Olivier M. Pachoud

BACHELOR OF SCIENCE

GAMES PROGRAMMING

What are the challenges of representing a dynamic implicit fluid in a real-time ray tracing pipeline using the DXR API?

Study based on a Smoothed Particle Hydrodynamics (SPH) simulation.

Module 6GST0XF10x 0325 - Bachelor thesis

Elias Farhan, Nicolas Siorak.

Student Number: # 12-136093

SAE Institute - Geneva

18.07.2025

Number of words: 11’704

I, Olivier Pachoud, certify having personally written this Bachelor thesis. I also certify that I have not resorted to plagiarism and have conscientiously and clearly mentioned all borrowings from others.

Echallens, 18 July 2025

O. Pachoud

Foreword

Since my final years in school, I have known that I wanted to turn my passion for video games into a career. The idea of creating interactive virtual worlds that could engage and inspire people deeply motivated me, and it naturally led me to pursue studies in game programming.

During this journey, I discovered a strong interest in computer graphics. I was drawn to the unique blend of visual creativity, artistic expression, and technical complexity that this field offers. The ability to craft striking images and believable environments through code quickly became a central focus of my academic and personal growth.

This bachelor project reflects that passion. By exploring real-time ray tracing, I aim to push my skills further in a domain that continues to shape the future of graphics rendering. More than just a technical challenge, this work is a way for me to express my enthusiasm for the discipline and contribute meaningfully to the world of interactive visual experiences.

Acknowledgment

I would like to express my sincere gratitude to Nicolas Siorak for his guidance and support throughout this bachelor project. His feedback, our insightful discussions, and the motivation he provided over the course of these six months were truly valuable.

I also want to thank Elias Farhan for his programming courses, and in particular the Computer Graphics module in OpenGL, which played a key role in helping me discover what has since become my professional passion.

Finally, a warm thank you to Constantin Verine, whose enthusiasm for the project was a great source of motivation. Collaborating with him throughout this process was both enriching and inspiring.

Abstract

This thesis explores real-time rendering techniques for particle-based fluid simulations using the DirectX Ray tracing (DXR) pipeline. Two rendering methods are compared: volumetric ray marching using a custom intersection shader, and surface reconstruction via the marching cubes algorithm with dynamic updates of the bottom-level acceleration structures (BLAS).

Both techniques use DXR to simulate key optical effects of fluids, such as reflection, refraction, and absorption, through recursive ray tracing. A 3D scene was developed to implement and compare both methods based on the same SPH simulation data.

The evaluation focuses on visual quality, performance (measured in frame time, in milliseconds), and implementation complexity. It includes a detailed analysis of the ray tracing pass duration and geometry memory consumption to assess the scalability of each method. This evaluation is supported by a review of the state of the art and interviews with industry professionals to contextualize the work within current real-time graphics practices.

Keywords: Ray tracing, DXR, Real-time, Fluid Rendering, Implicit Volume, Surface Reconstruction.

Table of Contents

Glossary

Rasterization

The process of converting geometric primitives (such as triangles) into pixels on a 2D image. It is a core stage of the traditional graphics pipeline, where each triangle is projected onto screen space and transformed into fragments that are later shaded and written to the framebuffer.

Vertex Buffer

A GPU-side data structure that stores a list of vertices, including attributes such as positions, normals, and texture coordinates. Vertex buffers are used to define the geometry of a mesh and are consumed by the graphics pipeline during rendering.

Graphics Pipeline

A sequential set of stages that a scene passes through to be rendered on screen. In APIs like DirectX or Vulkan, it typically includes vertex processing, primitive assembly, rasterization, fragment shading, and final image composition in the frame-buffer.

Graphics Pass

A discrete rendering stage in which a subset of the scene is processed with specific shaders and resources. Graphics passes are used to organize rendering into logical steps such as depth pre-pass, lighting, or post-processing.

Mesh

A 3D object representation composed of a collection of vertices connected into primitives, usually triangles. Meshes are the fundamental building blocks of geometry in a 3D scene and are rendered by the GPU.

Compute Shader

A programmable GPU stage designed for general-purpose parallel computation outside the fixed-function graphics pipeline. Compute shaders are commonly used for tasks such as physics simulation, texture generation, or data processing, offering flexible memory access and thread control.

Particle-Based Simulation

Particle-based simulation is a numerical modeling method that represents a complex physical system using a finite set of particles. Each particle carries physical properties such as mass, position, velocity, and density, and interacts with neighboring particles according to predefined laws. This technique is commonly used to simulate fluids, plasmas, or molecular systems.

SPH (Smoothed Particle Hydrodynamics)

SPH is a particle-based simulation method used to model fluid dynamics and other continuum phenomena. It approximates physical quantities (such as pressure and density) by smoothing values over neighboring particles using kernel functions. SPH is particularly suited for simulating incompressible fluids, splashes, and free-surface interactions in both offline and real-time environments.

PBR (Physically-Based Rendering)

PBR refers to a rendering approach that models the interaction between light and materials based on physical laws. It uses energy-conserving reflectance models, realistic material parameters (such as albedo, roughness, and metallicity), and often relies on image-based lighting. The goal of PBR is to produce visually consistent and plausible results under varying lighting conditions.

Isosurface

An isosurface is a 3D surface defined by the set of points at which a scalar field reaches a constant specified value. In practice—particularly in fluid rendering from particle-based simulations—this surface is often approximated at locations where the scalar field exceeds a certain threshold, such as a minimum density above which the fluid becomes visually significant.

Depth Map

In 3D computer graphics and computer vision, a depth map is an image or image channel that stores information about the distance between scene surfaces and a specific viewpoint. The concept is closely related to the depth buffer, Z-buffer, and Z-depth techniques. The "Z" typically refers to the camera’s view axis in a right-handed coordinate system, and not necessarily to the absolute Z-axis of the world space.

1. Introduction

Fluids are highly complex physical media that remain only partially understood by physicists, and whose study is now considered one of the major scientific challenges of the 21st century (The Conversation, 2023). Accurately reproducing their behaviour in a computer simulation is therefore a significant undertaking. In computer graphics, their visual representation is equally demanding: their dynamic behaviour, such as deformable volumes, and their optical properties, such as transparency, reflection, and refraction, require advanced rendering techniques to achieve realistic results.

Traditional 3D rendering methods commonly used in real-time simulations often fall short when it comes to convincingly representing deformable bodies like fluids and their complex interactions with light. These visual phenomena typically require computationally intensive algorithms such as ray tracing and the whole scene geometry data to produce visually accurate results.

Historically limited to offline rendering industries such as film production, ray tracing has recently become a viable option for real-time applications. This transition has been enabled by the emergence of NVIDIA RTX GPUs, equipped with RT Cores dedicated to accelerating and managing ray tracing computations, thereby facilitating the integration of this technology into real-time rendering pipelines (Legouge, 2024).

Although the necessary hardware is now widely available, and APIs such as DirectX 12 Ray tracing (DXR) provide simplified access to optimized ray tracing architectures, the performance of real-time rendering remains heavily dependent on the complexity of the 3D scene. The challenge lies in maintaining a delicate balance between the visual accuracy required to capture the subtle behaviour of fluids and the performance constraints needed to sustain high frame rates in interactive environments.

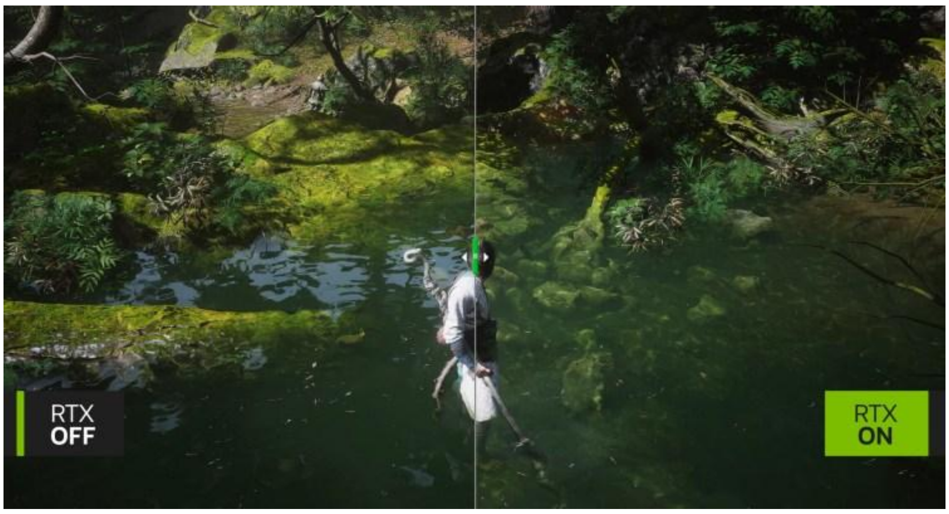


Figure 1 Difference in water rendering in Black Myth Wukong (Game Science, 2024) without ray tracing (left) and with ray tracing (right).

1.1 Problematic

Unlike rasterization, which only processes surface visible from the camera's point of view, ray tracing must simulate the full trajectory of rays through the entire 3D scene, including interactions with occluded objects. To avoid prohibitively expensive computations, DXR relies on a hierarchical acceleration structure (BVH), as illustrated in Figure 2, to optimize intersection tests between rays and scene geometry. This hierarchy consists of two levels:

- BLAS (Bottom-Level Acceleration Structure), which defines the internal structure of an object (typically a triangle mesh) and optimizes ray-triangle intersections
- TLAS (Top-Level Acceleration Structure), which organizes the instances of these BLAS in the global scene space

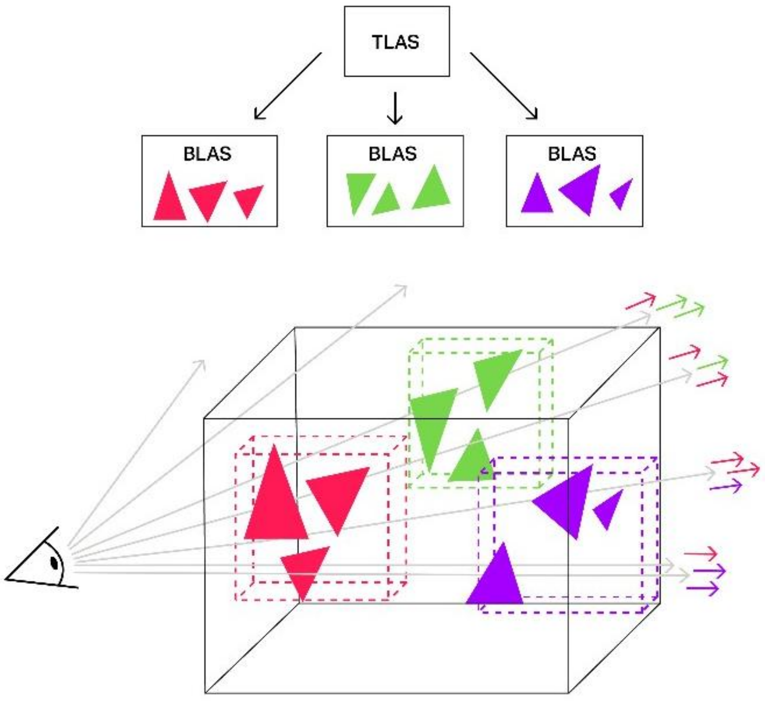


Figure 2 Diagram of the acceleration structure from DXR (Hong & Beets, 2020).

Thanks to this two-level hierarchy, DXR can launch millions of rays while maintaining reasonable performance, provided that all objects are represented using explicit geometry, i.e., triangle meshes that can be stored in BLAS. While this system is highly effective for rigid, well-defined surfaces, it poses a major challenge for implicit fluids, which by nature lack any directly accessible geometric surface. Without such a surface, rays cannot interact with the fluid, making its integration into DXR particularly complex if not impossible without prior surface reconstruction.

The core focus of this work is not limited to the realistic lighting effects applied to fluids, but also lies in the structural and technical constraints involved in representing such fluids within a real-time ray tracing pipeline using the DXR API. In the case of dynamic implicit fluids, whose surfaces are neither explicitly defined nor temporally stable, it becomes essential to find a way to make these surfaces visible and traceable by rays, while maintaining performance suitable for real-time rendering.

This thesis aims to identify, through a particle-based simulation, the key issues involved in integrating such fluid representations into an acceleration structure compatible with DXR, with the goal of enabling realistic light interaction within a real-time ray tracing pipeline.

1.2 Announcement of the Plan

This exploratory project begins with a historical and technical overview of fluid rendering and ray tracing techniques, highlighting the key challenges they pose within the real time rendering and interactive media industries. This review aims to establish a solid understanding of the theoretical and practical constraints associated with the visual representation of dynamic, implicit fluids.

In the second part, these insights will be confronted with real-world practices through a qualitative analysis. This includes interviews with industry professionals and a case study of an existing project, offering a grounded perspective on how developers make decisions under production constraints.

The final part of this work consists of a practical implementation, where a test protocol is used to evaluate the performance cost of using DXR for rendering dynamic, implicit fluids. While focused on frame time measurements, this analysis directly addresses the broader question of whether ray tracing can realistically support fluid representation in real-time contexts.

2. Challenges and Advances in Ray tracing and Particle-Based Fluid Rendering

Ray tracing, although only recently popularized by hardware advancements, has been studied for decades for its ability to produce photorealistic images. Fluids, on the other hand, present a major challenge in computer graphics due to both their physical and visual complexity. Today, these two domains are converging: ray tracing offers a promising solution for accurately representing the optical and dynamic properties of fluids.

In this chapter, we will define the key concepts of the topic, review the evolution of ray tracing and fluid rendering techniques, and analyse the challenges that emerge from their convergence in the context of real-time rendering.

2.1 Definitions

Ray casting:

According to the book *Ray tracing Gems* (Shirley, P., et al. 2019) Ray casting is considered as:

the process of finding the closest, or sometimes just any, object along a ray. A ray leaves the camera through a pixel and travels until it hits the closest object. As part of shading this hit point, a new ray could be cast toward a light source to determine if the object is shadowed.

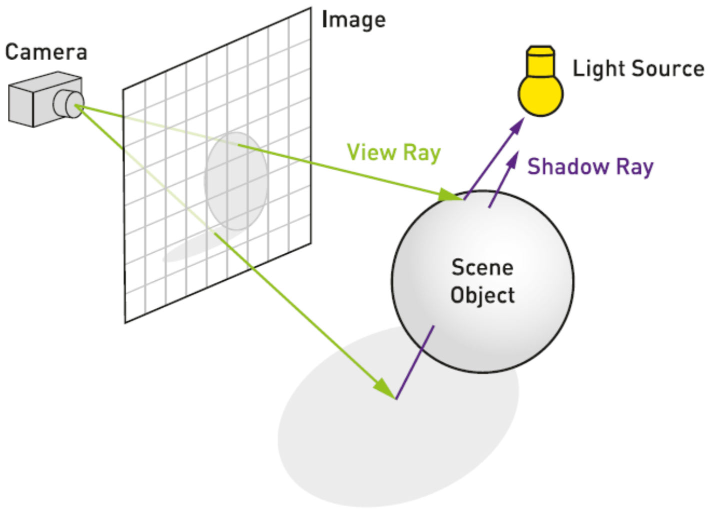


Figure 3 Ray casting diagram (Haines & Akenine-Möller, 2019).

Ray tracing:

According to the book *Ray tracing Gems* (Shirley, P., et al. 2019) Ray tracing is considered as:

A process which uses the ray casting mechanism to recursively gather light contributions from reflective and refractive objects. For example, when a mirror is encountered, a ray is cast from a hit point on the mirror in the reflection direction. Whatever this *reflection ray* intersects affects the final shading of the mirror. Likewise, transparent or glass objects may spawn both a reflection and a *refraction ray*.

This process occurs recursively, with each new ray potentially spawning additional reflection and refraction rays. Recursion is usually given some cutoff limit, such as a maximum number of bounces. This tree of rays is evaluated back up its chain to give a color. As before, each intersection point can be queried whether it is shadowed by casting a ray toward each light source.

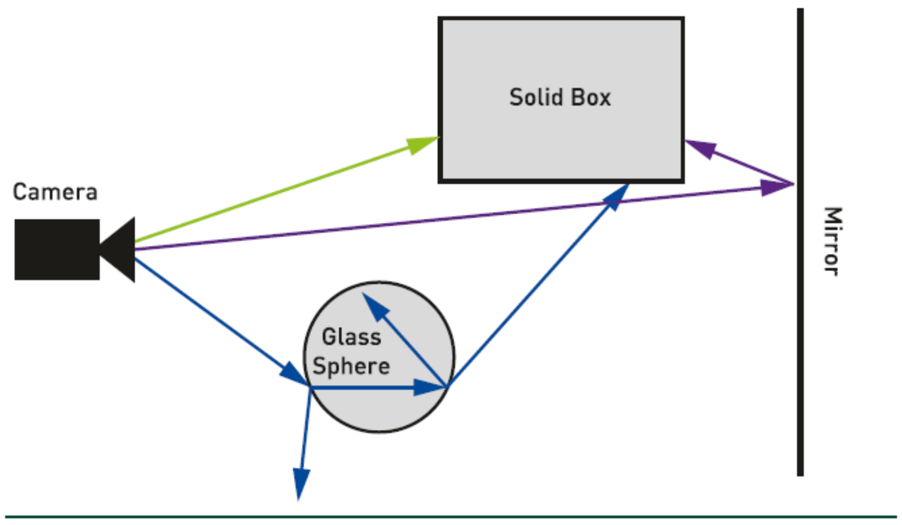


Figure 4 Ray tracing diagram (Haines & Akenine-Möller, 2019).

