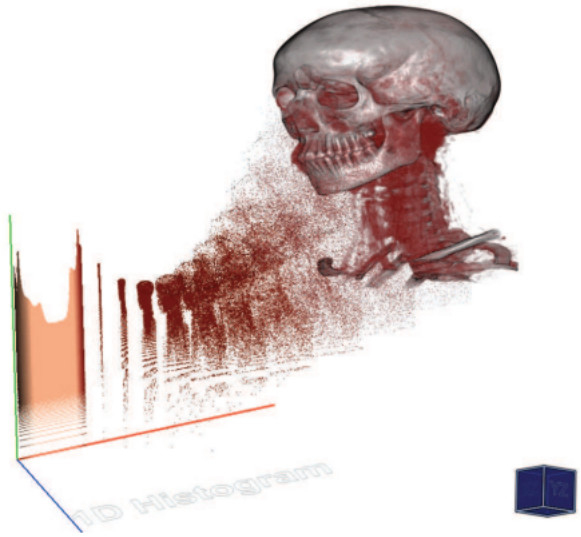

Figure 3.2: Example of Basch's animated transition (p. 55) [2]

animated transitions, Basch aimed to highlight the relationships and patterns within the data by smoothly transforming from a complex volumetric representation to a simplified histogram. This approach provided a bridge between the intricate three-dimensional space and the concise two-dimensional visualization, allowing viewers to observe correlations and variations in the data dimensions. Basch's research emphasized the significance of animated transitions in enhancing the understanding and communication of data relationships, offering valuable insights into how these transitions can be effectively applied. Inspired by this work, our thesis builds upon this concept by employing fluid simulation to create animated transitions from 3D models to 2D bar charts. We hope to show that using a fluid as a transition method is more intuitive and easily understood for most humans.

Chapter 4

Methodology

The Liquid Conductor aims to facilitate the creation of animated transitions that effectively communicate the volume differences in 3D models. It does so by transitioning from a 3D model's mesh representation to a bar chart representation using fluid simulation. It provides a user-friendly interface where users can input their 3D models and quickly start the visualization with very little setup required.

The application takes the user-inputted mesh, then applies a voxelization process to discretize the 3D space, creating a volumetric representation of the models. This new transformed data representation is further used to create a fluid container around the object's surface area, as well as seeding fluid particles inside its volume. The fluid simulation algorithms automatically generate realistic and visually appealing transitions between the 3D and 2D representations by utilizing fluid mechanics to transition. By leveraging the power of fluid simulation, the application enables users to convey complex information visually, engagingly, and intuitively, facilitating effective data communication and presentation.

The application also includes features for adjusting parameters such as particle density and viscosity. Users can also control the pacing of the transition, using the Death Plane component to control the fluid flow out of the container. This chapter will explain the main initialization pipeline, presenting components we identified required to create the Liquid Conductor. Later, we will present the components which are run during run-time.

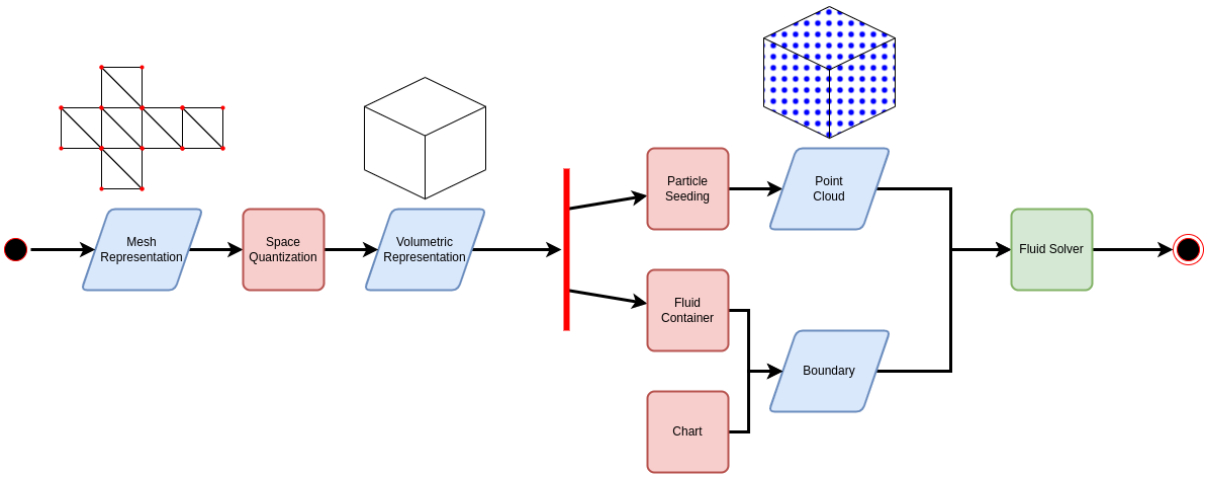

Figure 4.1: Flow diagram of how we go from a mesh representation to a fluid simulation

4.1 Initialization Pipeline

This section will explain the pipeline in which the Liquid Conductor takes a mesh representation of some 3D object and, through several transformation steps, results in a ready fluid simulation. This pipeline is part of the initialization process and runs on each user-inputted mesh. We will explain each step in the pipeline and how the data has been transformed along each step. Figure 4.1 shows what this pipeline looks like, with blue parallelograms showing the data transformation and red squares showing each component.

The pipeline takes one or more polygon meshes as input by the user. A Polygon mesh is a set of vertices and triangles, widely used for representing 3D objects [50]. This representation only contains surface information about the object. Before we can put fluid inside the mesh, we require some method that enables traversal of the mesh's volume. To do this, we need to quantize the space within the mesh. Therefore we call this component the *Space Quantizer*. A popular method for this is mesh voxelization [21].

Space Quantizer

Mesh voxelization converts a 3D mesh (a collection of triangles in 3D space) into a 3D grid of smaller cubes called voxels. Each voxel in the grid represents a small volume of space and has a value (such as empty or filled) corresponding to whether or not the original mesh occupies that volume. It enables us to identify voxels that intersect with the mesh boundaries, ensuring that fluid particles are placed within the confines of the mesh. Figure 4.2 shows mesh voxelization on an example model.

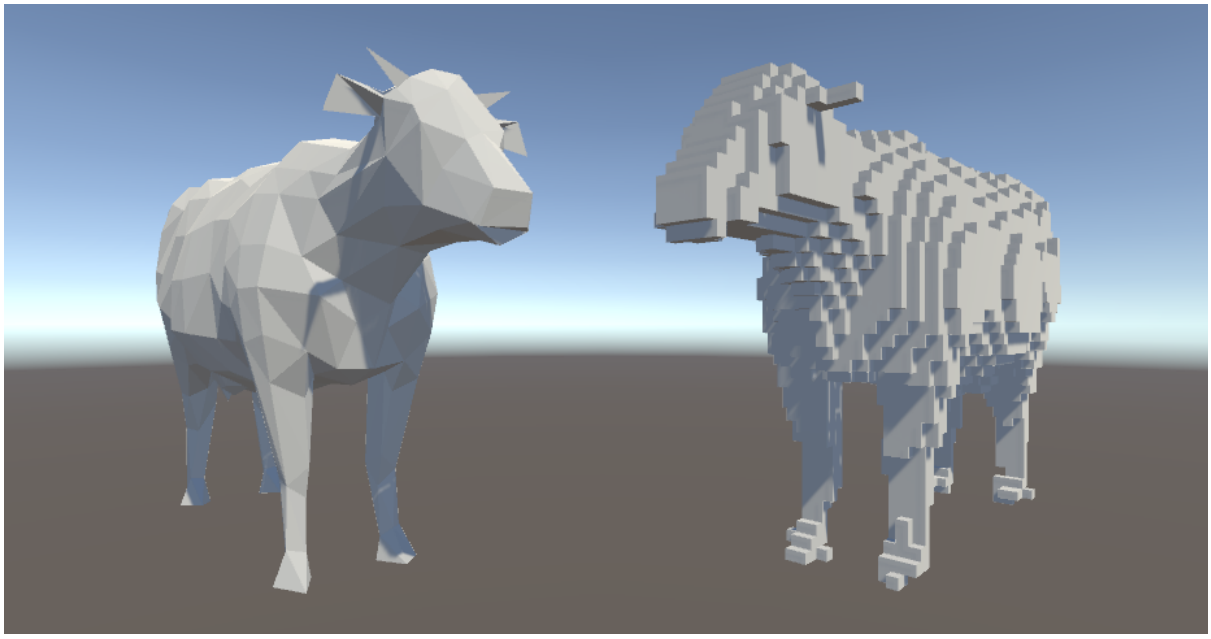

Figure 4.2: Cow 3D model before and after voxelization

After we have quantized the mesh volume, we now have a volumetric representation of the original mesh. This is where the pipeline fork in two, as seen in figure 4.1 by red fork-line. The next subsections will explain each branch.

4.1.1 From Volume to Point Cloud

This branch focuses on the non-greyed-out branch of the main pipelines, as shown in figure 4.3. This branch goes from the volumetric representation to a point cloud, which will eventually be given as input to the fluid solver

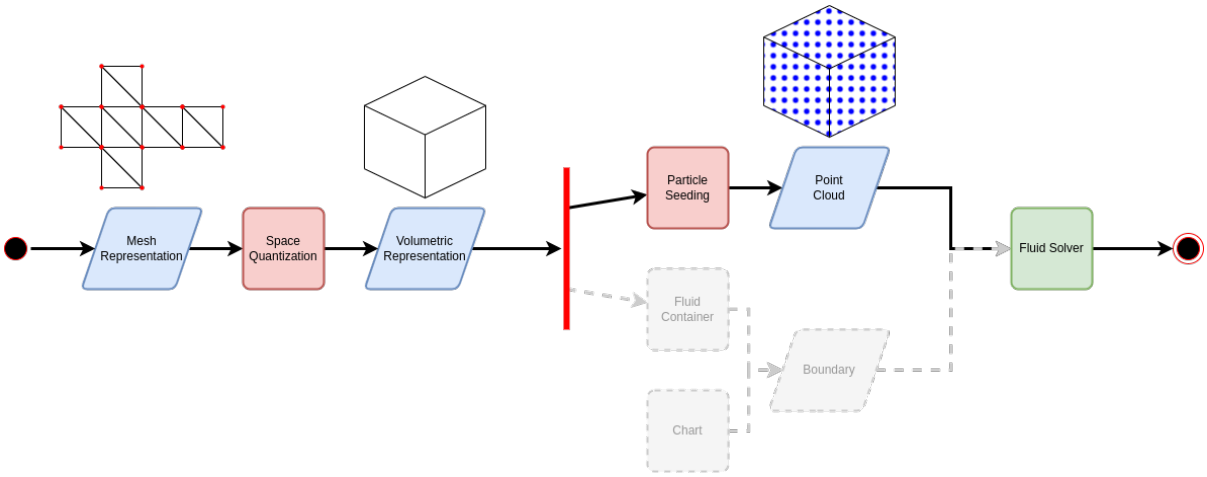

Figure 4.3: Flow diagram of how we go from a mesh representation to a fluid simulation

Particle Seeder

An essential step in the initialization process is seeding fluid particles within the mesh. This seeding method is crucial in defining the fluid particles' initial distribution and behavior, affecting the overall accuracy and realism of the simulation results. This section discusses the significance of a robust particle seeding method and highlights its key considerations.