Path tracing

Path tracing is a rendering algorithm that simulates the global illumination of a scene by randomly sampling the many possible light paths that contribute to a pixel's final color. Unlike basic ray tracing, which typically handles only direct lighting and simple reflections or refractions, path tracing accounts for complex light interactions such as soft shadows, caustics, color bleeding, and indirect illumination.

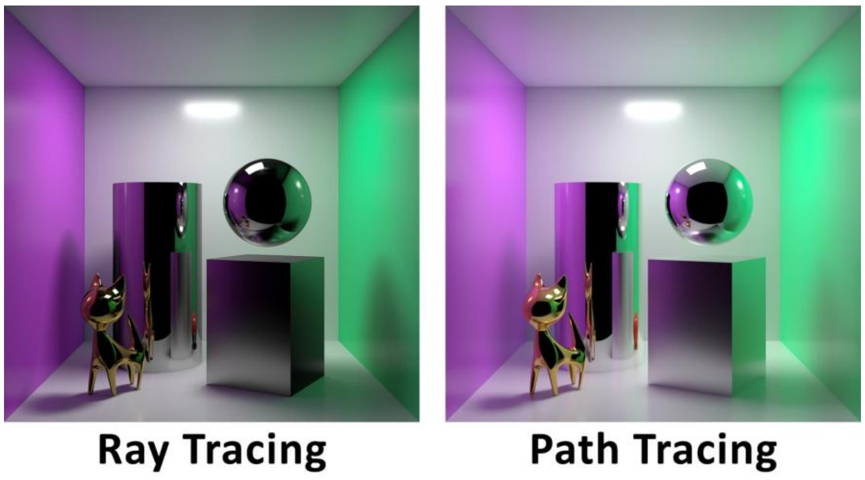


Figure 5 Comparison between ray tracing and path tracing.
Image posted by Stampf on the Unreal Engine Forum (2022)

Reflection:

Optical phenomenon in which part of the light bounces off the surface of a material and returns to the original medium. There are two main types of reflection.

Specular reflection, which follows a specific direction (as on a mirror) and diffuse reflection, where light is scattered in all directions (as on a rough surface).

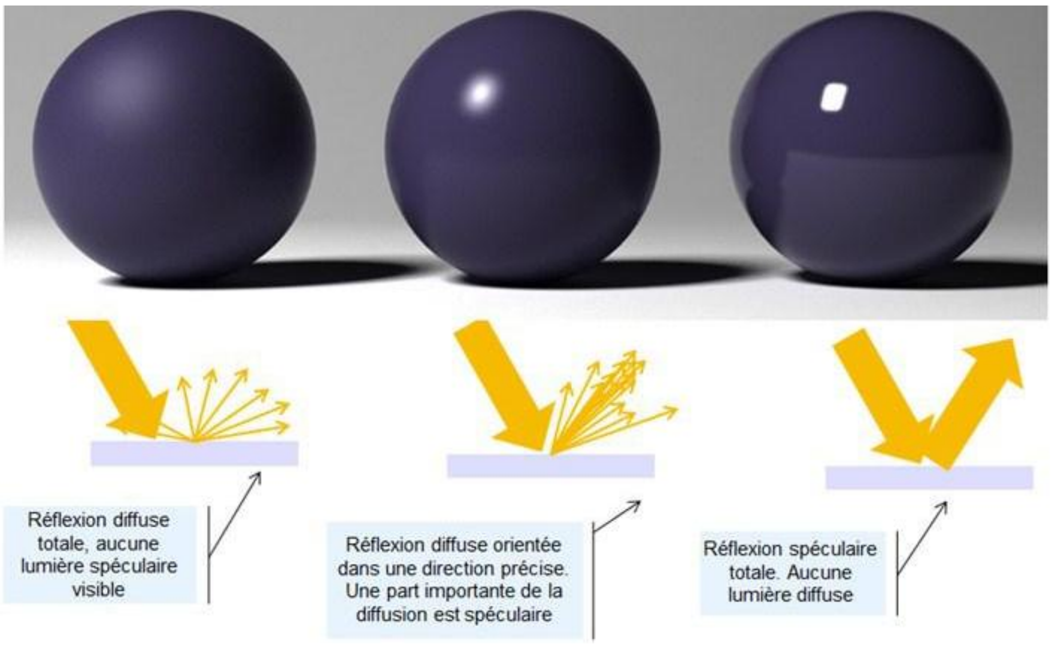

Réflexion diffuse totale, aucune lumière spéculaire visible
Réflexion diffuse orientée dans une direction précise. Une part importante de la diffusion est spéculaire
Réflexion spéculaire totale. Aucune lumière diffuse

Figure 6 Representation of the different types of reflections. (BELSPO, n.d.)

Refraction:

Optical phenomenon where light changes direction as it passes from one medium to another with a different refractive index (for example, from air to water or glass). This change in direction is caused by the variation in the speed of light between the two media.

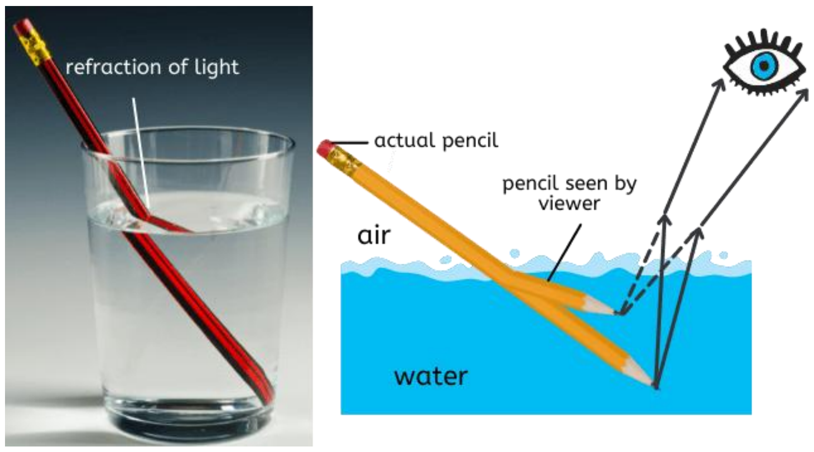


Figure 7 Refraction of light between air and water (Singh.Puja, 2021).

Dielectric Material:

In physics, a dielectric material is a pure electrical insulator, meaning it does not conduct electricity. However, in computer graphics, the term “dielectric” is used more broadly and pragmatically.

It refers to non-metallic and transparent materials that exhibit refraction, reflection, and sometimes absorption of light, regardless of their actual electrical conductivity. For instance, although impure water (as found in nature) can conduct electricity, it is still treated as a dielectric material in a rendering engine.

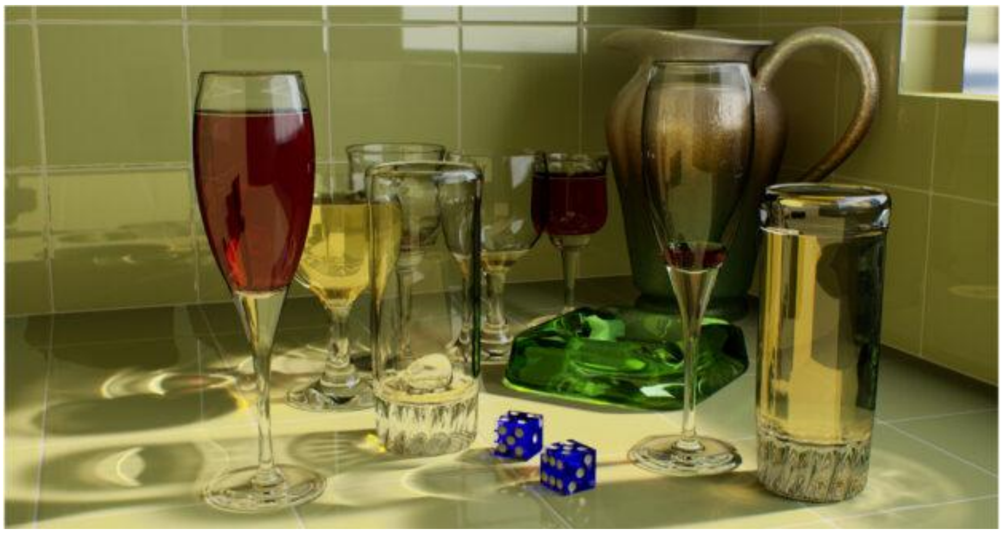


Figure 8 Dielectric material rendering (Ouyang & Yang, 2023)

2.2 Synthesis of Sources and Evolution of Practices

In this section, we will trace the evolution of ray tracing, explore the approaches used to model fluids, and highlight the scientific and technical challenges involved in achieving realistic real-time fluid rendering. We will also reference several video games, software and frameworks that integrate fluid simulation techniques, in order to better situate the current state of practice in both industry and research.

2.2.1 The evolution of ray tracing from the 16th century to the present day

Although often seen as a recent innovation, ray tracing has its roots in much older theoretical and artistic work. David Luebke, Vice President of Graphics Research at NVIDIA, points out that as early as the 16th century, Albrecht Dürer was already experimenting with geometric projection techniques to depict perspective, laying the groundwork for a conceptual approach to simulating light (Caulfield, 2022).

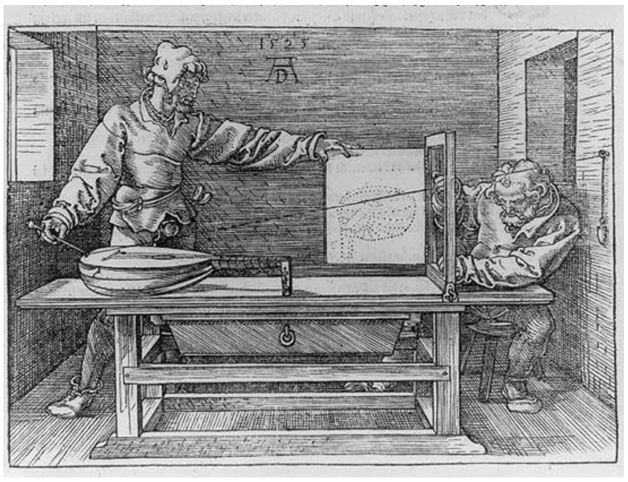


Figure 9 Albrecht Dürer's perspective machine (Caulfield, 2022)

However, it wasn't until 1969 that this idea found a practical application in computer graphics, when Arthur Appel used ray casting to compute visibility and shadows.

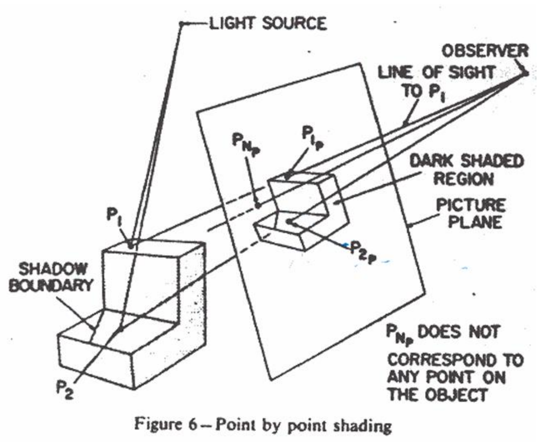


Figure 10 Arthur Appel's ray casting method for shading solids (Appel, 1969)

In 1979, Turner Whitted introduced recursive ray tracing, which enabled the simulation of reflections, refractions, and shadows, ushering in a new era of image realism.

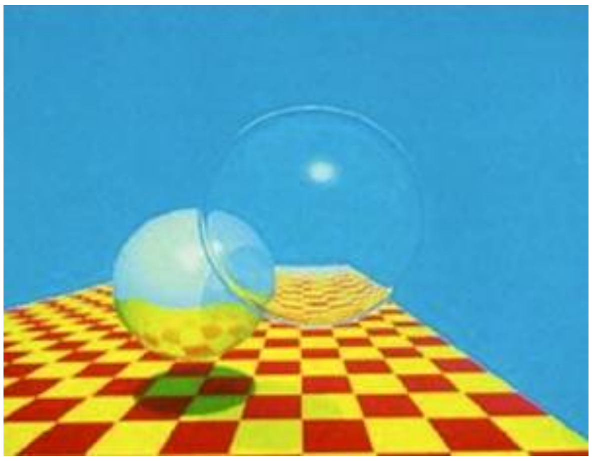


Figure 11 Whitted ray tracing rendering (Whitted, 1979).

Later, in 1986, Jim Kajiya formulated the rendering equation and introduced the path tracing algorithm, providing a physically based model for light transport. These foundational contributions established the basis of photorealistic rendering.

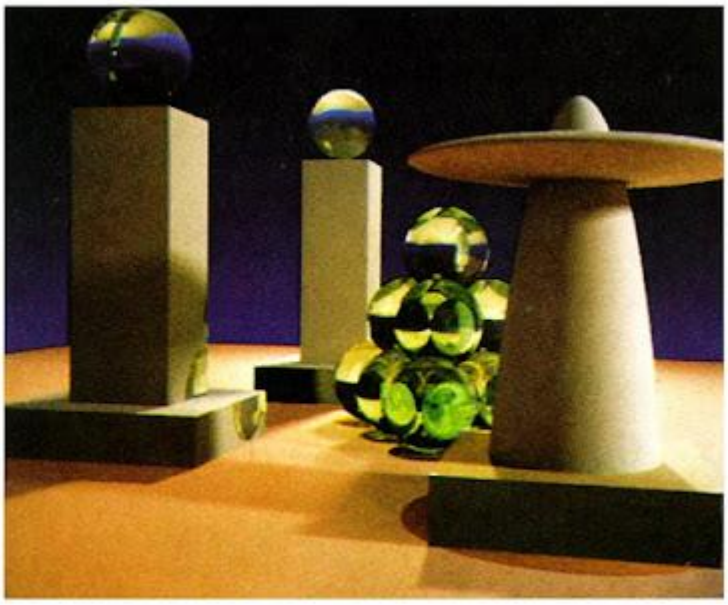


Figure 6. A sample image. All objects are neutral grey. Color on the objects is due to caustics from the green glass balls and color bleeding from the base polygon.

Figure 12 First path tracing rendering (Kajiya, 1986).

In the film industry, ray tracing began gaining traction in the late 1990s, notably in *A Bug's Life* (1998). Over time, path tracing emerged as the standard, particularly with *Monster House* (2006), the first feature film fully rendered using this technique via the Arnold rendering engine. Today, films like *Avatar: The Way of Water* (2022) continue to push the boundaries of visual realism, with renderings requiring tens of millions of thread-hours.

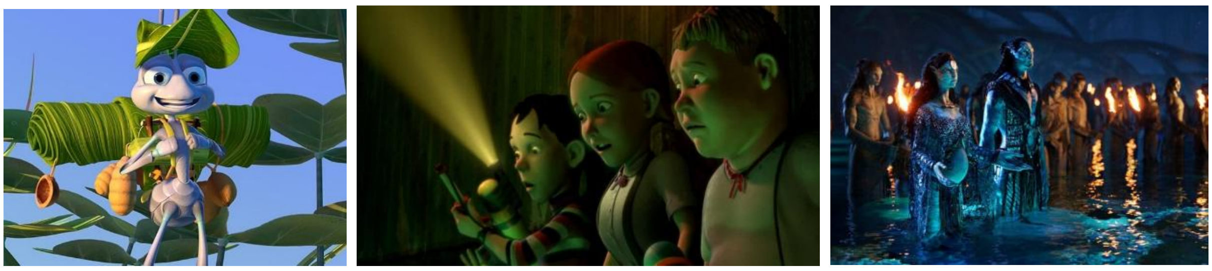


Figure 13 Left: A Bug's Life (1998). Center: Monster House (2006). Right: Avatar: The Way of Water (2022).

The video game industry followed a similar trajectory. With the rise of modern GPUs and RTX hardware, real-time ray tracing became feasible. Just like in film, early implementations in games were partial, gradually leading to more sophisticated lighting.

In 2018, two major developments marked a turning point: the introduction of DirectX 12 Ray tracing (DXR) by Microsoft, and the launch of the first generation of NVIDIA RTX graphics cards, featuring dedicated RT Cores for ray tracing operations. Shortly after, at GDC, Epic Games, together with NVIDIA and ILMxLAB, presented a technical demo titled *Reflections* (2018), showcasing characters from Star Wars: The Last Jedi in a fully raytraced real-time environment.

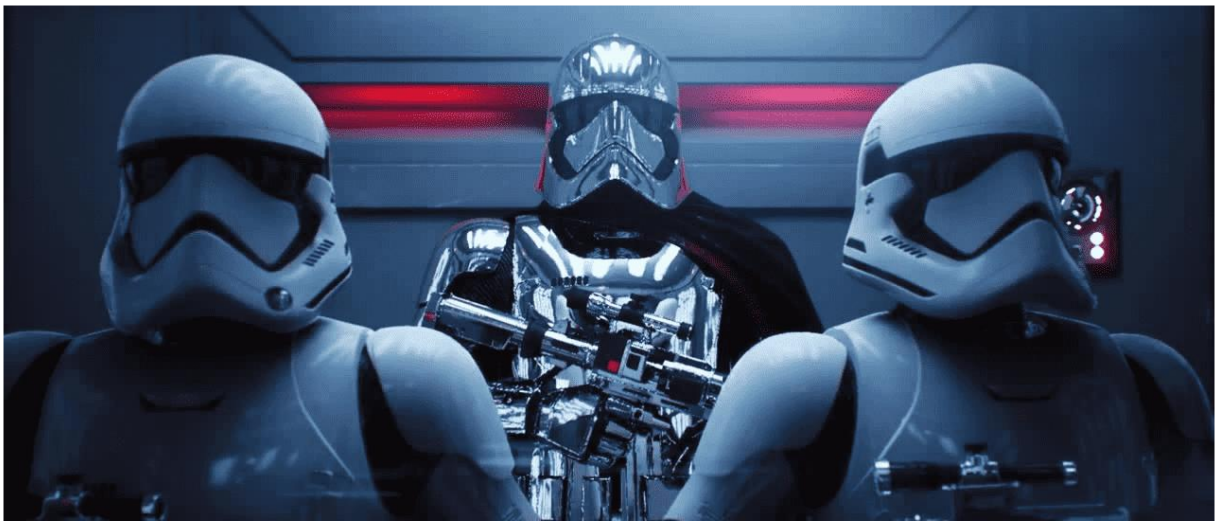


Figure 14 Reflections technical demo from Nvidia, Epic Games, and ILMxLAB (2018)

That same year, *Battlefield V* (DICE, 2018) became the first AAA video game to implement real-time ray tracing, specifically for dynamic reflections on surfaces, in combination with traditional rasterized rendering. This approach echoed early uses of ray tracing in cinema, where it was initially applied to specific effects only.

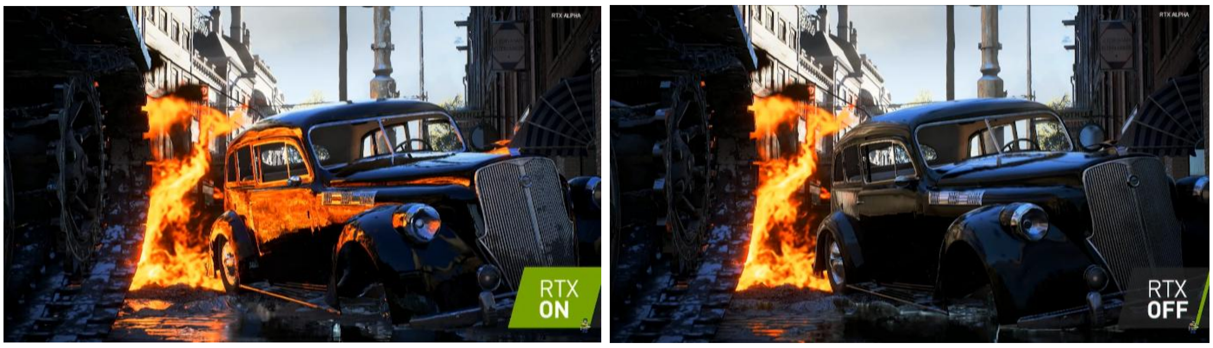


Figure 15 Comparison of rendering reflections with ray tracing (left) and without ray tracing (right) in Battlefield V (DICE, 2018)

A major milestone was reached with the Overdrive mode of *Cyberpunk 2077 (CD Projekt RED, 2023)*, in which path tracing was used to its full extent. Leveraging technologies such as DLSS 3, AI-driven frame generation, and denoising, the game achieves full global illumination in a dense, open world, in real time (Burnes, 2023). This accomplishment represents a significant step in the evolution of real-time rendering, showing that path tracing, once restricted to offline production, can now be considered for interactive experiences, provided that high-end graphics hardware is available.

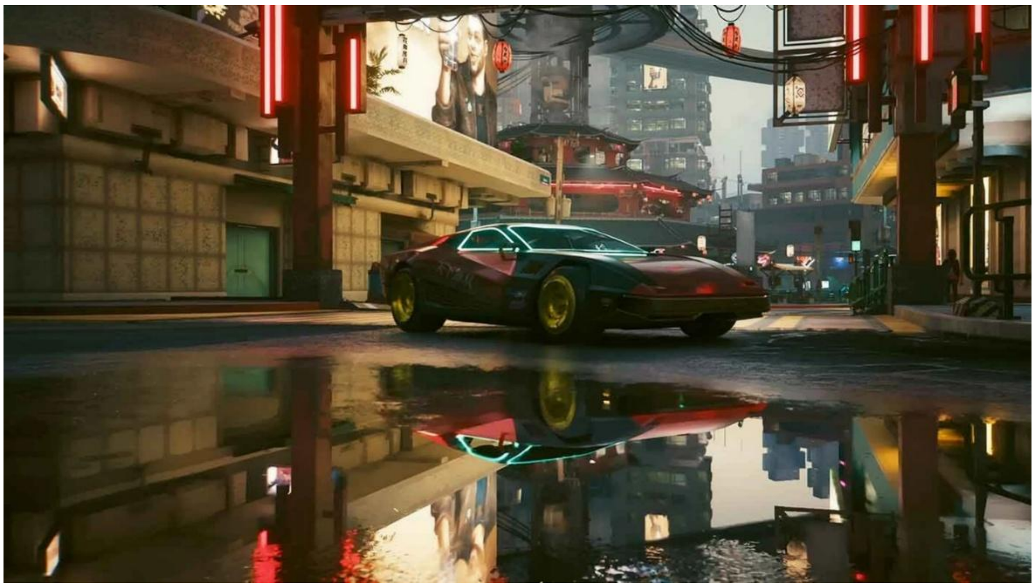


Figure 16 Image of the override mode of Cyberpunk 2077 (CD Projekt RED, 2023).

2.2.2 Surface Extraction Techniques from Particle-Based Simulations

To render a particle-based simulation, it is necessary to extract a surface from the simulation data. In the field of computer graphics, three main approaches have emerged for this purpose.

2.2.1.1 Screen Space rendering of particles.

Screen-space rendering of particle-based fluids, popularized by Simon Green (NVIDIA) at GDC 2010 and based on work by van der Laan et al. (2009), approximates fluid surfaces directly in image space. Each particle is rendered as a camera-facing quad using a radially symmetric kernel. These sprites contribute to a depth (and optionally thickness) buffer via additive blending, which is then smoothed using a bilateral filter. A normal map is derived for shading with physically-based models in a deferred pipeline.

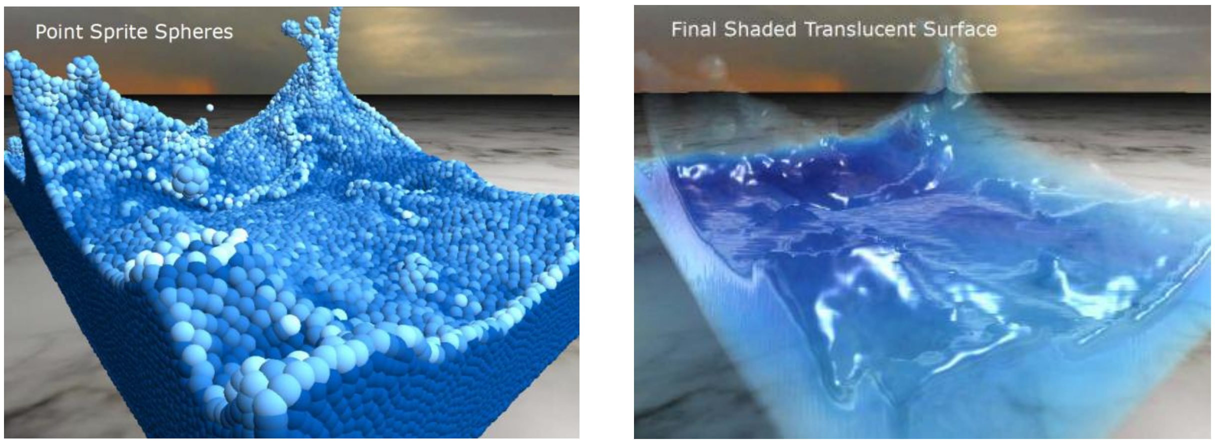


Figure 17 Image of the talk of Simon Green during the GDC 2010 showing the transformation of sprite points into a volume of fluid

This method is efficient for real-time rendering and large particle counts, avoiding the need for full 3D mesh reconstruction. However, it suffers from limitations such as missing back-facing geometry, artifacts at grazing angles, and poor reconstruction of thin features.

To improve surface continuity, (Xu et al., 2022). (2022) introduced anisotropic kernels that orient and stretch particles based on local geometry (see Figure 18).

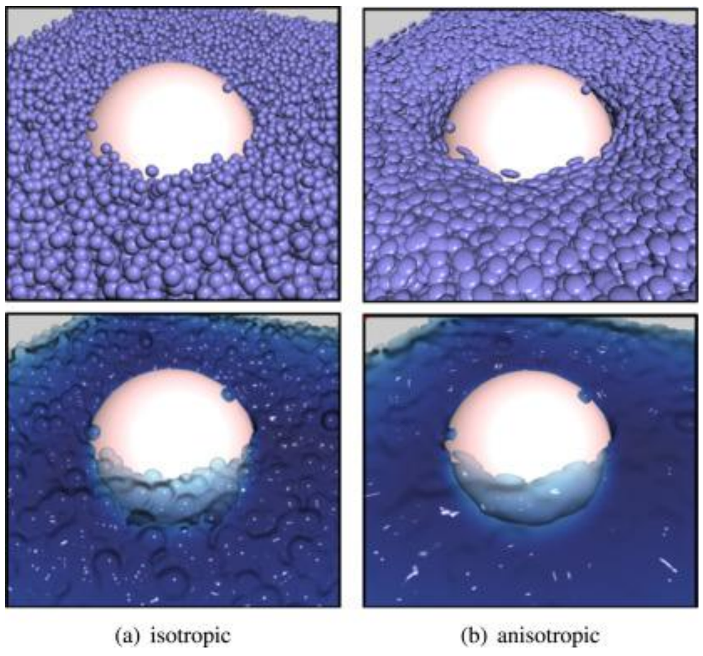


Figure 18 Comparison between isotropic rendering et anisotropic rendering (Xu, 2018)

Later, Ihmsen et al. (2012) extended this screen-space approach with visual effects like spray, foam, and bubbles to enhance realism without sacrificing performance.

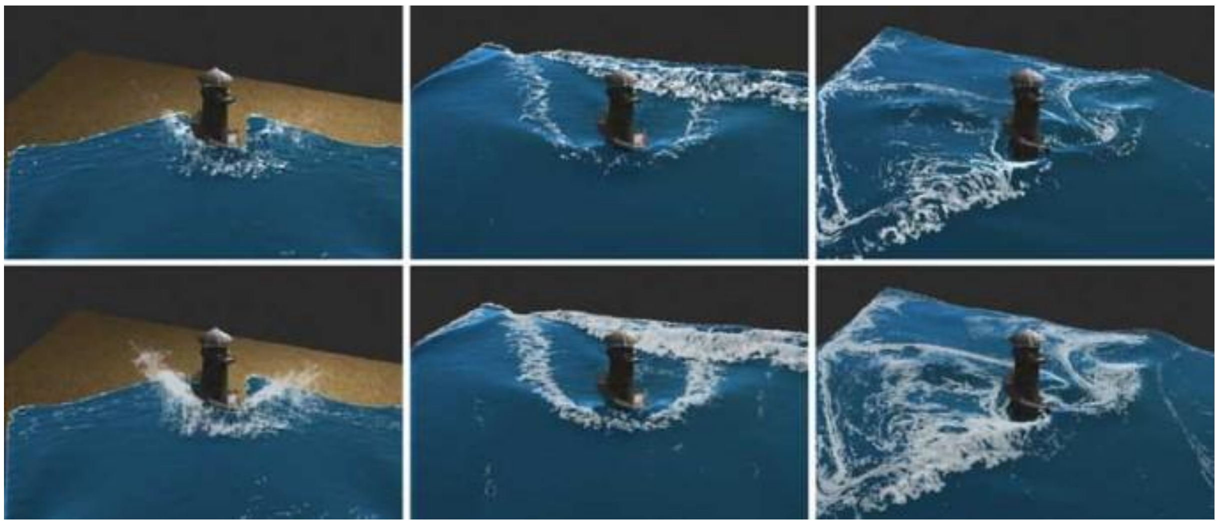


Figure 19 Images from the research paper "Unified Spray, Foam and Air Bubbles for Particle-Based Fluids" (Ihmsen et al., 2012).

2.2.1.2 The Marching Cubes algorithm

Introduced by William E. Lorensen and Harvey E. Cline in 1987 (ACM SIGGRAPH), the Marching Cubes algorithm is a foundational technique for extracting a polygonal mesh from a 3D scalar field. It is particularly relevant for visualizing isosurfaces in volumetric data, such as density fields in particle-based fluid simulations.

The method works by subdividing the simulation domain into a uniform 3D grid of cubes. For each cube, the algorithm evaluates the scalar field at its eight corners to determine how the isosurface intersects the cube's edges. Based on a precomputed lookup table of 256 configurations, it generates a set of triangles that approximate the local surface geometry.

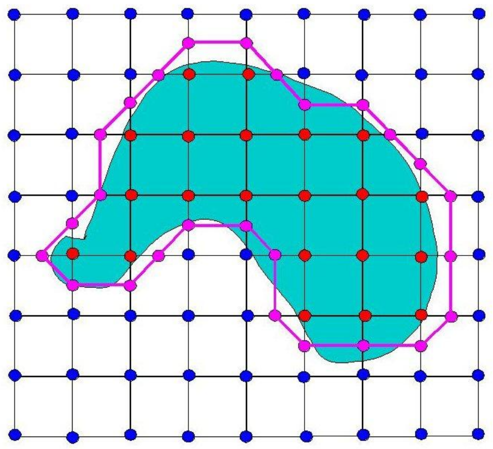


Figure 20 Example of a 3D voxel grid structure used to extract an isosurface from scalar fields (Carleton College, n.d.)

While effective and relatively simple to implement, the algorithm has notable limitations. A low grid resolution can lead to blocky or noisy surfaces, while a high resolution increases computation and memory costs, even on modern GPUs. For fluid rendering, the resulting mesh may appear rigid or artificial, especially when trying to capture smooth curves or thin structures. Furthermore, the output mesh is typically opaque, requiring additional shading or rendering techniques to simulate transparency, refraction, or subsurface light transmission.

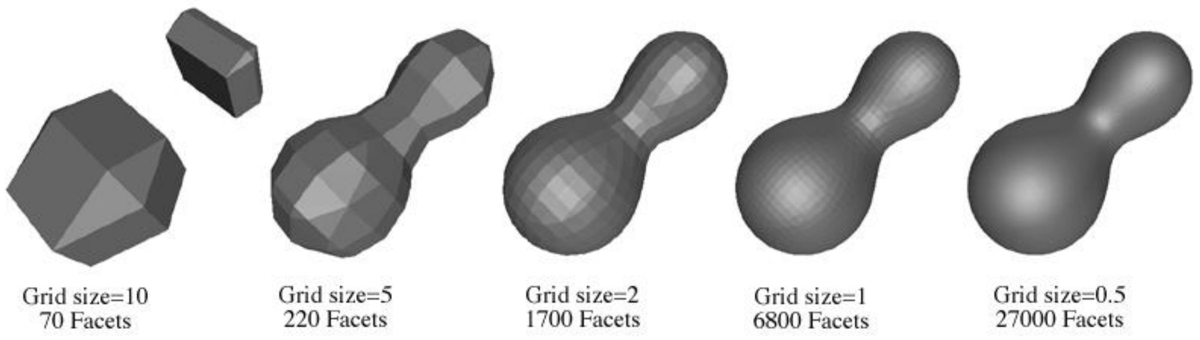


Figure 21 Comparison of surface detail generated at different grid resolutions (Bourke, n.d.)

Despite these challenges, Marching Cubes remains widely used. For example, the FluidX3D project by Dr. Moritz Lehmann (2022) employs this algorithm to convert particle-based fluid simulations into photorealistic meshes for rendering.