To ensure an accurate representation of the fluid volume, the particle seeder needs to traverse the entire volume of the mesh. This traversal process involves systematically sampling points within the mesh, considering both its interior and surface regions.

One critical aspect of the particle seeding method is the appropriate spacing between the particles. Incorrect spacing between particles can result in under/over-sampling of the fluid volume, leading to inaccuracies in the volume estimation. Therefore, it is crucial to determine an appropriate spacing that strikes a balance between capturing the fine details of the fluid domain and maintaining computational efficiency.

After seeding fluid particles, the data has been transformed into a point cloud, each point being a point where our last component, the *fluid solver*, will place a fluid particle. The fluid solver contains the logic for fluid mechanics. The fluid simulation is nearly ready after all the fluid particles are placed.

4.1.2 From Volume to Boundary

This pipeline branch deals with creating boundary conditions for the fluid solver. Figure 4.4 shows what part of the main pipeline we focus on in this section.

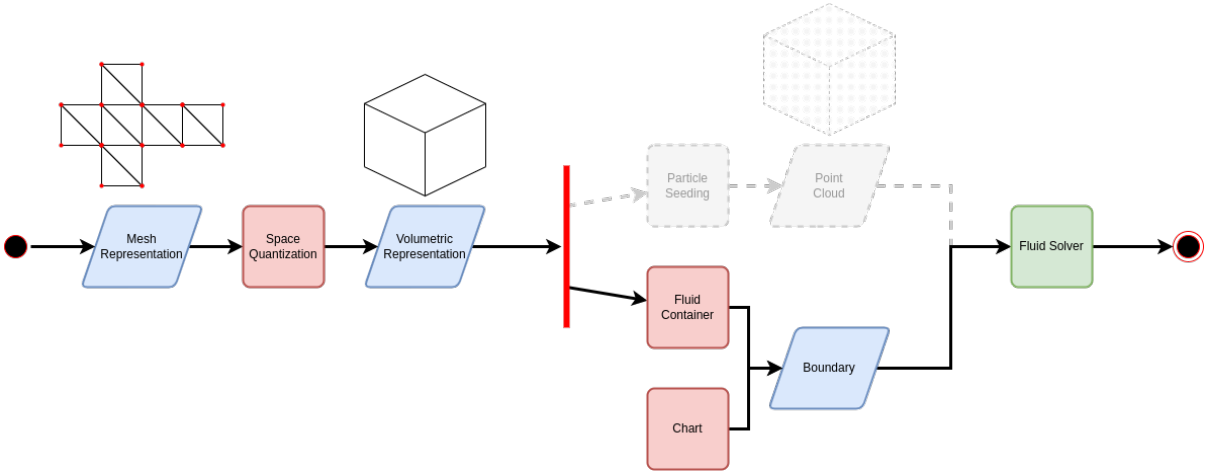

Figure 4.4: Flow diagram of how we go from a mesh representation to a fluid container

Fluid Container

We require a method that converts the surface of the mesh into a boundary condition for the fluid. This is so that the fluid is contained within the mesh at the start of the simulation, which will let us control the pacing of the transition. Doing so will also help users better understand the transition by letting them look at the liquid-filled 3D model. We can perform a morphological dilation step with the volumetric representation from the space quantization step to calculate the exact boundary needed to ensure a tight seal. We do not want to use the original mesh representation for this step because quantization methods such as voxelization approximating the volume will give some inaccuracies. Using the volumetric representation instead will be completely accurate, as the particle seeder uses the same data representation.

Morphological dilation [18] is a mathematical operation commonly used in image processing and computer vision to expand or grow the boundaries of objects in an image. The dilation operation works by expanding the boundaries of the voxels in the image according to some structuring element. For example, if the structuring element is a sphere of radius r, then each cell within a distance of r from the surface of the mesh will be included in the dilation, as shown in figure 4.5.

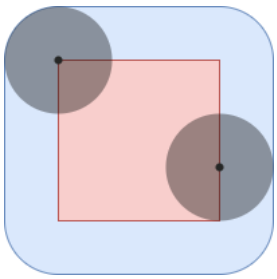

Figure 4.5: Dilation of a red square by a circular structuring element, the blue rounded square is the result of the dilation

Chart Boundary

In addition to the fluid container around the mesh, we need a boundary condition representing the chart we want to pour our liquid into. In our case, we focus solely on bar charts, which can be represented using cylinders. Creating boundary conditions for cylinders in 3D is quite simple. Using a Signed Distance Function/Field (SDF) as our boundary technique, we can use a capped cylinder distance function [38], with a small alteration to remove the top cap to let in fluids. When using boundary particles as the technique, we can seed boundary particles in vertically stacked circles, with a filled circle at the bottom to act as the bottom cap.

4.2 Run Time

This section will go over the components which run during the simulation. Firstly, we will review the fluid solver, which runs at initialization and run-time. Secondly, we will present our solutions for letting liquid pour into the bar chart, and lastly, we will present our solution on what projection to use in both 3D and 2D applications.

4.2.1 Fluid Solver

Fluid solvers are computational algorithms used to simulate the behavior and dynamics of fluid substances. These solvers enable the realistic modeling and animation of fluid motion.

The fluid solver takes the point cloud from the particle seeder and the boundary conditions from the chart and fluid container as input. The solver then utilizes these particles to simulate the fluid dynamics, interactions, and transformations over time. The boundary conditions act as solids, which the fluid collides and interacts with.

In the Liquid Conductor, we want our fluid simulation to mimic water, as it is the fluid humans are most familiar with. The solver must therefore enforce incompressibility, as a compressible fluid would have a varying volume depending on external forces.

4.2.2 Death Plane

The Fluid Container (as explained in section 4.1.2) ensures that the fluid stays within the space of the original mesh's volume at the start of the simulation. We now require a method that lets us slowly let out a controlled amount of fluid, which will pour directly into the bar. Simply releasing all the liquid at once would result in splashing and fluid particles possibly missing the bar entirely.

One solution is to ”pierce” a hole at the bottom of the container, meaning to remove a small part of the boundary on the fluid container, which will let the fluid particles out. The ability to control the size of the hole would also allow us to control the fluid flow out of the container, speeding up or slowing down the transition effectively.

One issue with this method is that local minimum points of the mesh will form pools of water, as shown in figure 4.6. One solution is to create a hole for every local and global minimum point on the mesh. Although this is simple to calculate, we found the effect to be visually confusing to look at.

Our alternate solution was to create a cube, which culls part of the Fluid Container within the cube's volume. Our solution goes as follows:

Step 1: Generate a cube with equal X and Z size as the mesh's bounding box but with a Y size of 0.

Step 2: Position the cube underneath the mesh's bounding box.

Step 3: Slowly increase the Y size of the cube

Step 4: Cull parts of the fluid boundary within the cube's volume

We call this solution the *Death Plane*, named after the plane placed in some video games to kill the player character if they ”fall” out of bounds. The Death Plane is a cube and not a plane. However, visually, it looks like a plane that cuts away fluid boundary conditions. Figure 4.7 shows how this works, with the Death Plane cutting the boundary.

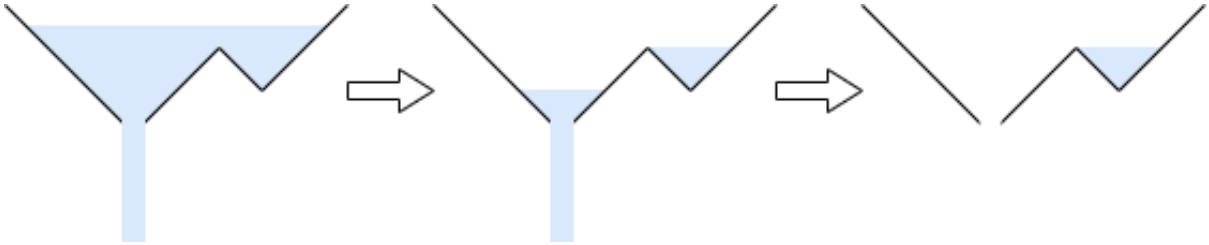

Figure 4.6: The problem of only creating a hole at the bottom-most point on the fluid container, local minimum points will create fluid pools.

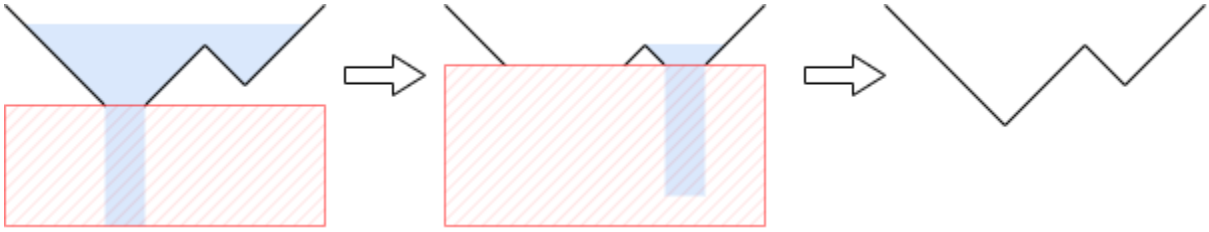

Figure 4.7: Deathplane (red box) cutting away the fluid container, allowing the blue fluid to pour out of the model, leaving it empty

4.2.3 Split Camera

This component is the least crucial, as it is not needed for the entire Liquid Conductor to work. However, this component enables users to read values off the final bar chart more accurately. This section will explain how cameras are used in 3D visualization, what camera projections are, and the two most common projections used for visualization: Perspective and Orthographic projection. Then an explanation of our split camera solution, which combines both projections.

In 3D visualization, virtual cameras determine how the three-dimensional scene is projected onto the screen. Camera projection is mapping the 3D points in the scene onto a 2D plane to create the final image. This projection is based on the type of projection chosen, such as perspective or orthographic projection.

The inherent traits of 3D visualization, such as depth perception and perspective distortion, make viewing such charts less effective. Therefore we need a method to alleviate or prevent these issues. One could use an orthographic projection of the entire view to solve both problems. Figure 4.9 shows how the same bar chart in 3D can be considerably more challenging to read when rendered in perspective projection than orthographic projection.