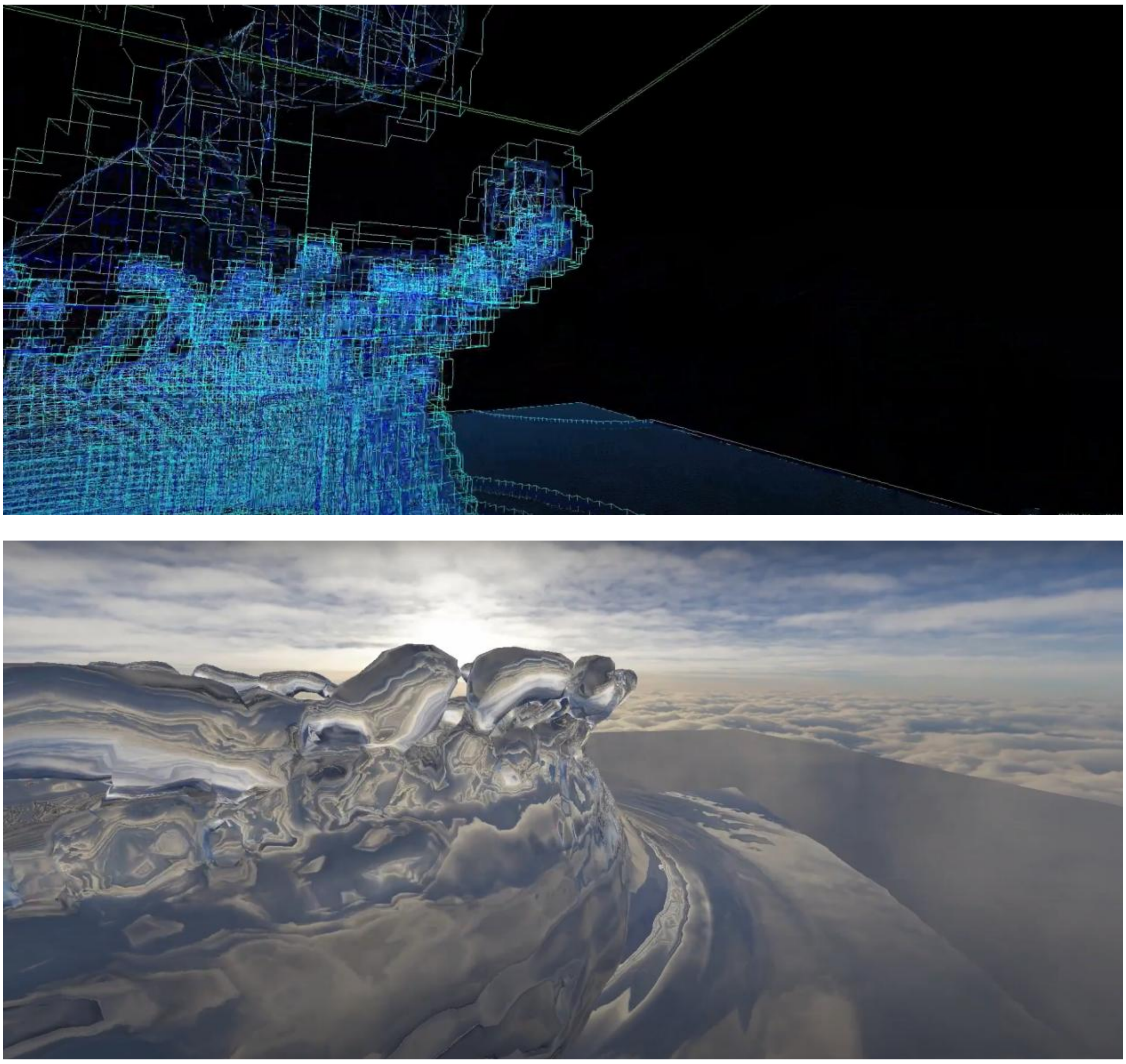


Figure 22 Mesh generated from particle-based fluid data using the Marching Cubes algorithm in the FluidX3D (Lehmann, 2022).

2.2.1.3 Ray Marching.

The third approach does not attempt to extract a mesh from the simulation, but instead renders the fluid directly as a volume using its underlying density field. This is achieved through a volumetric ray marching algorithm, which samples the 3D density grid along view rays cast from the camera.

At each step along the ray, the algorithm evaluates the local density and accumulates both color and opacity contributions. This iterative process reconstructs the visual appearance of the fluid on-the-fly, capturing its internal structure, gradients, and transparency effects without generating explicit geometry.

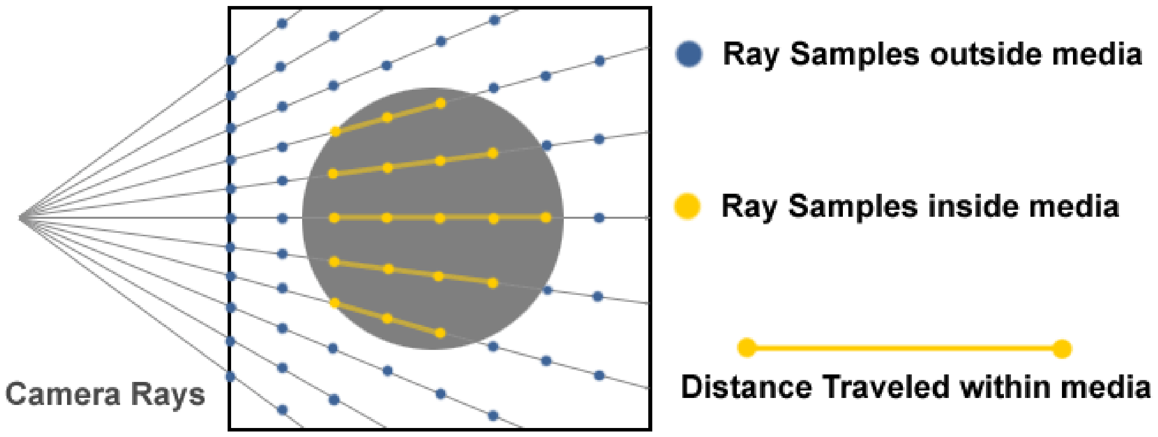


Figure 23 Ray marching diagram (Brucks, 2016)

This method captures fine volumetric detail, particularly in low-density regions, and is visually more faithful than mesh-based techniques. However, it is computationally expensive due to the high number of sampling steps required for artifact-free results.

Earlier foundational work on this topic can also be found in GPU Gems 3 (Crane et al., 2007), which explores strategies for simulating and rendering volumetric fluids in real-time on the GPU.

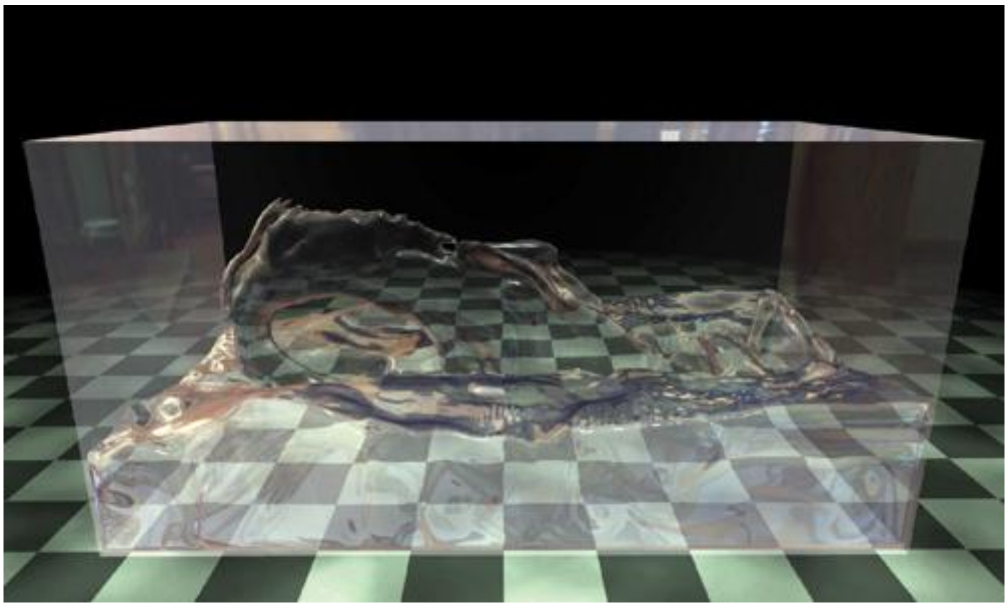


Figure 24 Ray marching rendering from GPU Gems 3 (Crane et al., 2007).

More recently, Sebastian Lague implemented this method in a Unity-based fluid simulation prototype (2024), illustrating its potential for real-time visual feedback. He even recoded raytracing software specifically for his scene

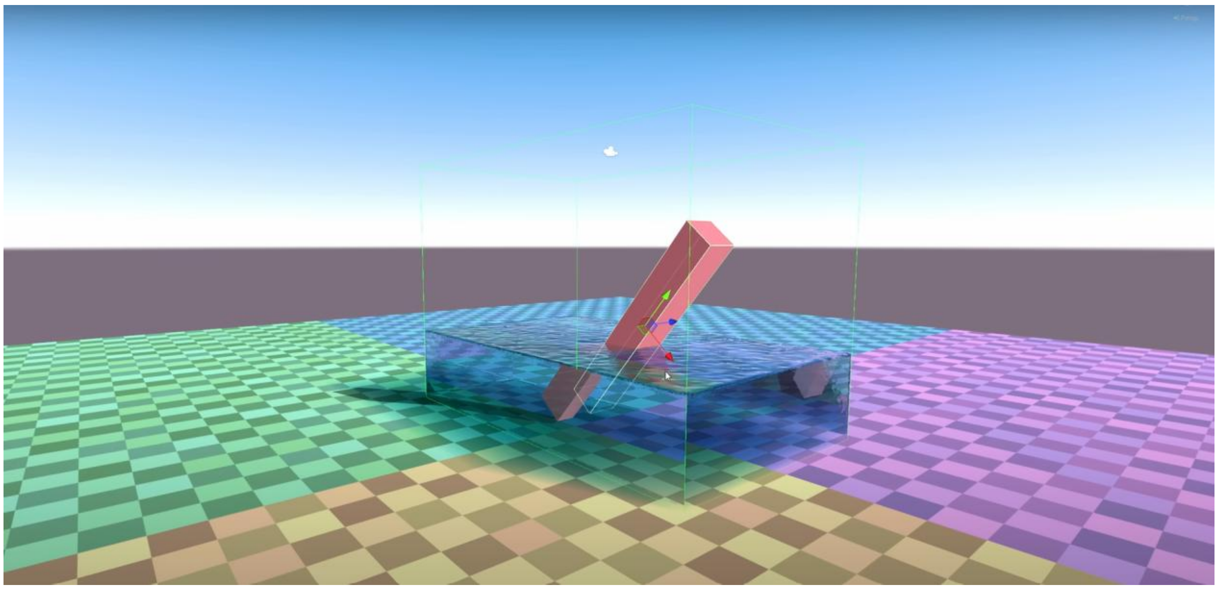


Figure 25 Screenshot of the ray marching rendering implemented of Lague's Unity project. (Source: Lague, 2024, Coding Adventure: Rendering Fluid [YouTube video])

2.2.3 Optical Properties of Fluids in Physically Based Rendering

In physically based rendering (PBR), fluids are typically modeled as dielectric materials. Transparent substances such as water, oil, or certain gases share optical properties with other non-metallic materials like glass or plastic.

As Christopher Wallis highlights in his article *Making of Moana* (*the Shadertoy)* (2020), three main optical properties are essential to achieving a believable visual representation of a fluid: reflection, refraction, and absorption. These are the physical foundations that govern the appearance of transparent fluids in real-time and offline rendering.

1. Reflection, which defines how much light is reflected at the fluid's surface.
2. Refraction, which describes how light changes direction as it passes through the interface between two media, based on the refractive index.
3. Absorption, which models the gradual loss of light intensity as it travels through the fluid volume.

2.2.3.1 Reflection

The law of reflection states, that the angle θ between the incoming light ray r_i and the surface normal $\vec{n}$ is equal to the angle θ_r between the reflected light ray r_r and the surface normal.

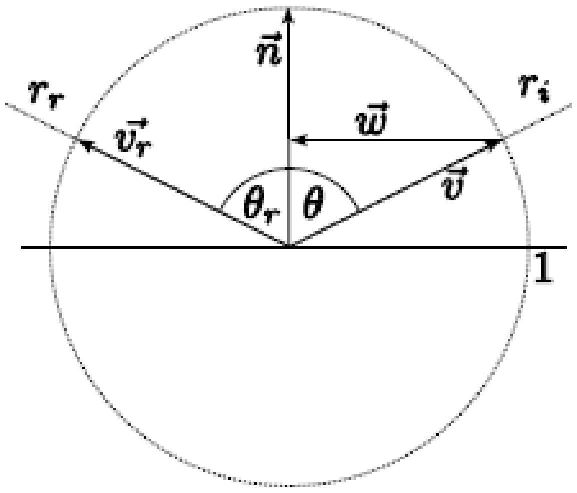


Figure 26 Diagram illustrating the law of reflection (Fleck, 2008).

This behaviour is mathematically simple and is supported natively in shader languages through functions.

In rasterization pipelines, reflections are typically approximated using screen-space reflection (SSR) techniques. However, these are limited to what is visible on-screen and can produce artifacts.

In contrast, ray tracing pipelines have full access to the scene's geometry, enabling more accurate and recursive reflections, even of objects not directly visible to the camera.

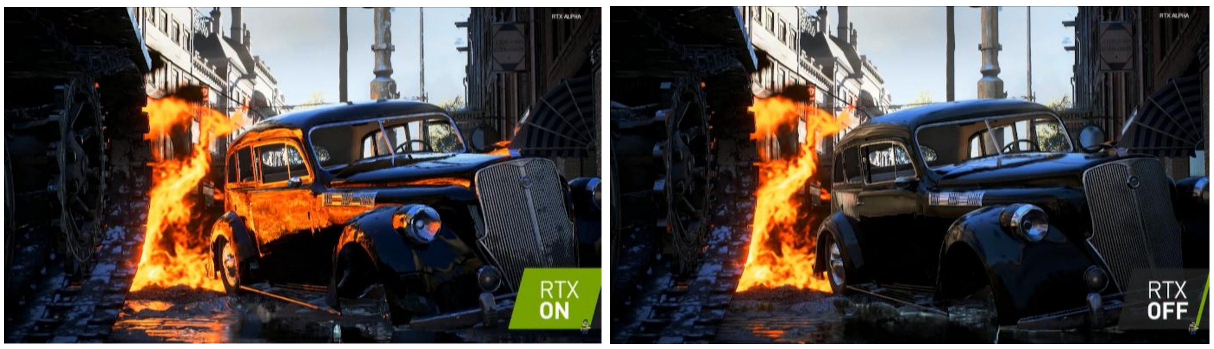


Figure 27 Comparison of rendering reflections with ray tracing (left) and without ray tracing (right) in Battlefield V (DICE, 2018)

2.2.3.2 Refraction

Snell's Law describes the bending of light as it passes from one medium to another with a different refractive index. It states that the angle θ between the incoming ray r_i and the surface normal $\vec{n}$ is related to the angle θ_t between the refracted light ray r_t and the inverse normal $\vec{n}_t$. This relationship is expressed as:

$$n_1 \cdot sin(\theta) = n_2 \cdot sin(\theta t)$$

where n1 and n2 are the refractive indices of the incident and transmitted media, respectively.

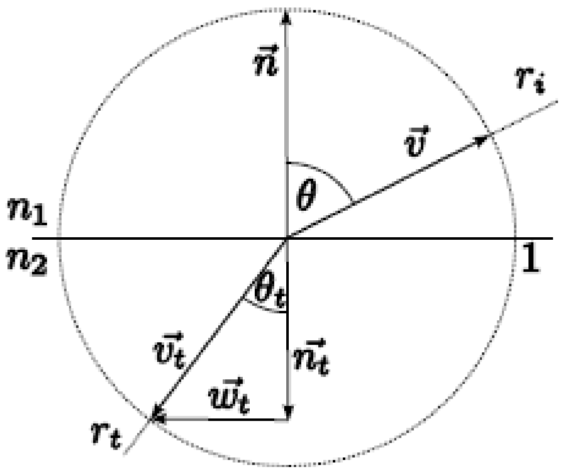


Figure 28 Diagram illustrating the law of refraction (Fleck, 2008).

Rasterization struggles with realistic refraction because it only sees screen-space geometry. Techniques like UV distortion can fake the effect, but they fail with off-screen or layered objects, causing artifacts. In contrast, ray tracing accurately simulates refraction by tracing rays through the full 3D scene, making it much better suited for rendering transparent fluids.

2.2.3.3 Fresnel Schlick's Approximation (Fresnel Effect)

The Fresnel effect describes how the amount of reflected light increases as the angle of incidence becomes more grazing. This is particularly visible on fluid surfaces such as water, where two key phenomena can be observed, as shown in Figure 29:

At direct angles (1), light enters the fluid more easily, making it appear transparent, and revealing underwater elements.

At grazing angles (2), more light is reflected, resulting in stronger reflections near the edges of the viewing direction.

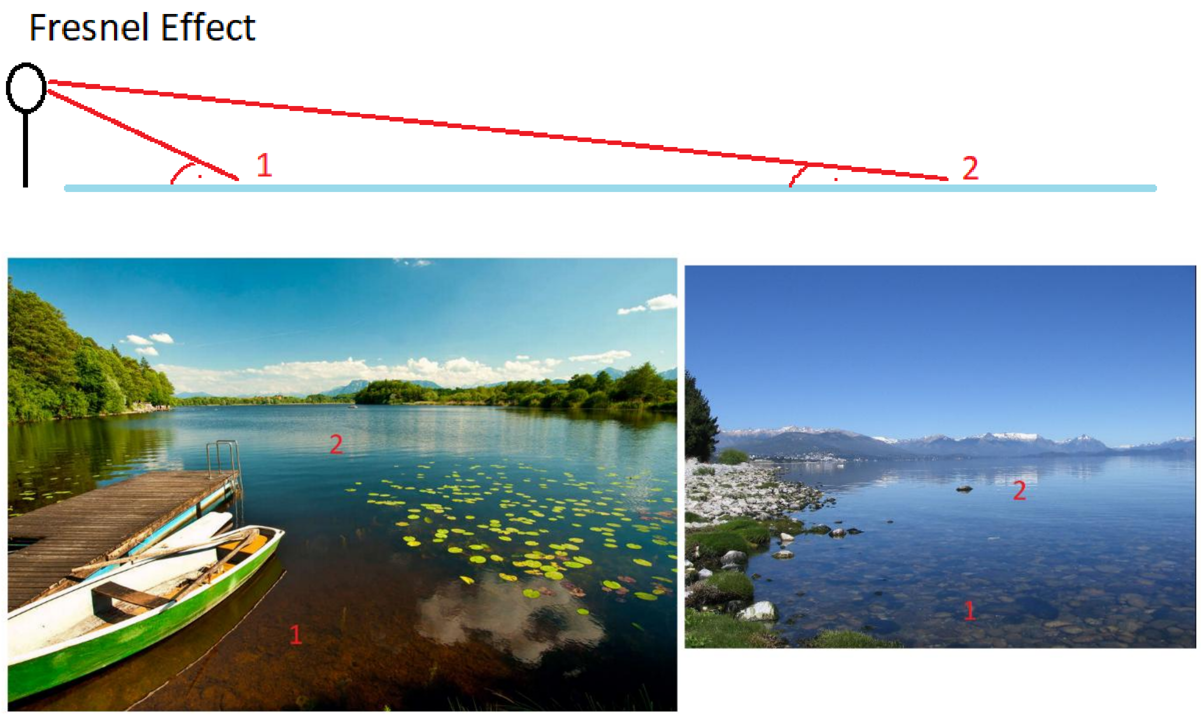


Figure 29 Illustration of the Fresnel effect on a water surface.

In real-time rendering engines, this behaviour is commonly modelled using Schlick's approximation, a simplified formula that estimates the reflectance coefficient without computing the full Fresnel equations:

$$R(\theta) = R_0 + (1 - R_0)(1 - \cos(\theta))^5$$

where:

$R(\theta)$ is the reflectance at angle θ (the angle between the incident ray and the normal),

R_0 is the reflectance at normal incidence, typically derived from the refractive indices of the two media,

θ is the angle of incidence.

As explained by Shirley et al. (2021), this approximation offers a strong balance between physical realism and computational cost, making it a standard technique in PBR workflows for rendering dielectric materials.

2.2.3.4 Absorption

Absorption models the gradual attenuation of light as it passes through the fluid, based on the Beer-Lambert law:

$$I = I_0 \cdot e^{-k \cdot d}$$

where:

I_0 is the incident light intensity,

k is the absorption coefficient (often RGB),

d is the path length within the fluid.

This law is fundamental for simulating volumetric coloration, such as the bluish tint of deep water or the amber hue of oil. The exponential decay governs how light is absorbed differently across wavelengths, depending on the material’s optical density.

In ray tracing, applying Beer’s law is straightforward, as the distance ddd traveled by each ray through the medium is explicitly known. This enables accurate simulation of volumetric light attenuation.

In contrast, rasterization lacks direct access to volumetric ray paths. As a result, it relies on approximations such as thickness maps or screen-space depth estimates, which are less precise and often fail to capture subtle absorption effects across varying depths.

2.2.4 Modern Fluid Rendering Sources

One of the most advanced real-time fluid simulation and rendering solutions to date is LiquiGen, a proprietary software released in closed alpha in February 2025. Designed for both the VFX and game development industries, LiquiGen features a real-time viewport that supports both rasterization and a software-based path tracing pipeline, depending on user preference.

In a short interview conducted for this thesis, Oliver Cruickshank, a rendering programmer at JangaFX, explained that LiquiGen initially used the marching cubes algorithm for surface reconstruction. However, the team later transitioned to dual contouring, a more advanced and topologically accurate alternative. For realistic refraction in rasterized rendering, LiquiGen uses a raymarching approach to compute the fluid entry and exit points along the view ray (Cruickshank, 2025).

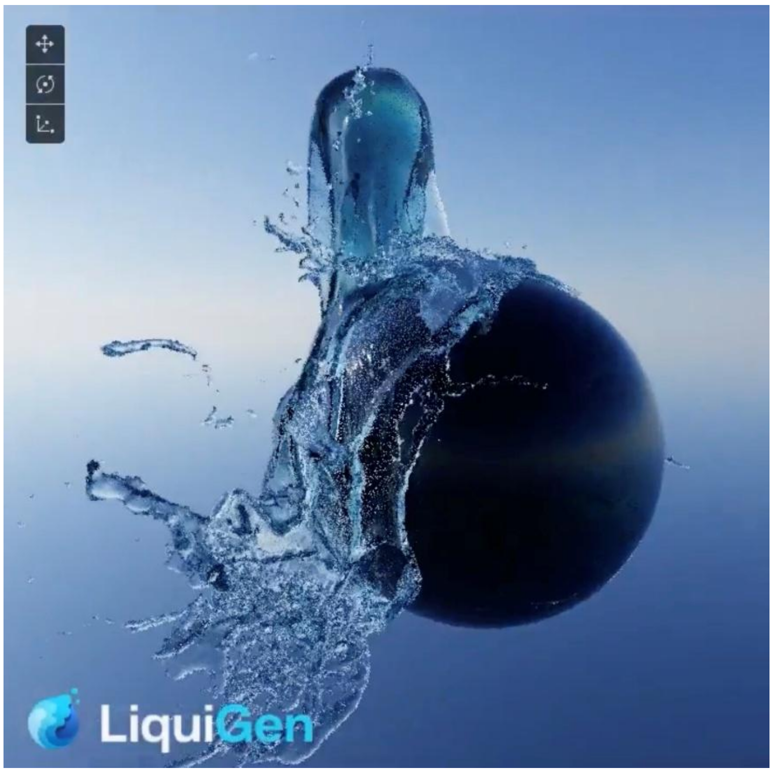


Figure 30 LiquiGen's real-time viewport (JangaFX, 2025).

A similar approach is implemented in Zibra Liquid, a plugin developed by ZibraAI for Unity and Unreal Engine 5. It mirrors LiquiGen's pipeline: dual contouring is used to reconstruct the fluid surface, followed by volumetric raymarching for refraction within the reconstructed mesh. According to Oleksandr Puchka in the *Zibra Liquid Glossary* blog post (2024), future support for DXR integration is under consideration.

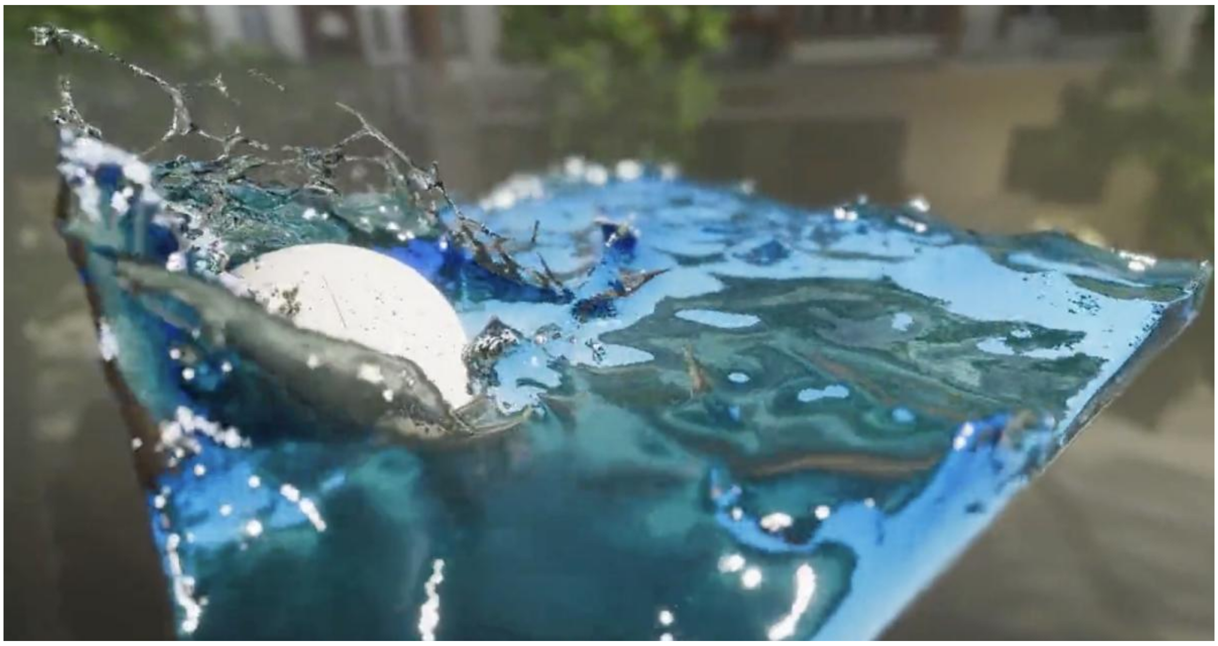


Figure 31 Zibra Liquid rendering in Unity (ZibraAI, 2024).

2.3 Anticipation of the Media Project

After reviewing and analysing existing practices in ray tracing and implicit fluid rendering based on particle-based simulations, it is now possible to anticipate the practical methodology that will guide the Media Project which composed of three milestones.

The first technical milestone will involve implementing a dielectric material model. This will validate recursive ray traversal and the simulation of physical light interactions, specifically reflection, refraction, and absorption, within transparent volumes.

The second phase consists of integrating a Smoothed Particle Hydrodynamics (SPH) simulation developed by Constantin Verine, a fellow student conducting a parallel bachelor thesis. This collaboration facilitates the use of a physically plausible particle-based fluid while allowing both projects to explore the challenges of coupling SPH with ray tracing.

Finally, once the simulation is integrated, the last step will be to render the fluid surface. Among the three techniques previously reviewed, the screen-space point sprite method has been excluded due to its reliance on depth-based approximations and image-space blurring, which are inherently incompatible with a ray tracing pipeline. Since the purpose of ray tracing is to overcome such limitations, reproducing a screen-space method would contradict the core objectives of this project.

The remaining two methods, marching cubes and volumetric ray marching, are both compatible with DXR. However, ray marching is preferred for its smoother, more visually realistic results, making it better suited for a study focused on optical fidelity in real-time raytraced fluid rendering.

3. Qualitative Analysis

After presenting the theoretical and technical foundations in the previous chapter, this section aims to examine how the challenges of representing dynamic and implicit fluids are addressed in practice. The objective is to understand how various actors, whether studios, researchers, or independent developers, approach these issues in different contexts, ranging from real-time production environments to experimental prototyping.

This qualitative analysis is based on two complementary sources of data. First, interviews conducted with professionals in the field of real-time rendering and simulation provide insight into current technological constraints, industrial practices, and expectations regarding the future of raytraced fluid rendering.

Second, the study examines concrete case examples that illustrate how different strategies are implemented in real projects, offering a critical perspective on the methods and compromises involved.

3.1 Interviews

To complement the technical references presented in the previous chapter, this section draws on insights gathered from interviews with professionals working directly in the field of real-time rendering. While academic sources provide a solid theoretical foundation, these interviews aim to reveal how fluid rendering challenges are actually approached in production contexts where constraints and priorities differ from purely research-oriented environments.

3.1.1 Profiles and selection of respondents

The first interview was conducted with Oliver Cruickshank, a rendering programmer at JangaFX, where he worked on both the real-time and path-traced pipelines of LiquiGen, a recent tool for fluid simulation for VFX and video games industries. His direct experience with fluid rendering and ray tracing made his perspective particularly valuable.

The second respondent, Thomas Poulet, is a graphics engineer at Something Mighty with a background in AAA development and R&D. He previously worked at Huawei Research and Ubisoft, contributing to high-fidelity rendering and engine systems. His broader expertise helps contextualize implicit surface rendering within larger production workflows.

By comparing these two profiles, one specialized, the other more generalist, the interviews provide a grounded view of how industry professionals approach fluid rendering challenges. Their insights offer a practical counterpoint to the theoretical discussions in this thesis, particularly regarding the adoption of DXR in real-time environments.

3.1.2 Overview

The interviews were structured around open-ended questions covering technical and conceptual aspects of fluid rendering. After brief introductions to each participant's professional background, the discussion focused on mesh reconstruction from particle data, volumetric rendering, optical properties of fluids, and the use of modern APIs like DXR. Each interview concluded with an invitation to share additional reflections or advice.

3.1.3 Synthesis of Responses

The interviews provided valuable and complementary insights into the technical challenges of implicit fluid rendering.

Implicit Volume Representation

Cruickshank explained that LiquiGen supports both Marching Cubes and Dual Contouring for surface reconstruction, with the method being selectable by the user. This flexibility facilitates the export of explicit meshes to external applications such as game engines or offline renderers.

In contrast, Poulet has worked primarily with volumetric ray marching, especially for stylized effects like clouds and 2D fluids. His use cases, typical of the video game industry, prioritize visual impact over physical accuracy and are usually implemented in screen space.

Optical Properties of Fluids

Both experts confirmed the importance of reflection, refraction, and absorption for achieving visually convincing fluid rendering, in line with the literature. Cruickshank further emphasized that subsurface scattering becomes critical for dense fluids (e.g., lava or honey), while caustics play a key role in the appearance of lighter, transparent fluids like water.

Refraction and Rendering Techniques

Accurate refraction remains a major challenge in rasterization-based pipelines. Cruickshank detailed LiquiGen's approach:

after rendering all opaque geometry, the front and back faces of the refractive volume are rendered. A ray marching pass is then performed between these two layers to estimate the refracted ray's exit point. The result is used to sample the opaque scene in UV space, while the traveled distance informs absorption via Beer's Law. However, this method has limitations, including an inability to refract through other refractive objects and visual artifacts due to the approximations involved. (Cruickshank, 2025)

Perspectives on DXR

Both interviewees expressed measured optimism toward DirectX Raytracing (DXR). They acknowledged its potential, but viewed it as still too resource-intensive for broad adoption in real-time applications.

Cruickshank noted that LiquiGen instead uses a custom path tracer based on compute shaders and a proprietary BVH. This design offers full control over the rendering process and is better suited to their needs, which involve rendering a single high-quality object rather than managing large dynamic scenes. He did not exclude future adoption of DXR as the API matures.

Poulet, by contrast, remains firmly committed to traditional rasterization, and is skeptical about current AI-driven technologies such as DLSS and frame generation. He believes real-time ray tracing will not match offline quality without significant compromises, and expects hybrid rendering pipelines, combining rasterization with selective raytraced effects to remain standard for the foreseeable future.

Ray Marching vs. Mesh Reconstruction

When asked to compare volumetric ray marching with explicit surface reconstruction in ray-traced pipelines, both interviewees favored the latter. Ray-triangle intersection is natively supported and hardware-accelerated in DXR, making it more efficient and straightforward than implementing custom volumetric intersection shaders.

That said, both acknowledged the performance cost of rebuilding the BVH every frame for dynamic scenes. Ray marching, while visually smoother and often easier to implement, does not benefit from DXR's acceleration structures and remains computationally expensive, similar to its behavior in rasterized pipelines.

3.1.4 Insights and Comparison with Industry Practice

These interviews reflect broader trends in the field. Particle-based simulations are still largely restricted to VFX, engineering, and academic contexts. In interactive media, developers prefer approximate, screen-space methods due to their speed and visual flexibility.

Ray marching is appreciated for its simplicity and visual quality but remains too expensive for widespread real-time use. Mesh reconstruction, though more complex, is preferred when integration with external tools or scene-wide lighting is required.

Regarding DXR, it remains a young and relatively underutilized API, mostly explored in research or demo scenarios, particularly by companies like NVIDIA. In the context of implicit fluids, DXR could significantly improve lighting for reconstructed meshes, assuming the BLAS can be updated every frame. Otherwise, ray marching remains a viable solution for leveraging DXR's ray generation and recursive reflection/refraction features, though it offers no performance gains over traditional rasterization in its current form.

3.2 Case Study: Unity Fluid Simulation Rendering by Sebastian Lague

To complement the interviews, this section presents a case study illustrating how the challenges of rendering dynamic, implicit fluid surfaces can be addressed in practice. The example is drawn from a personal project by Sebastian Lague, published on his YouTube channel on December 6, 2024, as part of his Coding Adventures series. In this project, Lague explores various real-time rendering techniques applied to a Smoothed Particle Hydrodynamics (SPH) simulation initially developed in Unity in October 2023.

The objective of the project is to compare three distinct rendering methods: (1) surface reconstruction using the Marching Cubes algorithm, (2) volumetric ray marching based on density fields, and (3) screen-space rendering using point sprites. These techniques directly echo those reviewed in Chapter 2 as the main strategies for visualizing implicit fluid surfaces in interactive applications.

Through Lague's implementation, this case study provides concrete insights into the technical trade-offs of each approach, identifying where traditional rasterization reaches its limits and where GPU-based ray tracing could either improve realism or introduce additional computational challenges, particularly in the context of the author's own Media Project.

3.2.1 Marching Cubes

The first method implemented is the Marching Cubes algorithm, used to extract a polygonal surface from the scalar density field of the fluid. In Lague's initial prototype, mesh data (vertices and triangles) is generated on the GPU using a compute shader, transferred to the CPU to construct a Unity mesh, then sent back to the GPU for rendering. This process, although functional, becomes increasingly inefficient at higher resolutions due to the cost of data transfer between CPU and GPU.

To overcome this limitation, Lague transitions to a fully GPU-based pipeline, using an indirect draw call that eliminates the need for CPU-side mesh construction.