Orthographic projection impairs human shape recognition and understanding as it does not encode depth. Perspective projection is better suited for shape recognition, as it both shows depth cues and projects images the same way our eyes do, making it much more visually

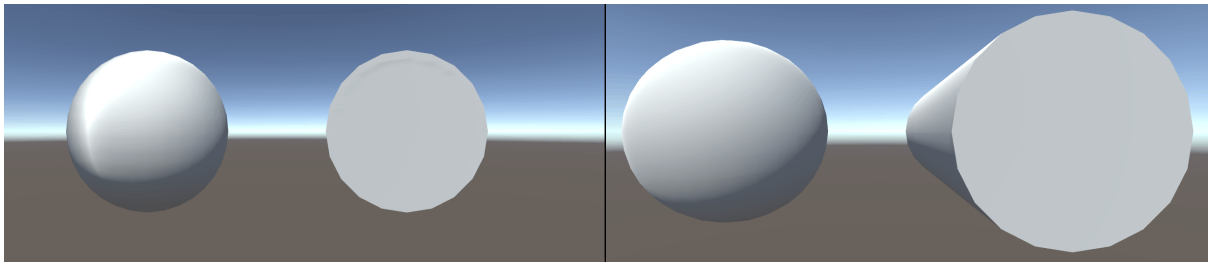

Figure 4.8: 3D models of a sphere and a cylinder, shown in perspective (left) and orthographic (right) projections

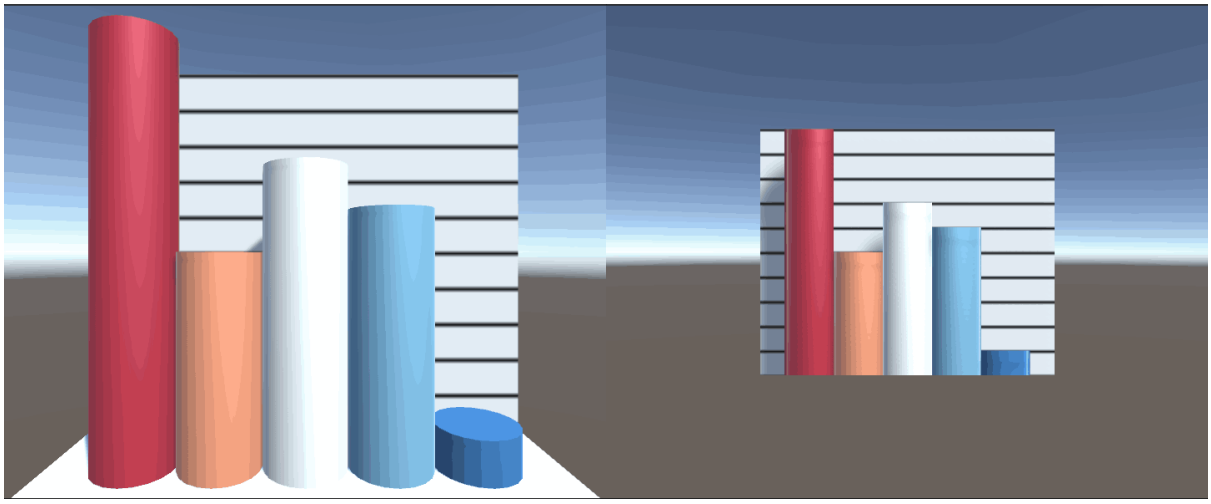

Figure 4.9: The same arbitrary bar chart shown in perspective(left) and orthographic(right) projections.

intuitive. Figure 4.8 shows the same two 3D models rendered in the two projection methods, and we can see how the orthographic projection does not let us interpret the object to the right as a cylinder. The shading alone is the only reason we can differentiate between the sphere and the cylinder.

Our solution is to split our view horizontally, with the top view showing the 3D model and using perspective projection. This ensures the best projection for the 3D model and the bar chart. The bottom view shows the bar chart in orthographic projection, as shown in Figure 4.10.

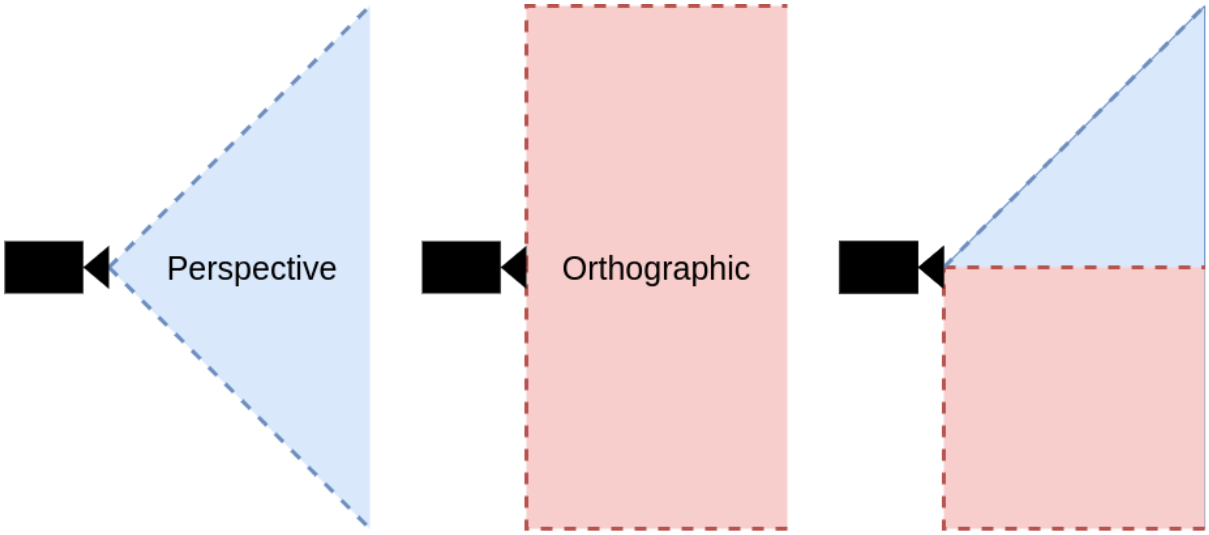

Figure 4.10: *Left*: perspective projection. *Middle*: Orthographic projection *Right*: How the split camera view uses both

4.3 Component Recap

This section serves as a quick recap of each component, briefly describing their role in the entire system and how they work.

Space Quantizer:
Transforms the mesh representation into a volumetric representation.

Particle Seeder:
Traverses the volumetric representation and calculates fluid particle positions. Converts the volumetric representation into a point cloud.

Fluid Container:
Ensures the fluid stays contained within the original mesh at the simulation start by creating a boundary along the surface area of the volumetric representation.

Chart Boundary:
It represents the bar that the fluid will eventually be poured into and acts as a container for the fluid, similar to the fluid container.

Fluid Solver:
It contains all the fluid mechanic logic and controls the fluid during the simulation.

Death Plane:
A plane which ”cuts” open the fluid container to let out the liquid in a controlled manner.

Split Camera:

Splits the view horizontally and uses both perspective and orthographic projection to ensure the best projection type for the chart and the 3D mesh.

Chapter 5

Implementation

This chapter reviews how the Liquid Conductor was implemented, including class diagrams, technologies, and algorithms. Firstly Additionally, we introduce solutions to issues not mentioned in chapter 4, such as spilling and volume estimation error. Figure 5.1 shows the liquid conductor's somewhat simplified Unified Modeling Language (UML) class diagram.

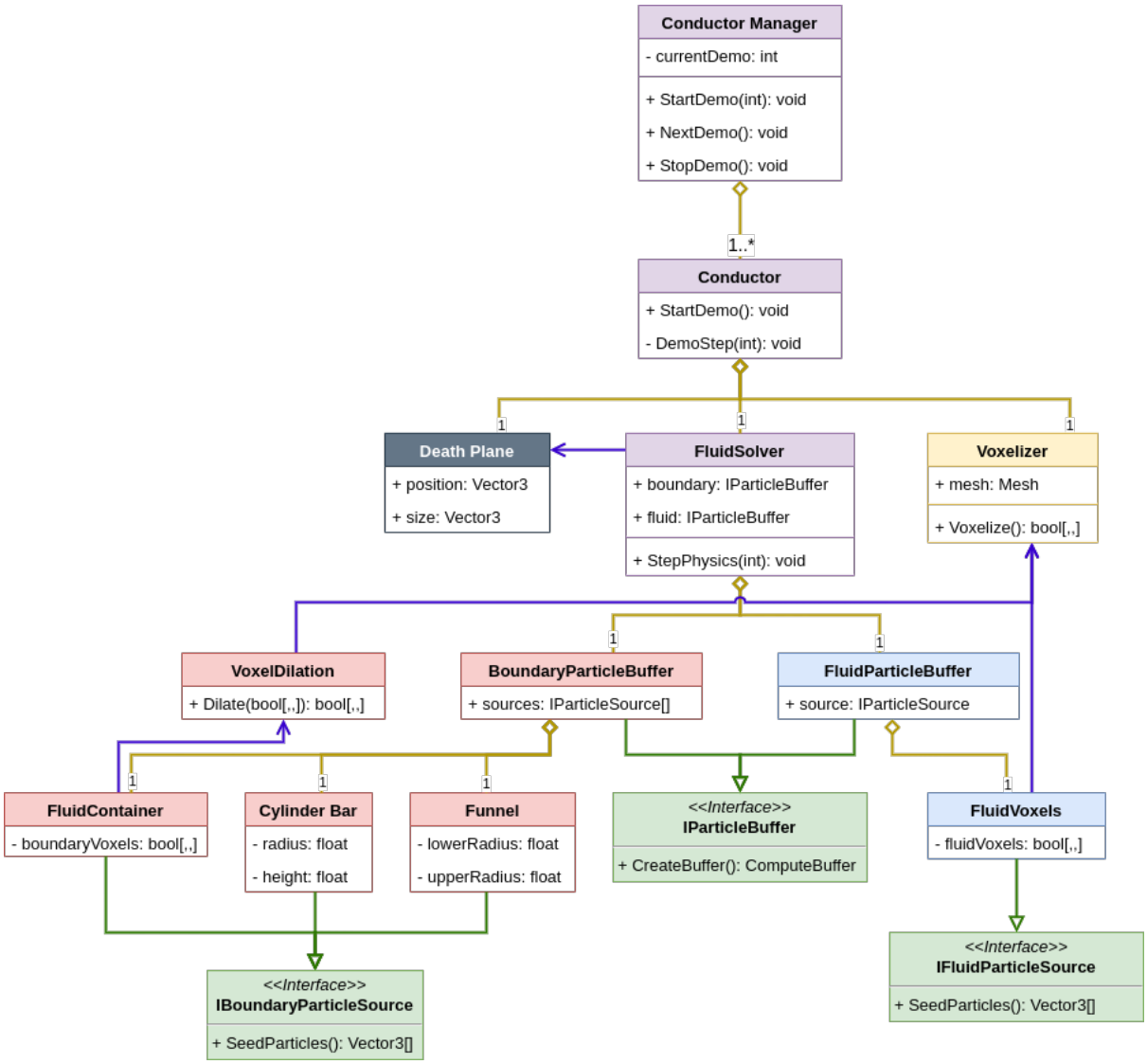

Figure 5.1: UML Class Diagram showing the architecture of the solution

5.1 Unity

To implement the software, I utilized the Unity Engine [49], a popular game engine widely used in the industry for game development and other interactive 3D applications. Unity offers a comprehensive set of tools and features, such as a powerful editor, physics engine, and scripting support, which allowed me to focus on the core functionality of my software, rather than spending time on low-level details of rendering, input handling, and other low-level systems.