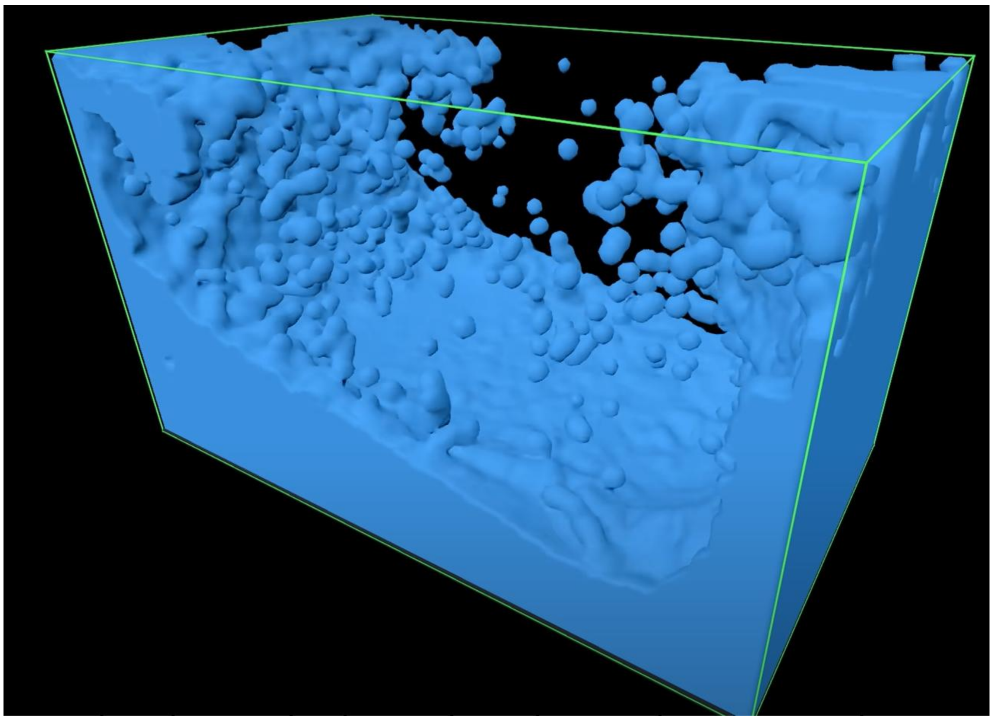


Figure 32 Fluid surface generated using GPU-side indirect draw call, avoiding CPU-GPU data transfer. Screenshot from Coding Adventure: Rendering Fluid (Lague, 2024).

However, Lague stops his Marching Cubes implementation at this point. He notes that the resulting mesh is inherently opaque, making it poorly suited for rendering transparent fluids. He considers using ray tracing on the mesh to simulate reflection and refraction but acknowledges the technical challenges involved, particularly the need to build and update a BVH every frame, a demanding task for GPU-side processing.

3.2.2 Ray marching

In response to the limitations of explicit surface meshing, Lague adopts a volumetric ray marching technique that directly samples the simulation's density field. This approach bypasses mesh generation entirely and enables the simulation of optical phenomena such as refraction, absorption, and internal reflection by tracing rays through the volume.

To evaluate the realism of this method, Lague constructs a full 3D scene, including a procedural ground plane, an immersive skybox, and a solid object submerged in the fluid. Custom ray-scene intersection functions are implemented manually, and textures are procedurally defined within the shaders, eliminating the need for UV coordinates.

Despite not using a formal ray tracing API, the implementation mirrors many ray tracing concepts, including recursive reflections, dielectric surface modeling, and basic light transport. It effectively emulates a ray tracing pipeline built atop rasterization techniques.

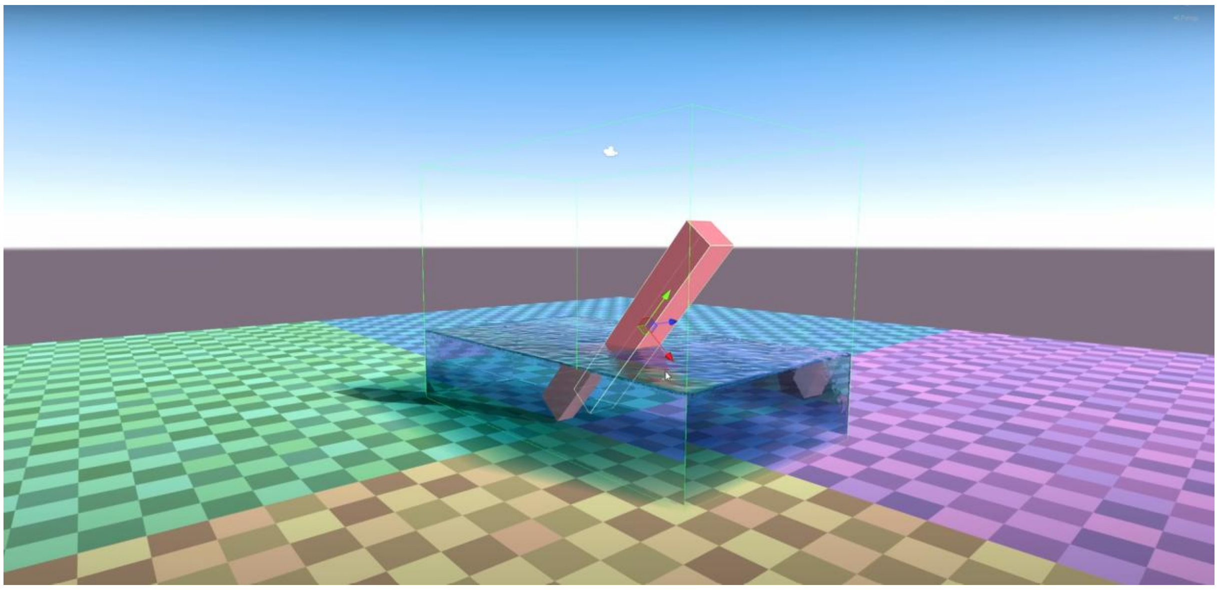


Figure 33 ray marched fluid rendering with visible reflection and refraction of the surrounding environment. Screenshot from Coding Adventure: Rendering Fluid (Lague, 2024).

To validate the physical plausibility of the simulation, Lague compares his rendering output to a photograph of a straw immersed in a bowl of water, highlighting the realistic bending of light through the fluid.

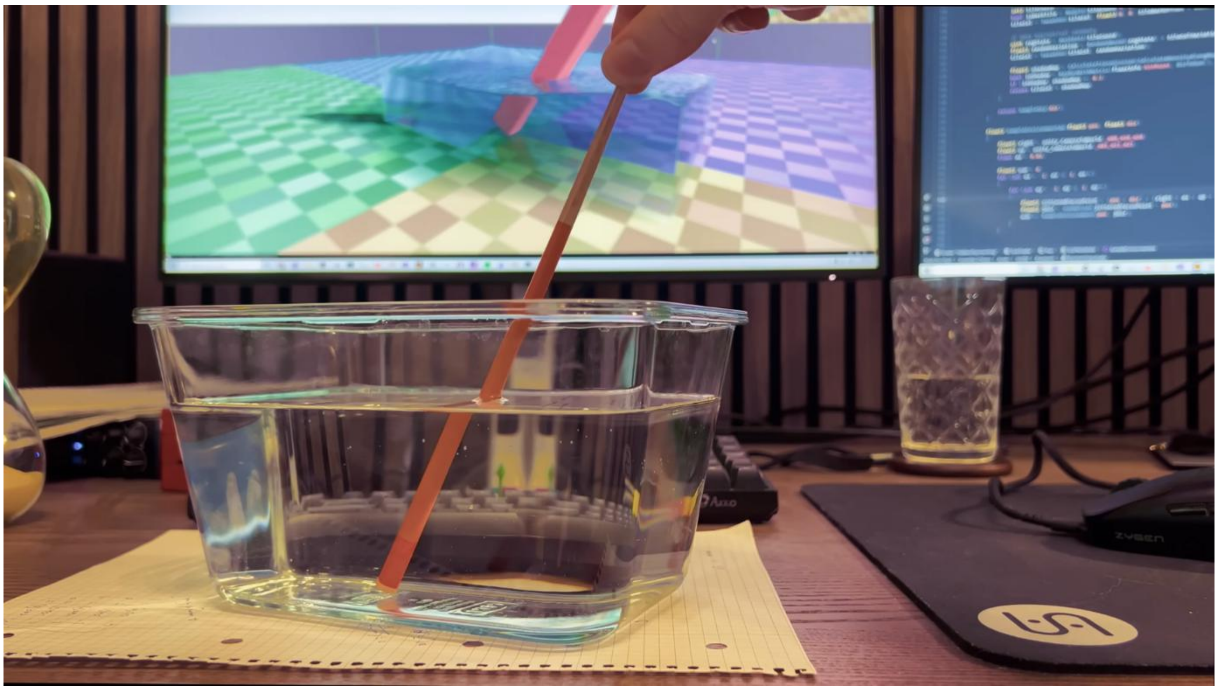


Figure 34 Comparison between simulated refraction and a real-world reference image of a straw in water. Screenshot from Coding Adventure: Rendering Fluid (Lague, 2024).

3.2.3 Point Sprite Screen-Space Technique

Finally, Lague explores the Point Sprite Screen-Space technique (van der Laan et al., 2009), a widely adopted method for real-time fluid rendering. He also integrates techniques from the work of Ihmsen et al. (2012) to simulate sprays, foam, and bubbles, enriching the visual detail of the fluid.

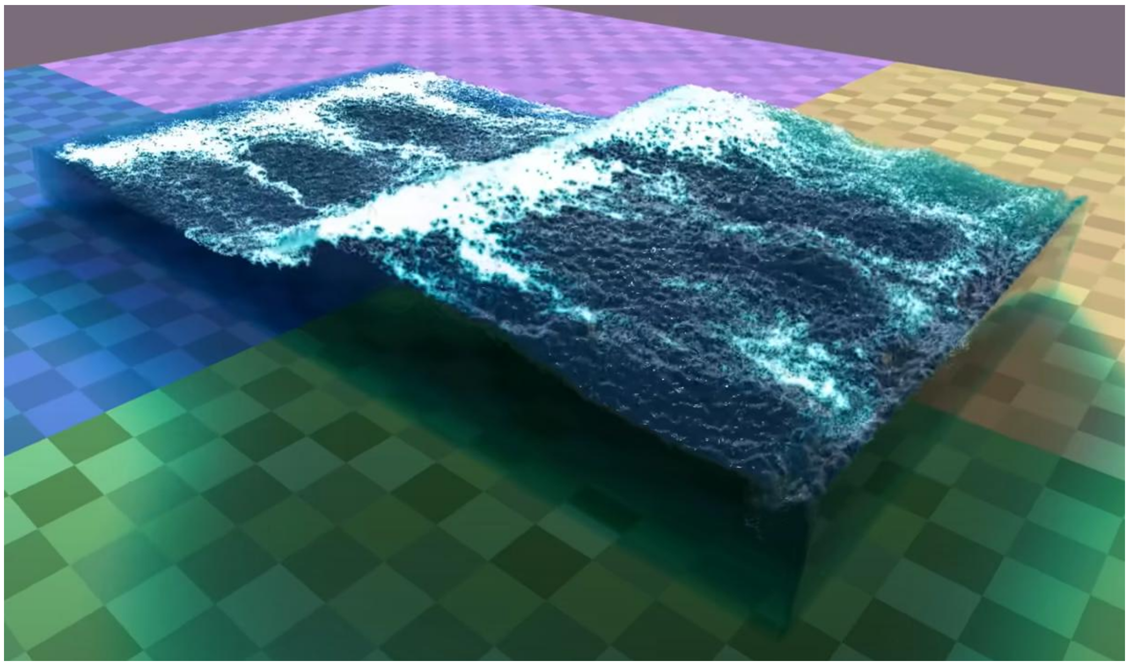


Figure X – Screenshot of the fluid surface enhanced with foam, bubbles, and spray effects based on the method of Ihmsen et al. (2012).

Lague ultimately concludes that the screen-space approach offers the best compromise for real-time applications. While it is less physically accurate than ray marching, its performance and visual quality make it the most viable option, especially with the integration of foam and spray effects derived from academic research.

He also notes the technique's limitations, such as its "blobby" look[1], the imprecise absorption, and poor visual fidelity when the camera enters the fluid volume. The blur-based surface representation quickly reveals its limits in close-up views.

[1] The blobby look refers to a visual effect where the fluid surface appears to be composed of blurred or merged spheres, typical of implicit renderings such as point sprites or si-gated distance fields. This phenomenon is common in computer graphics and is even intentionally exploited by NVIDIA in its rendering of fluids via ray tracing, as described in Ray Tracing Gems II (Pharr, 2021, chap. 35).

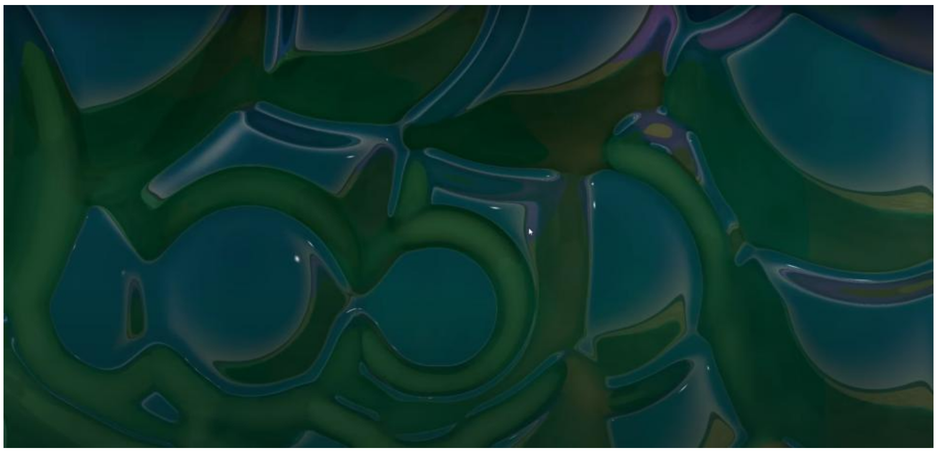


Figure 35 Artifacts observed when the camera is placed inside the fluid volume using screen-space rendering. Screenshot from Coding Adventure: Rendering Fluid (Lague, 2024)..

3.2.4 Conclusion

Lague’s fluid rendering project offers valuable insights into the trade-offs between surface realism, performance, and implementation complexity. His use of Marching Cubes revealed the limitations of mesh-based approaches for rendering transparent fluids, as the generated geometry remains opaque and visually limited without further optical processing. Given the capabilities of DXR to handle reflection, refraction, and acceleration structures, it represents a promising avenue for extending mesh-based methods, and it would be particularly interesting to test its integration with surface reconstruction.

Lague’s volumetric ray marching technique proved more physically accurate, allowing for detailed optical effects by sampling the fluid's density field directly. While his version was built manually on top of rasterization, this work integrates the same principles within a native DXR pipeline, offering more structured support for recursive rays, scene interaction, and dielectric materials.

Lastly, although the screen-space point sprite technique achieved the best real-time performance in Lague’s comparison, it remains fundamentally tied to rasterization. As such, it falls outside the scope of this ray tracing–focused implementation.

3.3 Methodological Reassessment of the Media Project

The qualitative analysis refined the technical direction of the Media Project. Interviews with industry professionals emphasized the potential benefits of testing a Marching Cubes approach to leverage DXR's hardware acceleration for ray-triangle intersections. While promising, this method may face limitations in mesh resolution and performance due to dynamic BLAS updates.

Additionally, the case study of Lague's fluid rendering highlighted the importance of fluid self-shadowing, especially soft blue shadows, which are difficult to achieve in rasterization. This insight led to the inclusion of volumetric shadow rendering in the ray marching pipeline.

As a result, the Media Project has been revised to:

- Implement both ray marching and marching cubes for surface reconstruction;
- Include shadow rendering techniques for added realism.

The rest of the methodology, including milestone-based validation and algorithm comparison, remains unchanged and is further supported by industry feedback.

4. Media Project

This chapter presents the practical component of the research, along with the testing protocol designed to address the central research question.

The first section outlines the testing environment, the data to be collected, the tools used, and the expected outcomes. The second section details the development process of the project, structured around key validation milestones.

4.1 Test Protocol

To evaluate the feasibility of rendering implicit, dynamic fluids within a DXR-based ray tracing pipeline, this section defines the testing protocol used throughout the study. Two rendering methods are assessed: ray marching and Marching Cubes, both implemented in real time using the Falcor 8.0 framework.

The primary metric is frame time, measured in milliseconds per frame. This value reflects the total computational cost of rendering a single frame. In interactive environments, real-time rendering is generally defined by a threshold of 30 ms or less, corresponding to approximately 33 frames per second. While video games often aim for 16 ms (60 FPS), this study considers the 30 ms threshold to be a realistic minimum for real-time applications such as simulation previews or interactive visualizations.

In addition to global frame time, the computation time of the ray tracing pass is measured independently. This allows a more precise evaluation of the cost introduced by fluid-specific algorithms within the ray tracing stage, and helps isolate their scalability and integration impact.

Visual quality is evaluated qualitatively. Although it cannot be measured directly, it is assessed by visual inspection based on a set of predefined optical criteria: accuracy of refraction, reflection sharpness, light absorption behavior, and surface continuity. These criteria are informed by both physical models and professional feedback obtained during the qualitative research phase.

To explore the trade-offs between performance and realism, two key parameters are systematically varied for both rendering methods:

- Number of ray bounces
- Resolution of the 3D density map

This protocol is designed to identify when a DXR-based pipeline becomes computationally prohibitive for real-time use, and conversely, under which conditions it provides tangible improvements in visual fidelity.

4.1.1 Test environment

The tests will be conducted using a 3D scene specifically designed to highlight the optical effects of fluid rendering. At the center of the scene lies the simulated fluid volume, partially enclosing a white box to observe light refraction and absorption through the medium. The floor features a checkerboard texture, providing a clear visual reference to assess distortion effects caused by the fluid's refractive index. An HDR cubemap surrounds the scene to simulate realistic lighting and reflections.

This configuration offers a consistent and controlled environment for comparing the visual and performance implications of different rendering techniques.