One of the main advantages of using Unity for my project is that it provided a pre-built rendering engine with a variety of rendering pipelines and shading models, including the High Definition Render Pipeline (HDRP) and the Universal Render Pipeline (URP), which allowed me to choose the best option for my project's specific requirements. Additionally, Unity has built-in support for various file formats and asset types, such as 3D models, textures, and animations. This greatly streamlined the development process and reduced the amount of custom code needed.

Another major advantage of using Unity is its vast and active community of developers and users, which provides a wealth of resources, tutorials, and support. This community also offers a wide range of third-party plugins and assets, which can significantly extend the capabilities of Unity and provide additional functionality, such as AI, networking, and physics simulation.

Compared to developing my rendering engine from scratch, using Unity saved significant time and effort while achieving a high-quality result. Furthermore, by utilizing Unity, I was able to leverage the knowledge and expertise of the existing Unity community, as well as take advantage of the numerous features and tools provided by the engine, which helped me to focus on the specific requirements of my project and achieve my goals efficiently and effectively.

5.2 Conductor

The Conductor is the component that controls everything within a single simulation, i.e. if the user inputs three 3D models to compare, then three Conductor instances will each handle one model.

The main tasks of the Conductor are as follows:

1. Call factory methods for the Death Plane, Bar Chart, and Funnel

2. Call all particle source instances' method to seed particles
3. Initialize the Fluid Solver
4. Call the fluid solver's step function every frame
5. Dispose all unneeded ComputeBuffers, when stimulation is stopped

5.3 Conductor Manager

The manager is the user-visible component that lets users control each simulation. This is a component not discussed in the previous chapter. The Liquid Conductor Manager is a component that serves as a factory class [16], producing a Conductor instance per 3D model input by the user. It also controls each simulation, handling starting and stopping them individually.

We create separate simulations for each 3D model, mostly due to performance. Each simulation is also run sequentially, meaning a simulation is not started until the previous one is complete. This allows us to dispose of nearly all buffers after a simulation completes, significantly saving on memory usage and performance. A stopped conductor will preserve the last known position of each fluid particle, so the now-static fluid can still be rendered.

The Conductor Manager is the communication interface between the user interface and each conductor, passing messages from the UI to the appropriate Conductor.

5.4 Fluid Solver

To implement the fluid simulation component of my software, I chose to use the position-based fluid (PBF) method. PBF is a relatively recent technique for simulating fluid dynamics in real-time applications [30]. It has become increasingly popular in the game development and graphics communities due to its robustness and efficiency.

We used a GitHub repository by user Scrawk [39] as the starting point of our implementation, as it contained an implementation of position-based fluids in Unity. Several parts of the code had to be heavily altered to fit our purposes.

One of the main advantages of PBF is that it provides a stable and accurate fluid dynamics simulation, even at high resolutions and under complex scenarios, such as interacting with

solid objects or other fluids. This is achieved by using a position-based approach to simulate the fluid particles' motion, which allows for better handling of collisions and interactions and more efficient parallelization of the computation.

Another advantage of PBF is that it is computationally efficient and scalable, even on modern hardware with multiple cores or GPUs. This is achieved using a highly parallelizable algorithm that can distribute the computation across multiple threads or devices and using techniques such as spatial hashing and neighbor lists to optimize the simulation's performance.

Compared to smoothed particle hydrodynamics (SPH), another popular technique for simulating fluids, PBF, has several advantages. SPH can be more challenging to implement and optimize due to its reliance on kernel functions and interpolation, which can lead to numerical instability and require careful tuning of parameters. Additionally, SPH can be computationally expensive, especially at high resolutions or scenarios with high particle counts.

By using PBF for my fluid simulation component, I achieved a stable and efficient simulation of fluid dynamics while benefiting from its scalability and ease of implementation. Additionally, PBF allowed me to focus on the higher-level aspects of the simulation, such as the interaction with other components of the application, without having to spend significant amounts of time on low-level details of the fluid simulation algorithm.

A crucial part of a fluid solver's efficiency is finding neighboring particles quickly. Accessing neighboring particles is used in the solver's density, viscosity, and pressure calculations. The grid needs to be re-sorted every frame; it is, therefore, preferable to perform the sorting on the Graphics Processing Unit (GPU). Sorting algorithms need to be parallel to run on the GPU. Bitonic sort [3] was one of the earliest examples of a parallel sorting algorithm [24]. Scrawk's implementation of Position-Based Fluid uses Bitonic sort to sort a grid hash to find neighboring particles easily.

5.5 Mesh Voxelization

To implement my software, I used mesh voxelization to quantize the mesh volume, allowing me to seed fluid and boundary particles within the mesh. Voxelization converts a continuous object, such as a mesh, into a discrete representation of voxels (volumetric pixels) arranged in a regular grid.

One of the main advantages of using mesh voxelization is that it provides a flexible and efficient way to represent the geometry of a mesh, especially for tasks that require fast and accurate

computation of volumetric properties, such as ray tracing or collision detection. Additionally, voxelization allows for efficient parallelization of computation and storage of the data in memory, which can be critical for real-time applications or large-scale simulations.

Another advantage of mesh voxelization is that it enables various algorithms and techniques designed to work with volumetric data, such as Marching Cubes or Signed Distance Fields, which can be used for mesh reconstruction or surface extraction tasks. Additionally, voxelization can provide a more accurate representation of the geometry of a mesh than traditional mesh representations, especially for complex shapes or sharp features.

Compared to using other methods to quantize the volume of a mesh, such as tetrahedralization or irregular hexahedralization, mesh voxelization has several advantages. Voxelization is relatively simple and can be performed efficiently on modern hardware, even for large and complex meshes. Additionally, voxelization can be performed on non-manifold meshes, which can be challenging to handle with other methods.

By using mesh voxelization for my software, I achieved an efficient and accurate representation of the mesh volume, which allowed me to perform various operations, such as collision detection and ray tracing, quickly and accurately. Additionally, voxelization allowed me to take advantage of various algorithms and techniques designed to work with volumetric data, which helped me achieve my goals more efficiently and effectively.

During the development of the Liquid Conductor, we originally implemented the mesh voxelization to run on a single thread on the Central Processing Unit (CPU), as the voxelization is only done once for each 3D object on startup. However, this resulted in long startup times. Changing the implementation to a parallel solution, run in a compute shader on the GPU, improved these startup times drastically.

5.6 Filling the Volume With Fluid

The seeding method plays a crucial role in determining the initial distribution and behavior of the fluid particles, which directly impacts the accuracy and realism of the simulation results. To accurately represent the fluid volume, the particle seeder must traverse the entire volume of the mesh. This involves systematically sampling points within the mesh's interior and surface regions.

One key consideration in the particle seeding method is determining the appropriate spacing between the particles. The spacing should be carefully chosen to balance capturing the fine

details of the fluid domain and maintaining computational efficiency. Incorrect spacing can lead to under or over-sampling of the fluid volume, introducing inaccuracies in the volume estimation.

Once the fluid particles are seeded, the data is transformed into a point cloud, where each point represents a location for a fluid particle. The fluid solver's final component contains the logic for simulating fluid mechanics. The fluid simulation is nearly ready to proceed with all the fluid particles in place.

The mesh voxelization process gives us the volumetric representation to seed particles appropriately within the mesh volume. We can seed particles with proper spacing within each voxel to get a somewhat accurate amount of particles representing the mesh volume. The voxel size is an important aspect, which can widely vary the accuracy of this volume estimation.

To simplify this, we set the voxel size to equal the size of a single fluid particle. This somewhat alleviates the accuracy variation resulting from using some other voxel size. This also simplifies the seeding process, as now we only need to seed a single particle in the center of each voxel. However, the volume estimation from mesh voxelization is not completely removed. The next subsection will explain this further and present our solution to this problem.

5.6.1 Alleviating Volume Estimation Discrepancy

When working with voxelized meshes in fluid simulations, it is important to acknowledge that the voxelized representation may not perfectly preserve the volume of the original mesh. Voxelization, which discretizes the mesh into a regular grid of voxels, can introduce small inaccuracies and deviations from the precise geometry of the original object. As a result, the voxelized mesh may have a slightly different volume than the original mesh.

To address this volume discrepancy, a method is needed to accurately calculate the volume of the original mesh and use that value to compensate for the volume error during the seeding process in the fluid simulation. By determining the precise volume of the original mesh, adjustments can be made in the number and distribution of fluid particles to ensure a more accurate representation of the fluid-solid interaction.

To calculate the volume of a 3D mesh, we implemented the method described by Cha Zhang and Tsuhan Chen in their paper [53]. Their method calculates the signed volume of a tetrahedron formed by a mesh triangle and origin (see figure 5.2). The mesh volume is simply the

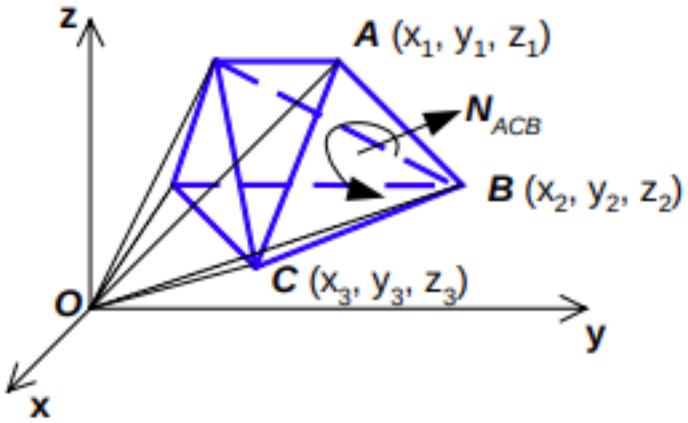

Figure 5.2: Calculation of 3D Volume [53] p.3

sum of the signed volume of each tetrahedron formed by each triangle in the mesh. This can be calculated with this formula:

$$V = \sum_i \frac{1}{6}(-x_{i_3}y_{i_2}z_{i_1} + x_{i_2}y_{i_3}z_{i_1} + x_{i_3}y_{i_1}z_{i_2} - x_{i_1}y_{i_3}z_{i_2} - x_{i_2}y_{i_1}z_{i_3} + x_{i_1}y_{i_2}z_{i_3}) \tag{5.1}$$

This method of calculating mesh volume does not assume the mesh is convex. Using this calculated volume, we can add extra fluid particles to compensate for the volume loss due to voxelization. The error margin is now the volume of a single fluid particle, thus significantly increasing the accuracy of our volume estimation.

5.7 Fluid Container

As discussed in the previous chapter, we discussed the need for a method to convert the surface of a mesh into a boundary condition for the fluid. This conversion is necessary to ensure that the fluid remains within the mesh during the simulation, allowing for better transition control and providing a visual representation of the liquid-filled 3D model.

We use the volumetric representation obtained from mesh voxelization instead of the original mesh representation. This representation is preferred over the original mesh representation because quantization methods like voxelization, which approximate the volume, introduce inaccuracies. Using the volumetric representation, a morphological dilation step can calculate the exact boundary required for a tight seal. Morphological dilation is a mathematical operation commonly used in image processing and computer vision. It involves expanding or growing

the boundaries of objects in an image using a structuring element. In this case, the structuring element is the 26 neighboring voxels of a given voxel. By applying morphological dilation to the volumetric representation, the boundaries of the voxels within a certain distance from the surface of the mesh can be expanded, ensuring a tight seal for the fluid simulation.

To implement this, we implemented a Flood fill algorithm with a modification that traverses the volume one extra step after hitting the border, marking them as boundary voxels. This ensures that our algorithm finds every cell outside the mesh, which neighbors a voxel within the mesh. The flood fill is implemented as follows:

Algorithm 1 Modified Iterative Flood fill using a Stack

```
procedure FLOODFILL(start)
    boundaryVoxels ← Empty List
    S ← Empty Stack
    S.push(start)
    while S is not empty do
        v ← S.pop()
        mark v as visited

        if not INSIDE(v) then
            add v to boundaryVoxels
            continue
        end if

        for all w ∈ NEIGHBORS(v) do
            if w is not visited then
                S.push(w)
            end if
        end for
    end while
    return boundaryVoxels
end procedure
```

The function INSIDE(v) returns a Boolean value representing whether the voxel v is inside the voxelized mesh, and the function NEIGHBORS(v) returns the 26 neighboring voxels of v. The returned value, *boundaryVoxels*, contains all the voxels in the dilation.

5.8 Bar Chart

We chose to use boundary particles to represent all solids in the fluid simulation, both due to the fluid solver already supporting them and due to boundary particles enabling us to easily create representations of complex shapes, which was needed for the fluid container.

We used boundary particles to represent the bar chart and the funnel presented in section 5.11 instead of a solution that uses both signed distance functions and boundary particles. The reason for this was to avoid overcomplicating the implementation. However, using such a hybrid solution could possibly improve performance.

We implemented the method of seeding boundary particles in a cylinder with a bottom cap to create a boundary condition for the cylindrical bar chart. The cylinder shape is made by seeding boundary particles in vertically stacked circles. The bottom cap is made by seeding concentric circles at the bottom of the cylinder. This is seen in algorithm 2, where the function SEEDCYLINDER is the method for creating the cylindrical bar chart.

Algorithm 2 Seed Bar Chart Particles

```
procedure SEEDCYLINDER(h, r, s)
    particles ← empty list
    SEEDFILLEDCIRCLE(0, r, s)
    for y ← 0, h do
        SEEDCIRCLE(particles, r, s, y)
    end for
    return particles
end procedure

procedure SEEDFILLEDCIRCLE(particles, r, s, y)
    while r ≥ s do
        SEEDCIRCLE(particles, r ,y)
        r− = s
    end while
end procedure

procedure SEEDCIRCLE(particles, r, s, y)
    n ← ⌊2 ∗ π ∗ r/s⌋
    θ ← 2π/n
    for i ← 0, n do
        v ← (r ∗ cos(θ ∗ i), y, r ∗ sin(θ ∗ i))
        add v to particles
    end for
end procedure
```

The parameters in SEEDCYLINDER in algorithm 2 h and r refer to the cylinder's height and radius, respectively, while the parameter s is the desired spacing between each particle.

5.9 Death Plane

We created the *Death Plane* to let the fluid out of the container. Which is a plane that effectively removes the boundary particles below it, as shown in Figure 5.3. As the simulation runs, the user can drag a GUI slider to move the death plane up, thus removing more boundary particles. This allows the user to control how much liquid is poured from the mesh.

In the fluid density and pressure calculations, a fluid particle iterates through boundary particles in neighboring grid cells to determine the fluid-solid forces on the fluid particle. Boundary particles within the Death Plane's volume are ignored in these calculations. We determine if a boundary particle is within the Death Plane by a SDF on a box [38]:

```
float sdBox( vec3 p, vec3 b )
{
  vec3 q = abs(p) - b;
  return length(max(q,0.0)) + min(max(q.x,max(q.y,q.z)),0.0);
}
```

Where p is the position of the particle relative to the box center, and b is the box size. A boundary particle is within the box if the return value of the signed distance function is negative. Boundary particles with a negative signed distance to the Death Plane are ignored in density calculations.

We do not delete boundary particles culled by the Death Plane because removing elements from a ComputeBuffer requires re-initialized and re-filled, which will often hurt performance more than gained from freeing the space.

To properly convey that the model is being "cut" to the user, we created a special surface shader, as shown in figure 5.4. The shader hides any part of the mesh within the Deathplane box and creates a red "cutting line" to communicate the actual cutting. Normally, the interior of 3D models is not visible to the camera and therefore is not typically rendered. However, we would expect to see the interior since we are cutting away some parts of the outer surfaces. Therefore in our Death Plane culling shader, we render the interior and darken the interior slightly to create the illusion of shade.

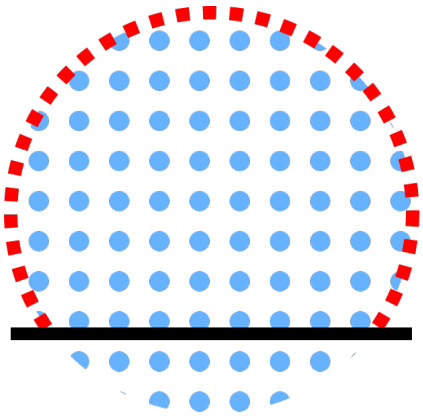

Figure 5.3: Deathplane (black line) cutting away the red boundary particles, allowing the blue fluid particles to pour out of the model

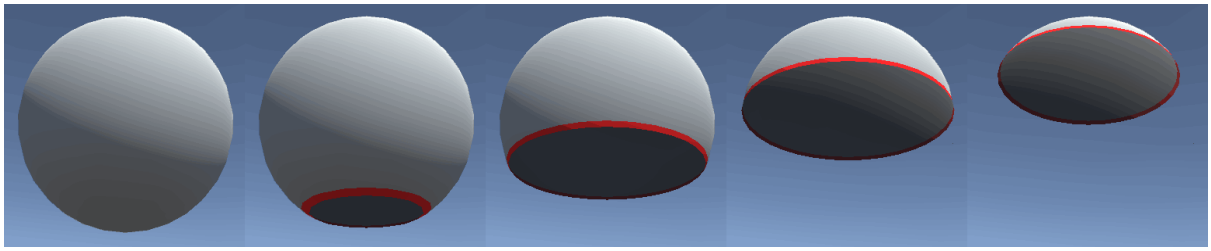

Figure 5.4: Sphere using the Deathplane culling shader

5.10 Split Camera

Perspective projection is commonly used in 3D visualization to mimic how the human eye perceives depth in the real world. It stimulates the convergence of parallel lines towards a vanishing point, creating a sense of depth and distance. In a perspective projection, objects closer to the camera appear larger, while those farther away appear smaller. This projection technique provides a more realistic representation of the scene, making it suitable for applications where spatial relationships and depth perception are important, such as architectural visualization or virtual reality experiences.

On the other hand, *orthographic projection* is a technique that preserves parallelism and does not consider the depth or distance of objects in the scene. In orthographic projection, objects maintain their size and shape regardless of their distance from the camera. This projection is often used in technical drawings, engineering visualizations, or when a more flattened or geometric representation is desired. Orthographic projection is useful when precise measurements or accurate scaling of objects are critical, as it eliminates the distortion caused by perspective projection.

3D visualization applications can create immersive and accurate representations of the virtual environment by manipulating these camera parameters and selecting the appropriate projection type. Whether employing perspective projection to simulate realistic depth perception or orthographic projection for precise scaling and measurement, understanding the principles of camera projection is fundamental to achieving compelling visualizations in 3D visualization.

Unity supports both perspective and orthographic projections in its Camera component. We use a combination of three cameras to implement our Split Camera component. Two of the cameras are the perspective and orthographic cameras, and the third combines the results from the other cameras and renders onto the screen.

The perspective and orthographic cameras each render into separate RenderTexture instances. A third camera samples these textures based on the y coordinate of the screen pixel. The third camera also samples the skybox and blends it in. This is implemented as follows:

```
fixed4 split_camera(float2 uv) {
    const float frag_height = uv.y - _SplitHeight;
    // Perspective Camera
    if (frag_height > 0){
        return tex2D(_PerspectiveTex, uv);
    }
    //Orthographic
    return tex2D(_OrthographicTex, uv);
}
```

Where *uv* is the position of the screen pixel in texture space.

5.10.1 Minimizing Seam

Rendering two different camera projections on the screen shows an obvious seam at the split point. This is due to the same screen pixel not always equating to the same position in the world space in two different projections. However, there is a specific point at which orthographic and perspective projection align. This alignment point is where the projection lines of the orthographic and perspective cameras intersect. Figure 5.5 shows how the seam's visibility changes based on how far a cylinder is from the alignment point. The cylinders are all placed directly in front of their cameras. The seam distortion is even more apparent when objects are

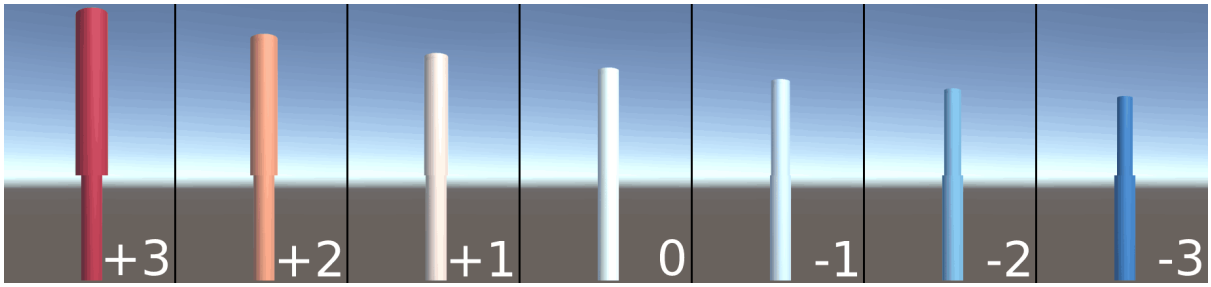

Figure 5.5: The split camera seam, shown on centered cylinders placed at different distances from the alignment point, with 0 being placed at the alignment point

placed off-center. We place the camera to minimize the seam distortion so that the simulation bound is on the alignment point. Ideally, we would place the simulation center on the alignment point, but then some of the top parts of the simulation will not be in the frame. Figure 5.6 shows how we place the camera for the least amount of seam distortion while still having the entire simulation in the frame. We position the camera in the following way:

$$C = \begin{cases} x & = S_x \\ y & = S_y \\ z & = S_z - B_z - \frac{O}{tan\frac{\theta}{2}} \end{cases}$$

$C =$ Camera Position
$S =$ Simulation Center
$B =$ Simulation Bounds Size
$O =$ Vertical Orthographic View Size
$\theta =$ Vertical Perspective Projection Angle

This is when the center of the simulation bounds is on the same XZ-coordinate as the line-line-intersection between the two camera projections. The camera is placed at the same Y-coordinate as the center of the simulation bounds. The seam is now the least visible at the center of the simulation bounds, which the pouring fluid often pours through.