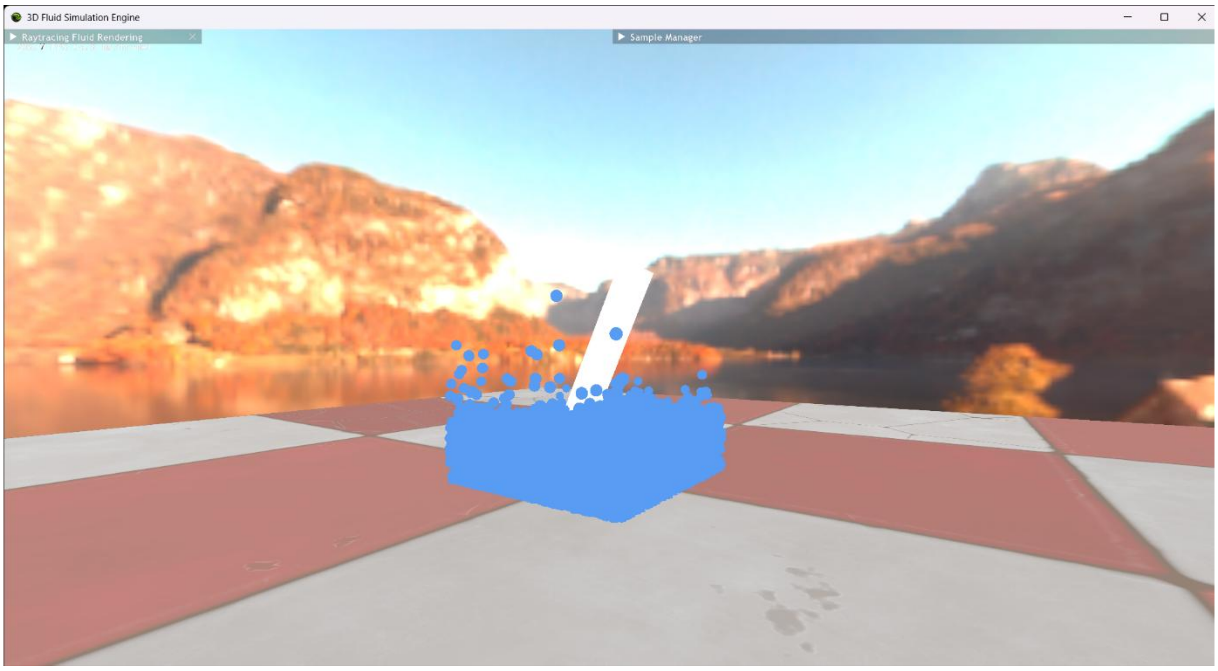


Figure X - Screenshot of the 3D scene made with the Falcor framework 8.0 (NVIDIA, August 2024).

4.1.2 Test data

The following parameters are used as the basis for all tests, unless explicitly varied as part of the performance study:

- SPH simulation with 20’000 particles
- Density map voxels count: 64^3, 128^3, 256^3
- Ray bounce count: 3, 4, 5
- Camera setup: fixed field of view, fixed position, looking at the fluid volume

These values ensure comparability between tests while focusing the analysis on the computational impact and visual output of the fluid rendering techniques. Adjustments are introduced progressively to evaluate performance thresholds and visual gains.

4.1.3 Tools

As previously mentioned, the rendering application was developed using Falcor 8.0 (NVIDIA, August 2024). This framework was selected for its efficient abstraction of the DirectX 12 API (Microsoft, March 2014) and its DXR ray tracing extension (Microsoft, October 2018), which significantly simplifies the creation of ray tracing-based 3D scenes.

The SPH fluid simulation used is the one developed by Constantin Verine, as mentioned in the media project anticipation. This choice aimed to directly experiment with the technical challenges of manually integrating a custom simulation into a ray tracing-based rendering pipeline.

Development was carried out using Visual Studio 2022 for C++ coding and dependency management.

Visual Studio Code was used to write Slang shader code, with the Slang extension installed.

CMake was employed as the build system to generate a Visual Studio solution that links all required dependencies.

To measure frame time with a breakdown between CPU and GPU execution, the built-in profiling system included in the Falcor framework was used.

4.1.4 Hardware

All performance tests and rendering evaluations were conducted on a system equipped with an NVIDIA GeForce RTX 3050 Ti Laptop GPU, based on the Ampere architecture. This GPU supports hardware-accelerated ray tracing via dedicated RT cores and is compatible with DirectX 12 Ultimate, providing full support for Direct3D feature level 12_2.

The system includes 12 GB of total graphics memory, composed of 4 GB of dedicated VRAM and 8 GB of shared memory. Tests were performed on Windows 11 (version 24H2), using WDDM 3.2 driver model and driver version 32.0.15.7283.

4.2 Technical Implementation

This section presents the major technical components implemented in the Media Project to enable the rendering of an implicit SPH-based fluid simulation using a DXR ray tracing pipeline. The focus lies on validating each necessary feature for real-time rendering, from the integration of dielectric material, particle simulation to the reconstruction of a fluid surface using ray marching and Marching Cubes.

4.2.1 Dielectric Material Rendering

To enable realistic rendering of transparent media such as fluids, recursive ray tracing was first implemented and validated using dielectric materials. This step was essential to test Fresnel reflection, Snell's law for refraction, and Beer-Lambert absorption before applying them to more complex scenarios like volumetric rendering.

The dielectric material was applied to both spherical and box geometries, producing realistic light interactions. These results were compared visually to a *ShaderToy* implementation by demofox (2017), known for accurately modeling dielectric surfaces using Fresnel and Beer’s laws. The comparison confirmed the validity of the implementation.

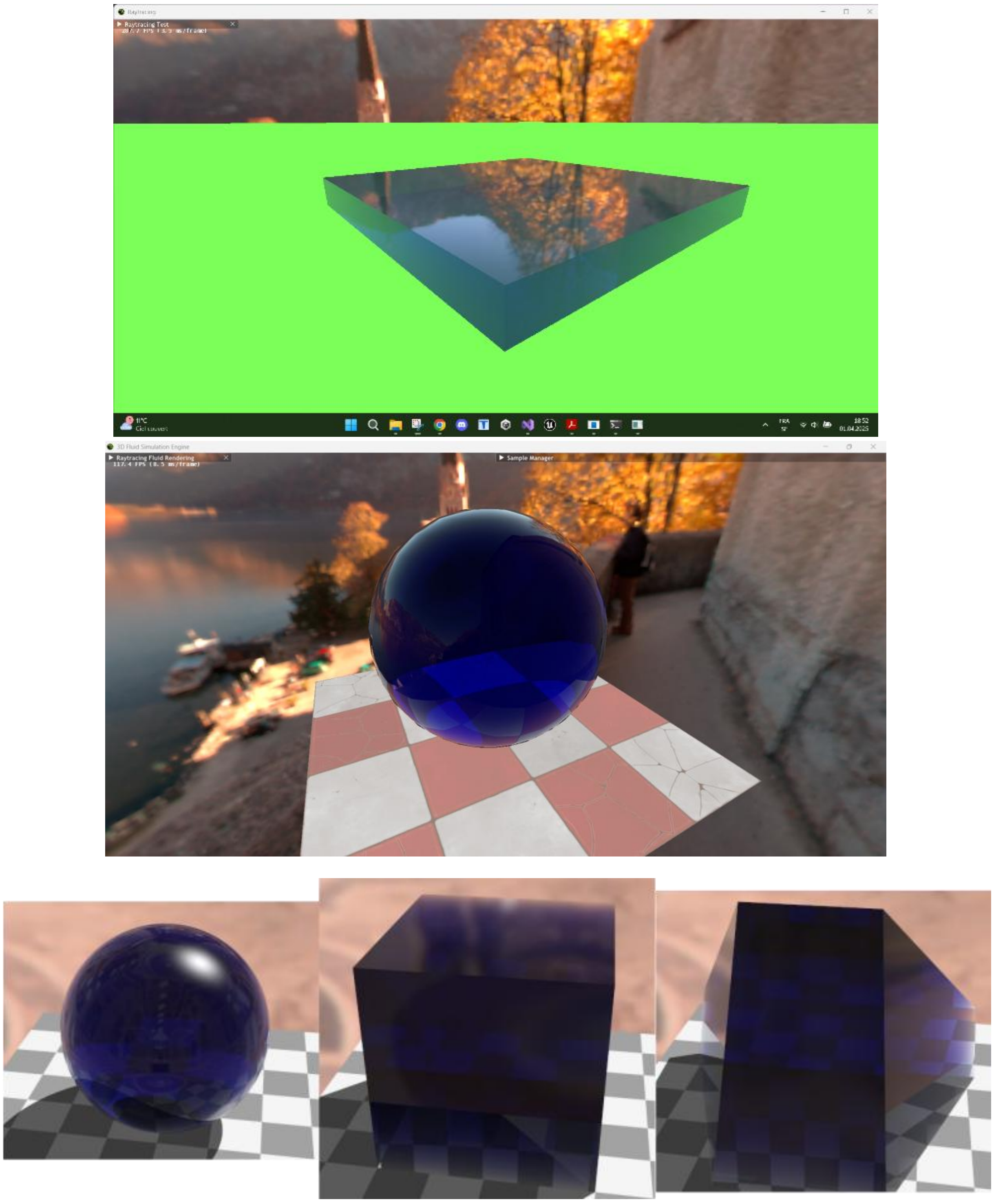


Figure 36 Dielectric material rendering in the Falcor application (first two screenshots), compared with a reference implementation by demofox (2017) on ShaderToy (third screenshot).

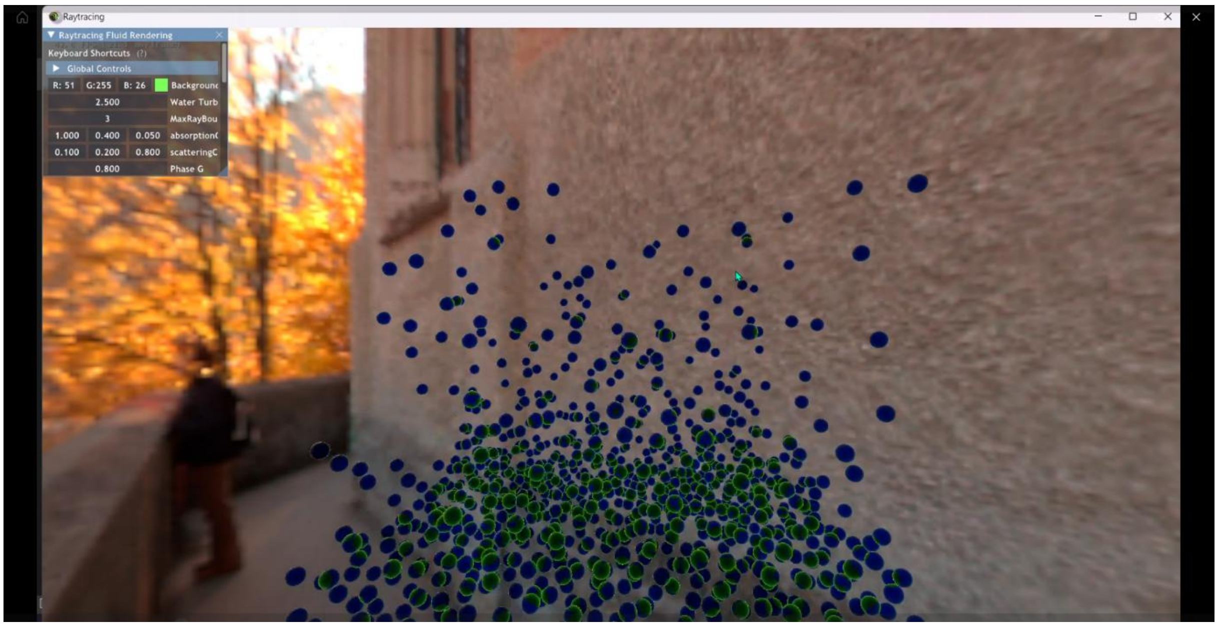
4.2.2 Integration of the SPH Simulation

In order to extract a surface from a fluid simulation and enable volumetric rendering through ray marching or Marching Cubes, the simulation data must be made accessible to the GPU. Initially developed as a CPU-based Smoothed Particle Hydrodynamics (SPH) system by Constantin Verine, the simulation was progressively ported to the GPU using compute shaders. This porting was driven by performance constraints: sampling the fluid’s density across a volumetric grid for the purpose of creating a density map proved prohibitively expensive on the CPU. Enabling this sampling efficiently required that both simulation and rendering logic reside on the GPU.

4.2.2.1 Debugging Render of the Simulation via TLAS Updates

Before porting the full simulation logic to the GPU, an intermediate validation step was introduced. The particle system was rendered as instanced spheres using the existing CPU simulation data. Each particle's transform matrix was uploaded and used to update the Top-Level Acceleration Structure (TLAS) every frame.

This approach allowed for visual debugging of the simulation’s behaviour and validation of the particle data pipeline, ensuring that the memory layout and update logic between the simulation engine and the rendering framework were functioning correctly in the GPU context.



```
R: 51   G:255   B: 26   🟩 Backgrounc
        2.500           Water Turb
          3             MaxRayBou
1.000   0.400   0.050   absorption(
0.100   0.200   0.800   scatteringC
        0.800           Phase G
```

Figure 37 Screenshot of the SPH particles rendered as instanced spheres.

4.2.2.2 Compute Shader Implementation and Neighbour Search Optimization

Once the data pipeline had been validated through TLAS-based visualization, the simulation itself was ported to a compute shader. This step enabled the particle system to be simulated entirely on the GPU and removed the bandwidth overhead of transferring data between CPU and GPU.

A key requirement for density computation in SPH is efficient neighbour lookup. To achieve this, the GPU implementation employed a spatial hash grid to group particles by cell index.

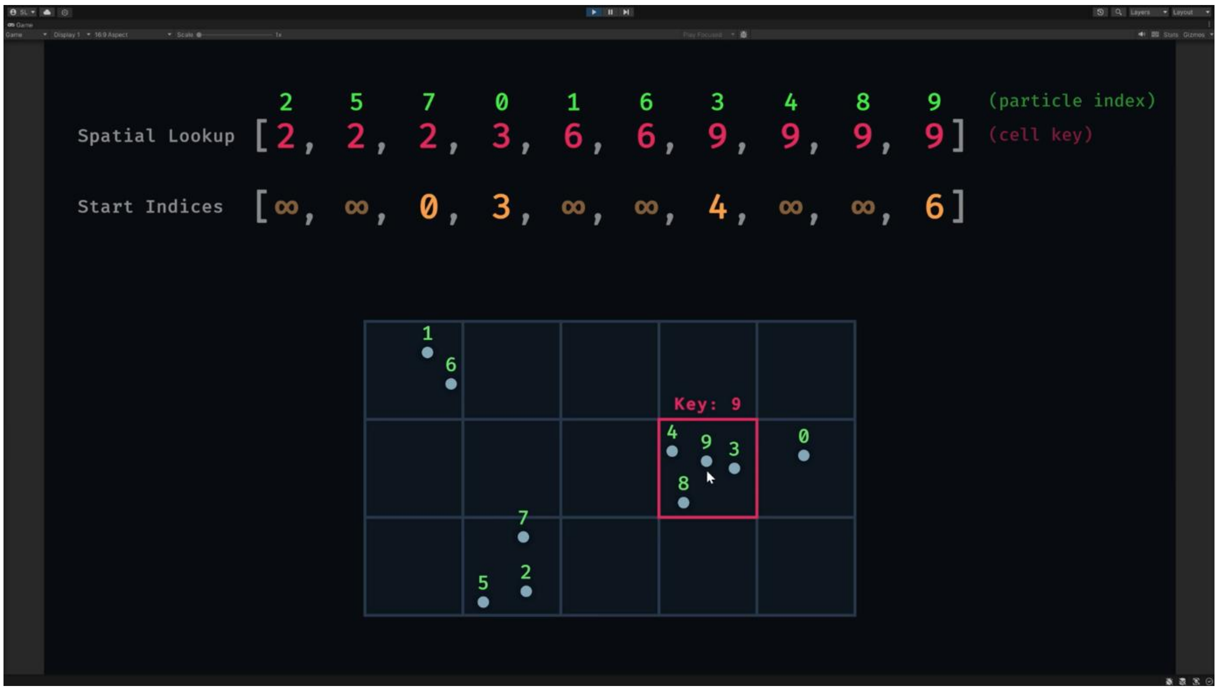

2 5 7 0 1 6 3 4 8 9
(particle index)
Spatial Lookup [2, 2, 2, 3, 6, 6, 9, 9, 9, 9]
(cell key)
Start Indices [∞, ∞, 0, 3, ∞, ∞, 4, ∞, ∞, 6]
Key: 9
1
6
4
9
3
8
0
7
5
2

Figure 38 Illustration of the principle of a spatial hash grid for neighbour lookup. Screenshot from Coding Adventure: Simulating Fluid (Lague, 2024)..

Then, a parallel Bitonic Sort was used to sort particles based on their spatial hash, allowing subsequent range queries to be executed in constant time. This design enabled each thread of the compute shader to calculate the local density by only inspecting particles in adjacent grid cells, rather than performing an O(n²) brute-force scan.