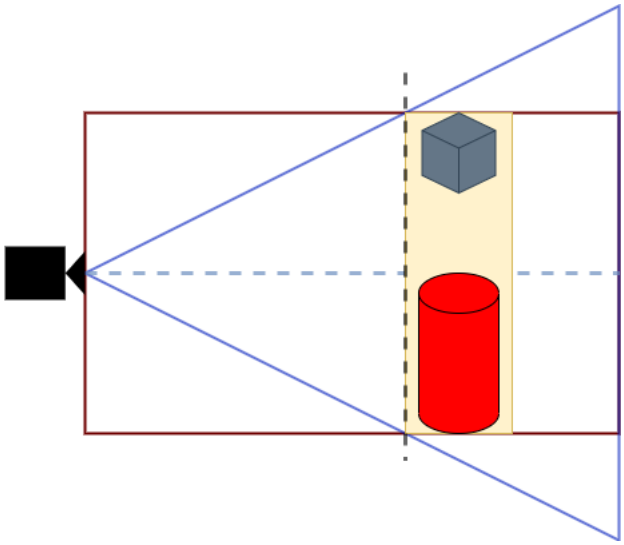

Figure 5.6: The Camera is placed so the projection lines of the perspective (blue) and orthographic (red) intersect at the z-coordinate of the front face of the simulation bounds

5.11 Spill Prevention: Funnel

Due to the unknown size and shape of the mesh, we cannot guarantee that all fluid particles will naturally fall into the cylindrical bar chart. To prevent this, we created a ”funnel” component to guide fluid particles into the bar.

The funnel consists of boundary particles aligned in a conical shape placed at the top of the bar. The smaller bottom radius of the funnel is set to equal the radius of the bar it sits upon. While the larger top radius is set to the extent of the simulation bounds. The funnel height is set so that the top is on the same y-coordinate as the bottom of the mesh.

Algorithm 3 Seed Funnel Particles

```
procedure SEEDFUNNEL(h, r_0, r_1, s)
    particles ← empty list
    for y ← 0, h do
        r ← lerp(r_0, r_1, y/h)
        SEEDCIRCLE(particles, r, s, h)
    end for
    return particles
end procedure
```

In algorithm 3, the parameters h, r_0, and r_1 are the funnel's height, lower radius, and upper radius, respectively. The parameter s is the spacing between each particle. The called function, SEEDCIRCLE, is the same as shown in algorithm 2 in section 5.8.

5.12 Rendering

For the isosurface reconstruction of the fluid in my application, sphere tracing is performed on a signed distance field (SDF) representation of the fluid density. SDF is a method for representing the distance to a surface from a point in space. They can be computed efficiently from various sources, including meshes, point clouds, or volume data.

Sphere tracing [19] is a technique for traversing a volume to find isosurfaces. The algorithm iteratively samples the SDF in a sphere until it reaches a zero-crossing in the SDF, meaning the fluid surface is found. This approach allows for efficient computation of the surface and can produce high-quality results, especially for smooth and well-behaved surfaces.

One advantage of using sphere tracing on SDFs for isosurface reconstruction is that it provides a flexible and efficient way to represent the fluid volume and its surface in real time. Additionally, sphere tracing allows for efficient parallelization of computation and can be performed on modern hardware, even for large and complex volumes.

To calculate the lighting in the fluid surface rendering surface shader, we need to calculate the surface normal for a point on the fluid surface. This is done by sampling the gradient of the density field, which represents the direction of the steepest increase in density at each point in the volume [19]. This approach allows for accurate surface orientation and shading computation, critical for realistic rendering and visualization of the liquid surface.

Due to our mixed camera projection from the split camera component, the ray direction needs to be changed based on what projection the camera currently rendering is using.

```
struct Ray {
    float3 origin;
    float3 dir;
};

Ray get_ray(float3 camera_world, float3 frag_world){
    Ray r;
    r.origin = frag_world;

    if (unity_OrthoParams.w == 1.0){ // Orthographic projection
        r.dir = UNITY_MATRIX_IT_MV[2].xyz;
    }
    else{ // Perspective projection
        r.dir = normalize(frag_world - camera_world);
    }
    return r;
}
```

`unity_OrthoParams.w` is a float4 provided by the camera, where the w value is 1 when the camera is in an orthographic projection, and 0 otherwise. `UNITY_MATRIX_IT_MV` is the model*view matrix of the current camera. This means that `UNITY_MATRIX_IT_MV[2].xyz` will give us the camera's forward vector in world space.

I rendered the fluid completely opaque, without any complex shading or transparency effects, as shown in figure 5.7. Rendering it opaque was to make the edges of the fluid more distinct, which was essential for users to read the value on the bar chart.

5.13 Running the Liquid Conductor

To be able to run the liquid conductor, the user will need to do a few tasks first. Firstly they must input the 3D models they wish to run the program on. Then, the user has to input the size of the simulation bounds and particle size. The bar height is set to be half the height of the simulation bounds. This is to ensure that the seam created by the split camera is less obvious. The radius of a bar is determined automatically so that the volume of the cylinder equals the

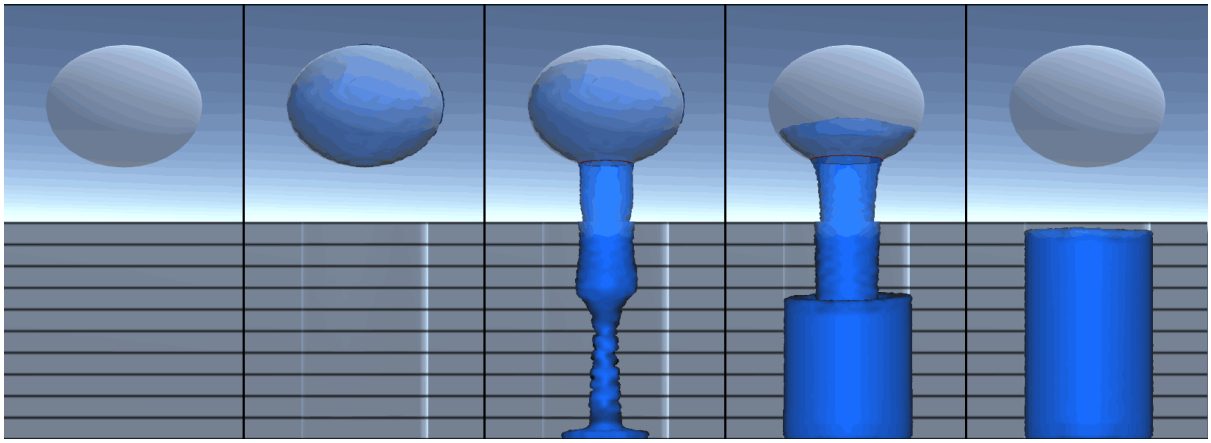

Figure 5.7: Liquid Conductor being run on a sphere, showing how the fluid is poured

volume of the mesh with the most volume. The result is that the most voluminous mesh will fill its bar fully. This is done so that the fill height of the other meshes is then its volume compared to the most voluminous mesh. If a mesh fills its bar halfway, this means that its volume is equal to half of the largest mesh.

Users are given a vertical UI slider, which controls the Death plane, to directly control the fluid flow out of the mesh boundary hull. The Deathplane could be controlled automatically, but different meshes require wildly varying amounts of fluid flow to be as expressive as possible. Therefore we decided to let users control the flow themselves. Also, we found the slider to make the application more engaging.

Figure 5.7 shows the entire pouring process of the Liquid Conductor being run on an arbitrary sphere, with the left-most image showing the first frame, where only the 3D model is shown. The next image is when the user has started the simulation, but has not moved the Deathplane yet, so all the liquid is held within the mesh boundary shell. The next two images show the sphere being cut open by the Deathplane, letting out liquid to fill the bar. The last image shows the final result showing the empty 3D model and the bar chart filled with all the liquid.

Chapter 6

Results

This chapter presents the results of implementing and evaluating the Liquid Conductor application. In this chapter, we showcase two applications where the Liquid Conductor proves to be valuable in the domain of data visualization. Furthermore, we delve into the application's performance, highlighting its efficiency and effectiveness in generating animated transitions from 3D models to 2D bar charts. The results presented in this chapter validate the capabilities of the Liquid Conductor and shed light on its potential for enhancing the presentational and communicative aspects of data visualization.

6.1 Use Case: Platonic Solids

Platonic solids are a set of five regular polyhedra that have identical faces, edges, and vertices. The Platonic solids include the *tetrahedron*, *hexahedron*, *octahedron*, *dodecahedron*, and *icosahedron* (as shown in figure 6.1). Each Platonic solid has a unique set of geometric properties, such as symmetry, regularity, and uniformity, Which makes them a study topic in mathematics, geometry, and physics.

If we compare the volume of each polyhedron when they have the same *circumradius*, meaning the radius of a sphere that touches each of the vertices of a polyhedron. Intuitively, one would assume that the polyhedron most similar to a sphere (the polyhedron with the most vertices) would have the greatest volume. This is not the case, as seen in table 6.1. One can see that given the same circumradius, the dodecahedron has the greatest volume, rather than the icosahedron.

Figure 6.2 shows the result of running our program on all these five solids; the solids are sized so their circumscribed sphere would be of equal size. This indicates that the dodecahedron has the most significant volume of the five shapes.

Shape	Faces	Circumradius r	Volume	Volume (r=2)
Tetrahedron	4	$\frac{\sqrt{6}}{4}a$	$\frac{\sqrt{2}}{12}a^3$	≈ 4.11
Hexahedron	6	$\frac{\sqrt{3}}{2}a$	a^3	≈ 12.32
Octahedron	8	$\frac{\sqrt{2}}{2}a$	$\frac{1}{3}\sqrt{2}a^3$	≈ 10.67
Dodecahedron	12	$\frac{\sqrt{3}}{4}(1+\sqrt{5})a$	$\frac{1}{4}(15+7\sqrt{5})a^3$	≈ 22.28
Icosahedron	20	$\sin\frac{2\pi}{5}a$	$\frac{5}{12}(3+\sqrt{5})a^3$	≈ 20.29

Table 6.1: Table of the platonic solid's volume equations

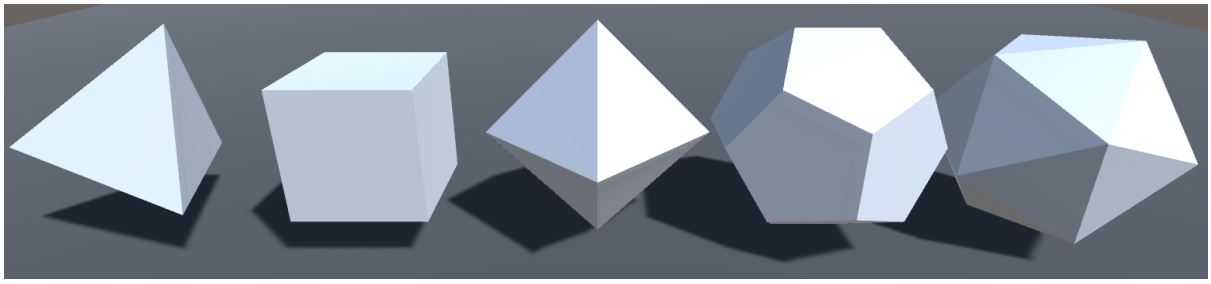

Figure 6.1: All five platonic solids

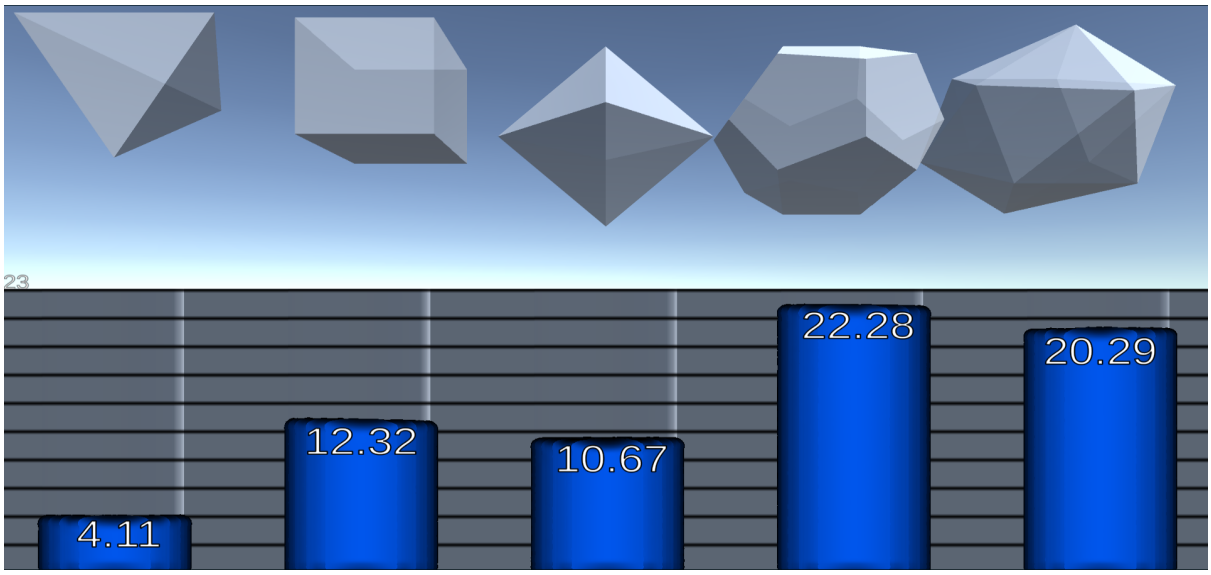

Figure 6.2: Result of the platonic solids being run, where r=2

6.2 Use Case: Comparing Earth to Its Neighboring Planets

This section presents a specific use case to demonstrate the practical application of our proposed visualization approach. We utilize our application to compare the volumes of 3D models representing Earth, Mars, and Venus. By employing fluid simulation and animated transitions, we aim to provide a visually engaging and intuitive representation of the variations in planetary volumes.

It is commonly observed that people often hold preconceived assumptions about the relative sizes of celestial bodies such as Earth, Mars, and Venus. However, these assumptions are frequently inaccurate due to the vastness of planetary scales. People tend to underestimate or overestimate the size differences, leading to misconceptions about the actual variations in volume.

By utilizing our application to compare the volumes of 3D models of Earth, Mars, and Venus, we aim to create an engaging and interactive way to rectify these misconceptions and provide a more accurate representation of the size disparities. Our approach's animated transitions and fluid simulation techniques offer viewers a novel, immersive experience that vividly portrays the true differences in planetary volumes. Through this visual exploration, we can help correct misperceptions and foster a deeper understanding of the actual size relationships between these celestial bodies.

Comparing planetary volumes is important to understanding celestial bodies' physical characteristics and scale. However, traditional methods of comparing volumes, such as numerical measurements or static visualizations, often fail to effectively convey the true magnitude and spatial relationships. Through our application, we seek to overcome these limitations and offer a more immersive and informative experience.

The fluid simulation techniques employed in our approach enable the creation of a fluid container that acts as a hull around each planetary model, providing a visual representation of the volume. Through the animated transitions, viewers can observe and compare the relative volumes of the three planets.

Our use case not only demonstrates the capability of our application in visualizing the volume of 3D models but also showcases the potential for broader applications in the field of planetary science, education, and public outreach. By providing an engaging and interactive platform, we enable users to explore and comprehend complex scientific concepts in an intuitive and accessible manner.

Planet	Equatorial Radius		Volume	
	km	earths	km³	earths
Earth	≈ 6371	1	$\approx 1.0832 * 10^{12}$	1
Venus	≈ 6051	0.949	$\approx 0.9280 * 10^{12}$	0.857
Mars	≈ 3389	0.531	$\approx 0.1630 * 10^{12}$	0.151

Table 6.2: Table comparing the equatorial radius and *volume* using the metric system, and a comparative unit to earth's radius and volume. (eq. radius data from [41])

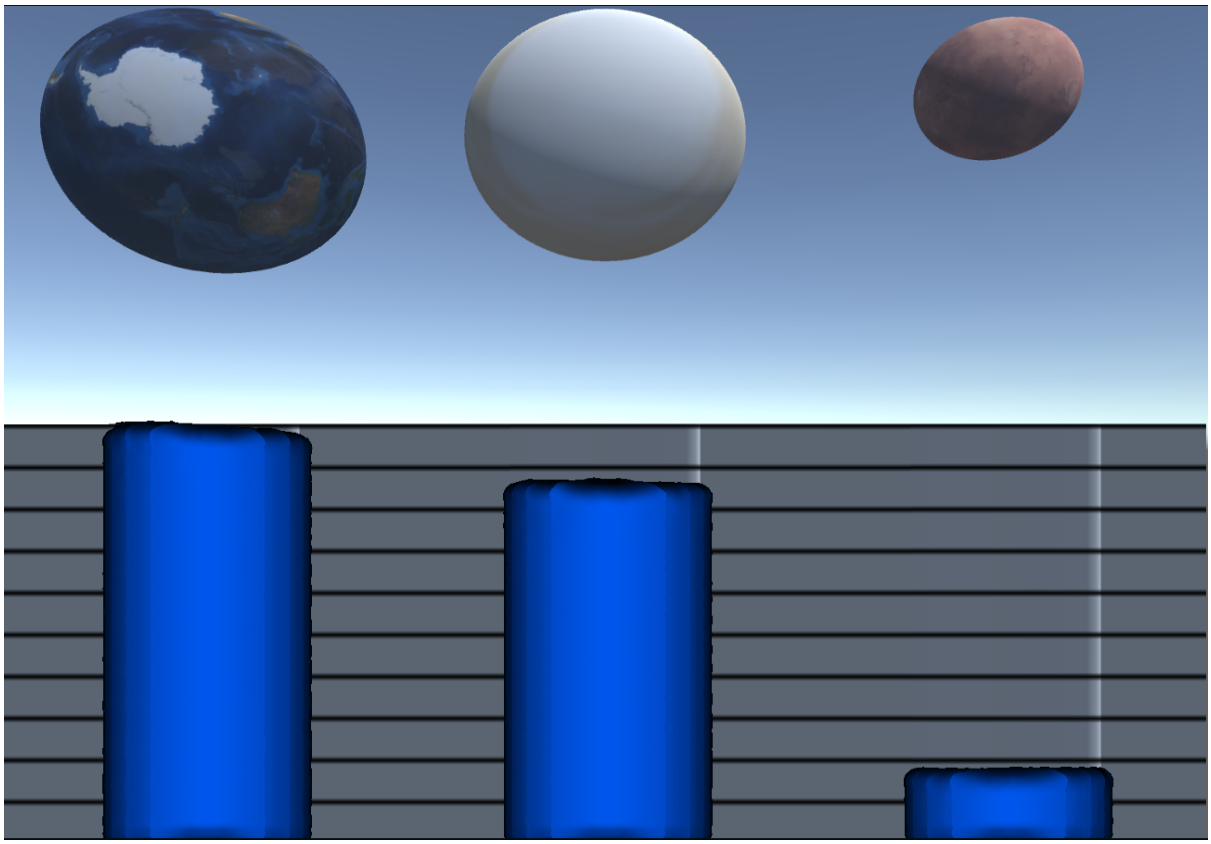

Figure 6.3: Result of the Liquid Conductor being run on 3D models of Earth, Venus, and Mars

6.3 Performance

Our fluid surface reconstruction is straightforward due to not doing any reflection or refraction, so it does not affect performance too much. Performance is highly dependent on the number of particles in the simulation. So larger 3D models will perform significantly worse. Increasing particle size will improve the performance of the simulation. However, larger particle sizes will result in a visually lower quality simulation. Scaling down every model by the same amount will produce fewer particles, resulting in better performance, but will also lower simulation quality.

Using Unity's profiler, we can see that the fluid solver's simulation steps is the process that affects the performance the most, but this was expected and can only be improved by optimizing the solver algorithm further. The split camera also affects the performance significantly. This is because the split camera makes the screen render thrice for every frame, once for the perspective camera, once for the orthographic camera, and finally once for the third camera, combining the two other cameras rendering.

The start-up time heavily depends on each 3D model's size and vertex count. This is because the voxelization must be run on each model at startup. Additionally, the volume discrepancy alleviation discussed in section 5.6.1 is a much more significant factor in startup time and is directly affected by vertex count. To fix this, we give users the option not to do this step, making the volume estimation less accurate and the startup time much faster.

Fluid Particles	Boundary Particles	CPU time (ms)
15K	14K	16ms
36K	23K	28ms
71K	35K	45ms
122K	48K	75ms

Table 6.3: Table showing performance of the entire Liquid Conductor with different particle counts. *(Program was run on a NVIDIA GeForce RTX 3060 Laptop GPU)*

Chapter 7

Discussion

Our application currently only supports using spacial volume as the encoding feature; further work could be done to increase its feature options, such as surface area could be visualized without much change to the source code. More work could be done to make the transitions more narrative, such as showing the model better by zooming in and rotating the model so users get a better mental reference of the 3D object. The application could include other 2D representations, such as pie charts and stacked bar charts with different colored liquids.

Using PBF, which contains a relaxation parameter ε [30] in its fluid-fluid interaction calculations. This parameter is user-defined and stays constant during the entire simulation. Normally one would tweak this parameter based on factors such as fluid particle count. However, changes in the parameter are most noticeable in situations where many fluid particles are stacked vertically, as in our case with the cylindrical bars. If the ε value is too high, it will result in the fluid being more compressible, which is unwanted. If ε is too low, fluid particles behave chaotically or, worst-case, penetrate through boundary particles. This could be fixed by finding some equation based on bar size and particle count, but it would require further experimentation.

The narrative visualization aspect of the liquid conductor could also be worked on further. The application could be extended to improve this aspect, such as showcasing each 3D model before starting the simulation. This will improve the user's perception of each 3D model before liquidizing them. Some multi-stage animation could be implemented without altering the existing source code.

Changing the fluid rendering technique could improve the graphical quality, such as using a particle splatting method [1] or combining splatting and ray tracing [51]. Changing the rendering technique could improve accuracy when reading bar chart values, such as the recent work by Xu et al. [52] applies anisotropic transformations to the fluid particle spheres based on particle distribution to render more realistic fluid edges.

Chapter 8

Conclusions

In this thesis, we presented a novel approach to creating animated transitions from 3D models to 2D bar charts using fluid simulation. Our method uses morphological dilation to create a fluid container around the 3D model. Then seed fluid particles within it create a fluid-filled container, which can be poured into a bar chart during the transition. We demonstrated the effectiveness of our approach through several examples.

Our method provides several advantages over traditional techniques for creating animated transitions. By using fluid simulation, we can create realistic and visually appealing transitions without requiring the tedious manipulation of keyframes that traditional methods often demand. Our approach also reduces the time and effort required to create high-quality data visualizations while maintaining traditional techniques' effectiveness.

We also addressed several key challenges when creating animated transitions, such as the need for context retention, the problems of occlusion in 3D, and the limitations of using volume as a channel for encoding magnitude. Our approach overcomes these challenges and produces informative and engaging transitions.

Overall, our method represents a significant contribution to the field of data visualization and has the potential to enable a wide range of applications in industry, education, and research. Our approach will interest researchers and practitioners in computer graphics, visualization, and related fields. We look forward to seeing how it will be further developed and applied.

Glossary

compute shader a program run on the GPU, but is run outside the rendering pipeline.

ComputeBuffer Unity class for creation of GPU data buffers, to be used in Compute Shaders.

equatorial radius The radius of a planetary body at its equator.

mesh Part of a 3D model which contains vertex, triangle, and normal vector data.

RenderTexture Unity class, a texture that can be rendered to.

surface shader a program run on the GPU, that calculates the appropriate color and light for each point on a surface.

voxel *Volume element*, a single unit of space in a three-dimensional regular grid.

voxelization Process of converting a continuous mesh into a set of voxels of a set size.

List of Acronyms and Abbreviations

CAD Computer Assisted Drawing.

CPU Central Processing Unit.

GPU Graphics Processing Unit.

PBD Position-Based Dynamics.

PBF Position-Based Fluid.

SDF Signed Distance Function/Field.

SPH Smoothed Particle Hydrodynamics.

UML Unified Modeling Language.

Bibliography

[1] Bart Adams, Philip Dutré, and Toon Lenaert. Particle splatting: Interactive rendering of particle-based simulation data. 2006.

[2] Christian Basch. *Animated transitions across multiple dimensions for volumetric data*. na, 2011.

[3] Kenneth E Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, 1968.

[4] Markus Becker and Matthias Teschner. Weakly compressible sph for free surface flows. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 209–217, 2007.

[5] Richard A Becker and William S Cleveland. Brushing scatterplots. *Technometrics*, 29(2): 127–142, 1987.

[6] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D^3 data-driven documents. *IEEE transactions on visualization and computer graphics*, 17(12):2301–2309, 2011.

[7] Richard Brath. 3d infovis is here to stay: Deal with it. In *2014 IEEE VIS International Workshop on 3DVis (3DVis)*, pages 25–31. IEEE, 2014.

[8] Robert Bridson. *Fluid simulation for computer graphics*. CRC press, 2015.

[9] Andy Cockburn and Bruce McKenzie. An evaluation of cone trees. In *People and Computers XIV—Usability or Else! Proceedings of HCI 2000*, pages 425–436. Springer, 2000.

[10] Andy Cockburn and Bruce McKenzie. 3d or not 3d? evaluating the effect of the third dimension in a document management system. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 434–441, 2001.

[11] Andy Cockburn and Bruce McKenzie. Evaluating the effectiveness of spatial memory in 2d and 3d physical and virtual environments. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 203–210, 2002.

[12] Andrea Colagrossi and Maurizio Landrini. Numerical simulation of interfacial flows by smoothed particle hydrodynamics. *Journal of computational physics*, 191(2):448–475, 2003.

[13] Pierre Dragicevic, Anastasia Bezerianos, Waqas Javed, Niklas Elmqvist, and Jean-Daniel Fekete. Temporal distortion for animated transitions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2009–2018, 2011.

[14] Niklas Elmqvist and Philippas Tsigas. A taxonomy of 3d occlusion management for visualization. *IEEE transactions on visualization and computer graphics*, 14(5):1095–1109, 2008.

[15] Blender Foundation. Blender, 2023.
URL: `https://www.blender.org/`.

[16] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Design Patterns. Elements of reusable object-oriented software. *Design Patterns*, 1995.

[17] Sarah FF Gibson. Using distance maps for accurate surface representation in sampled volumes. In *Proceedings of the 1998 IEEE symposium on Volume visualization*, pages 23–30, 1998.

[18] Robert M Haralick, Stanley R Sternberg, and Xinhua Zhuang. Image analysis using mathematical morphology. *IEEE transactions on pattern analysis and machine intelligence*, (4):532–550, 1987.

[19] John C Hart. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12(10):527–545, 1996.

[20] Jeffrey Heer and George Robertson. Animated transitions in statistical data graphics. *IEEE transactions on visualization and computer graphics*, 13(6):1240–1247, 2007.

[21] Jian Huang, Roni Yagel, Vassily Filippov, and Yair Kurzion. An accurate method for voxelizing polygon meshes. In *IEEE symposium on volume visualization (Cat. No. 989EX300)*, pages 119–126. Ieee, 1998.

[22] Jessica Hullman, Steven Drucker, Nathalie Henry Riche, Bongshin Lee, Danyel Fisher, and Eytan Adar. A deeper understanding of sequence in narrative visualization. *IEEE Transactions on visualization and computer graphics*, 19(12):2406–2415, 2013.

[23] Autodesk Inc. Maya software, 2023.
URL: `https://www.autodesk.com/products/maya/overview`.

[24] Mihai F Ionescu and Klaus E Schauser. Optimizing parallel bitonic sort. In *Proceedings 11th International Parallel Processing Symposium*, pages 303–309. IEEE, 1997.

[25] Younghoon Kim and Jeffrey Heer. Gemini: A grammar and recommender system for animated transitions in statistical graphics. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):485–494, 2020.

[26] Younghoon Kim, Michael Correll, and Jeffrey Heer. Designing animated transitions to convey aggregate operations. In *Computer Graphics Forum*, volume 38, pages 541–551. Wiley Online Library, 2019.

[27] Marián Kireš. Archimedes' principle in action. *Physics education*, 42(5):484, 2007.

[28] Robert Kosara. Presentation-oriented visualization techniques. *IEEE computer graphics and applications*, 36(1):80–85, 2016.

[29] Markos Kyritsis, Stephen R Gulliver, Sonali Morar, and Robert Stevens. Issues and benefits of using 3d interfaces: visual and verbal tasks. In *Proceedings of the Fifth International Conference on Management of Emergent Digital EcoSystems*, pages 241–245, 2013.

[30] Miles Macklin and Matthias Müller. Position based fluids. *ACM Transactions on Graphics (TOG)*, 32(4):1–12, 2013.

[31] Andrian Marcus, Louis Feng, and Jonathan I Maletic. 3d representations for software visualization. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 27–ff, 2003.

[32] Andrew Vande Moere and Helen Purchase. On the role of design in information visualization. *Information Visualization*, 10(4):356–371, 2011.

[33] Joe J Monaghan. Smoothed particle hydrodynamics. *Annual review of astronomy and astrophysics*, 30(1):543–574, 1992.

[34] Joe J Monaghan and Jules B Kajtar. Sph particle boundary forces for arbitrary boundaries. *Computer physics communications*, 180(10):1811–1820, 2009.

[35] Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. Position based dynamics. *Journal of Visual Communication and Image Representation*, 18(2):109–118, 2007.

[36] Tamara Munzner. *Visualization analysis and design*. CRC press, 2014.

[37] Rick Parent. *Computer animation: algorithms and techniques*. Newnes, 2012.

[38] Inigo Quilez. distance functions, 2023.
URL: https://iquilezles.org/articles/distfunctions/.

[39] Scrawk. Pbd-fluid-in-unity. https://github.com/Scrawk/PBD-Fluid-in-Unity, 2022.

[40] Edward Segel and Jeffrey Heer. Narrative visualization: Telling stories with data. *IEEE transactions on visualization and computer graphics*, 16(6):1139–1148, 2010.

[41] P Kenneth Seidelmann, Brent A Archinal, Michael F A'hearn, Al Conrad, Guy J Consolmagno, Daniel Hestroffer, James L Hilton, GA Krasinsky, G Neumann, Jürgen Oberst, et al. Report of the iau/iag working group on cartographic coordinates and rotational elements: 2006. *Celestial Mechanics and Dynamical Astronomy*, 98:155–180, 2007.

[42] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings 1996 IEEE symposium on visual languages*, pages 336–343. IEEE, 1996.

[43] Ben Shneiderman. Why not make interfaces better than 3d reality? *IEEE Computer Graphics and Applications*, 23(6):12–15, 2003.

[44] Barbara Solenthaler and Renato Pajarola. Predictive-corrective incompressible sph. In *ACM SIGGRAPH 2009 papers*, pages 1–6. 2009.

[45] Johannes Sorger, Peter Mindek, Peter Rautek, Eduard Gröller, Graham Johnson, and Ivan Viola. Metamorphers: Storytelling templates for illustrative animated transitions in molecular visualization. In *Proceedings of the 33rd Spring Conference on Computer Graphics*, pages 1–10, 2017.

[46] Mark St. John, Michael B Cowen, Harvey S Smallman, and Heather M Oonk. The use of 2d and 3d displays for shape-understanding versus relative-position tasks. *Human Factors*, 43(1):79–98, 2001.

[47] John T Stasko and Joseph F Wehrli. Three-dimensional computation visualization. In *Proceedings 1993 IEEE Symposium on Visual Languages*, pages 100–107. IEEE, 1993.

[48] LLC Tableau Software. Tableau: Business intelligence and analytics software, 2023.
URL: https://www.tableau.com.

[49] Unity Technologies. Unity real-time development platform, 2023.
URL: https://unity.com/.

[50] Alan Watt. *Fundamentals of three-dimensional computer graphics*. Addison-Wesley Longman Publishing Co., Inc., 1990.

[51] Xiangyun Xiao, Shuai Zhang, and Xubo Yang. Real-time high-quality surface rendering for large scale particle-based fluids. In *Proceedings of the 21st ACM siggraph symposium on interactive 3D graphics and games*, pages 1–8, 2017.

[52] Yanrui Xu, Yuanmu Xu, Yuege Xiong, Dou Yin, Xiaojuan Ban, Xiaokun Wang, Jian Chang, and Jian Jun Zhang. Anisotropic screen space rendering for particle-based fluid simulation. *Computers & Graphics*, 110:118–124, 2023.

[53] Cha Zhang and Tsuhan Chen. Efficient feature extraction for 2d/3d objects in mesh representation. In *Proceedings 2001 International Conference on Image Processing (Cat. No. 01CH37205)*, volume 3, pages 935–938. IEEE, 2001.