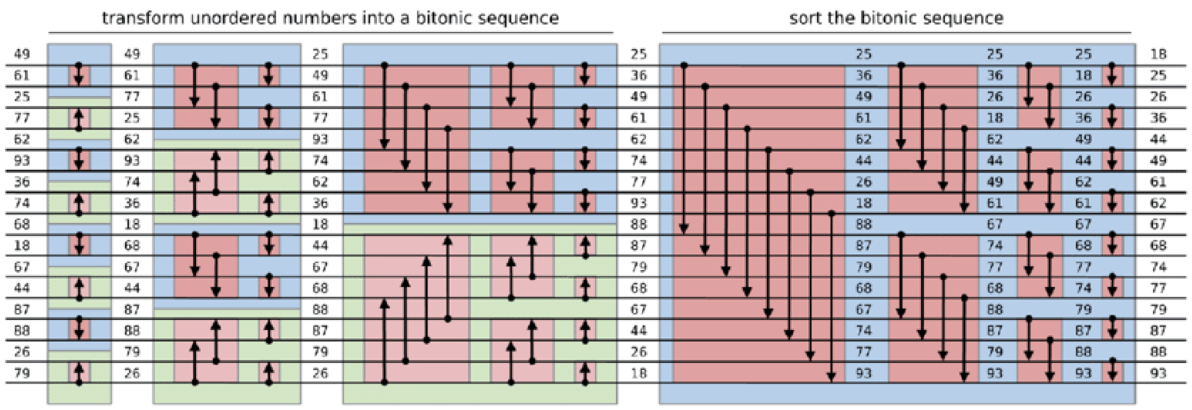

transform unordered numbers into a bitonic sequence
sort the bitonic sequence

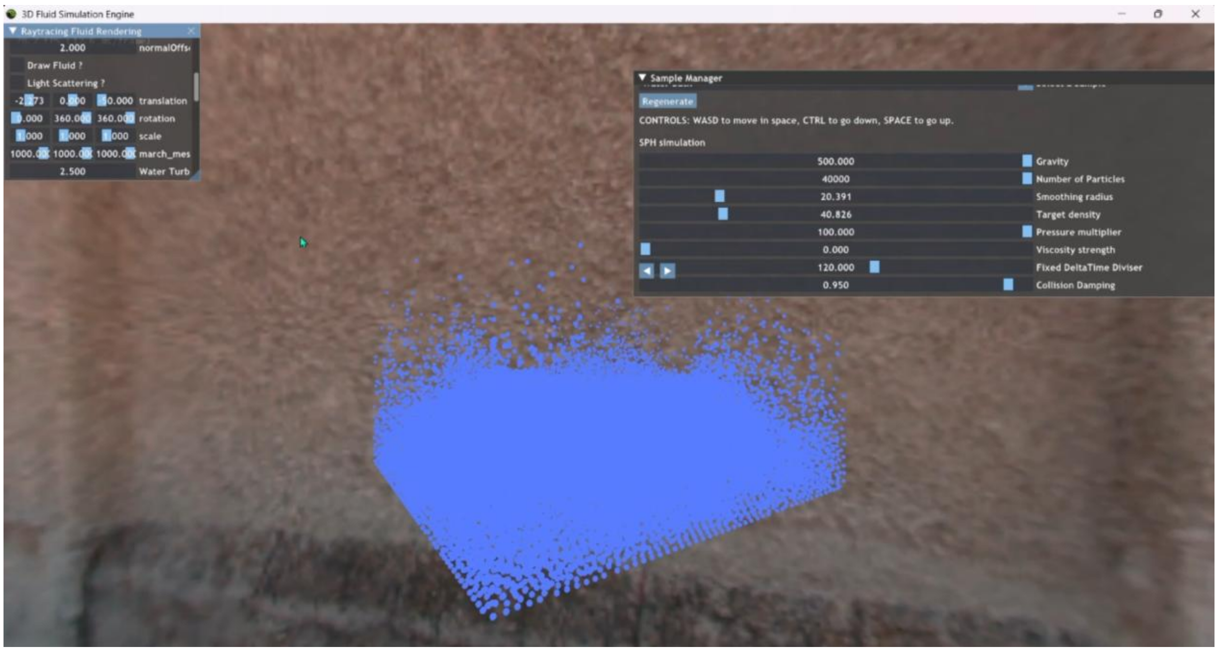
Figure 39 Visualization of the Bitonic Sort algorithm.

This architecture allowed the SPH simulation to scale up to 40,000 particles in real time, and, more importantly, made it feasible to compute a dense 3D texture (the density map) each frame as a foundation for rendering.

4.2.2.3 AABB Transformation for Dynamic Scenarios

To simulate more dynamic behaviours, such as fluid sloshing, directional motion, and strong splashing, the simulation volume was enclosed in a custom Axis-Aligned Bounding Box (AABB), which was registered as a dynamic primitive within the DXR acceleration structure.

Transformations, including rotations and scaling, were applied to this AABB via a matrix-based system. This allowed the entire simulation to react to external movement, making it possible to test the robustness of surface reconstruction algorithms under varying boundary conditions and fast-moving volumes.



```
2.000                          normalOffs
-2.273   0.000   50.000        translation
0.000    360.000 360.000       rotation
1.000    1.000   1.000         scale
1000.000 1000.000 1000.000     march_mes
2.500                          Water Turb
```

CONTROLS: WASD to move in space, CTRL to go down, SPACE to go up.

SPH simulation

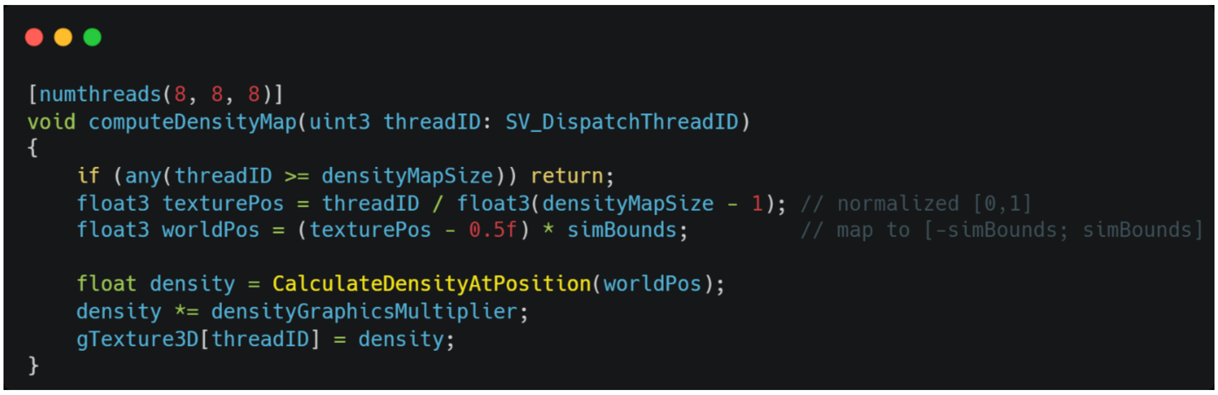
```
500.000    Gravity
40000      Number of Particles
20.391     Smoothing radius
40.826     Target density
100.000    Pressure multiplier
0.000      Viscosity strength
120.000    Fixed DeltaTime Diviser
0.950      Collision Damping
```

Figure 40 Screenshot of the fluid simulation with 40'000 particles after a significant Y-axis rotation, generating multiple splashes.

4.2.3 Creation of the Density Map

Once the SPH simulation was ported to the GPU, the next step involved generating a density map, a 3D texture representing the fluid's local density throughout the simulation volume. This texture acts as a bridge between the simulation and the rendering pipeline, enabling surface reconstruction via volumetric rendering with ray marching and via Marching Cubes.

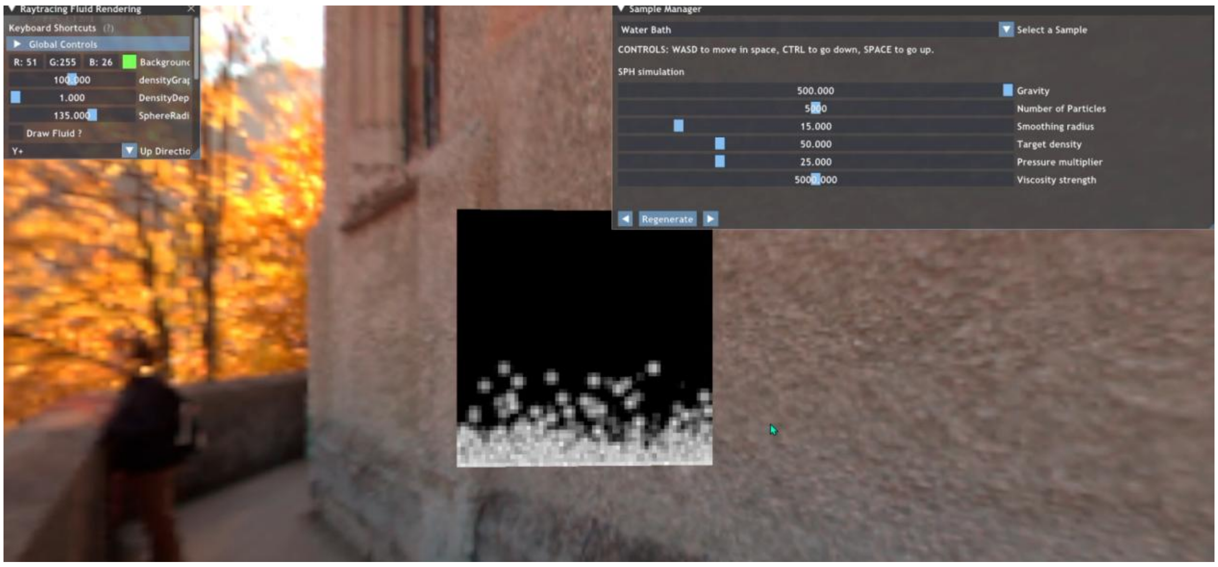
To build the density map, a compute shader is dispatched with a number of threads equal to the number of voxels in the 3D grid. Each thread samples the fluid density at its corresponding location in space, based on the simulation's current state. The spatial resolution of the map is defined by its texture size, which can be adjusted to trade-off between performance and surface fidelity.



```
[numthreads(8, 8, 8)]
void computeDensityMap(uint3 threadID: SV_DispatchThreadID)
{
    if (any(threadID >= densityMapSize)) return;
    float3 texturePos = threadID / float3(densityMapSize - 1); // normalized [0,1]
    float3 worldPos = (texturePos - 0.5f) * simBounds;         // map to [-simBounds; simBounds]

    float density = CalculateDensityAtPosition(worldPos);
    density *= densityGraphicsMultiplier;
    gTexture3D[threadID] = density;
}
```

Figure 41 The compute shader code used to compute the density map.

The result is a compact, GPU-resident volume texture that can be sampled efficiently during rendering passes.



R: 51 G:255 B: 26 🟩 Backgrounc
100.000 densityGra
1.000 DensityDep
135.000 SphereRadi
Y+ Up Directio

Water Bath Select a Sample
CONTROLS: WASD to move in space, CTRL to go down, SPACE to go up.
SPH simulation
500.000 Gravity
5000 Number of Particles
15.000 Smoothing radius
50.000 Target density
25.000 Pressure multiplier
5000.000 Viscosity strength

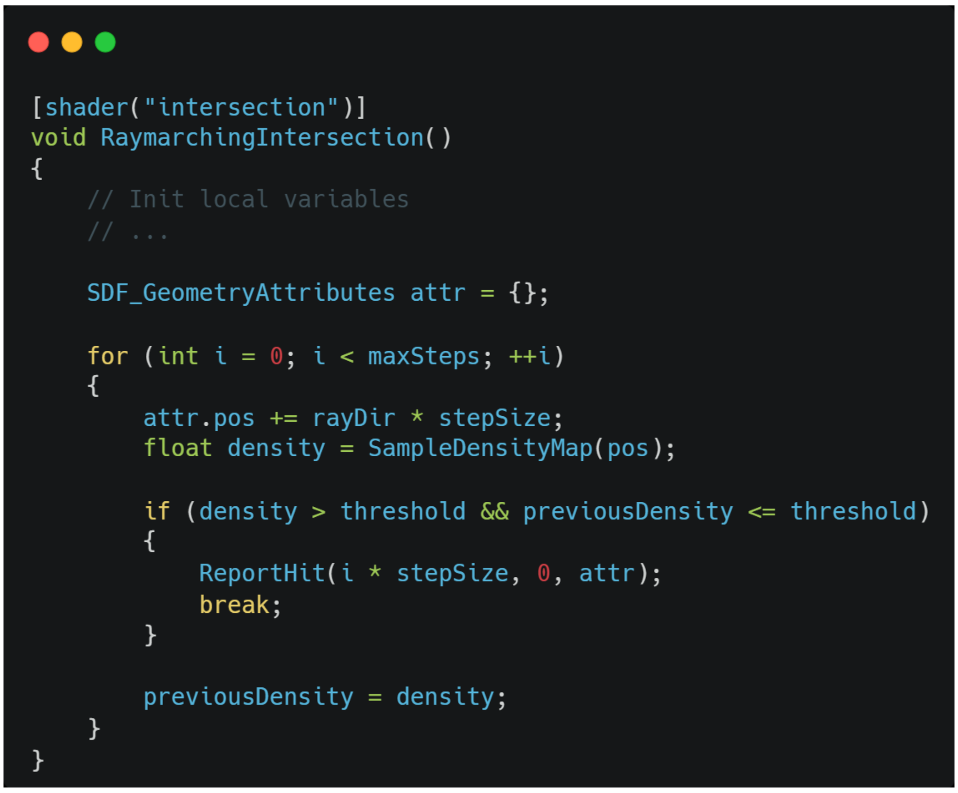
Figure 42 Screenshot of one slice of the density map

4.2.4 Ray marching

Once the 3D fluid density map has been generated via a compute shader, the rendering stage proceeds using a volumetric ray marching algorithm. This method is particularly well suited to rendering implicit surfaces, as it avoids the need for explicit mesh construction and integrates seamlessly with custom intersection shaders defined within the DXR ray tracing pipeline.

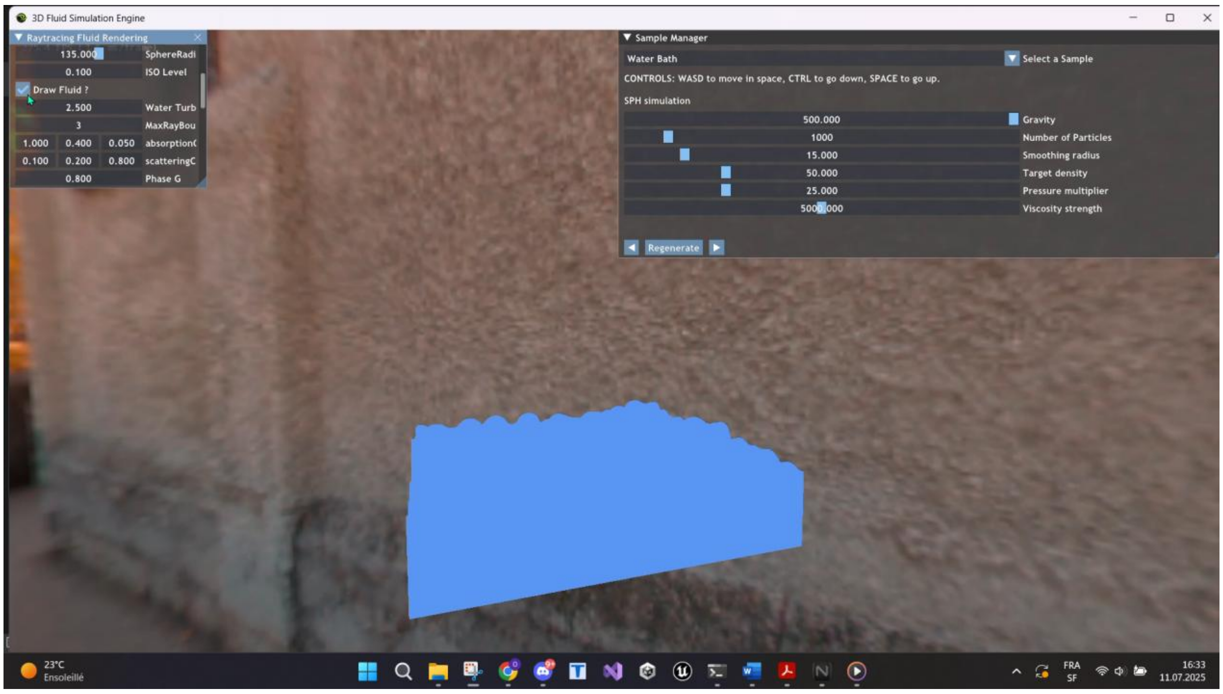
To detect the surface of the fluid, the custom intersection shader performs a forward march along the primary ray direction. An intersection is reported when a transition is detected from an empty region (low density) to a dense region, corresponding to the fluid boundary. Once this surface hit is found, the intersection shader provides relevant attributes (position, normal, local density, etc.) to the closest hit shader, which handles material shading.

The Figure 43 expresses the ray marching algorithm used to detect the fluid surface. Then the Figure 44 shows the fluid surface drawn with a uniform value.



```
[shader("intersection")]
void RaymarchingIntersection()
{
    // Init local variables
    // ...

    SDF_GeometryAttributes attr = {};

    for (int i = 0; i < maxSteps; ++i)
    {
        attr.pos += rayDir * stepSize;
        float density = SampleDensityMap(pos);

        if (density > threshold && previousDensity <= threshold)
        {
            ReportHit(i * stepSize, 0, attr);
            break;
        }

        previousDensity = density;
    }
}
```

Figure 43 Pseudocode illustrating the surface detection algorithm implemented in the custom intersection shader.


3D Fluid Simulation Engine

Raytracing Fluid Rendering

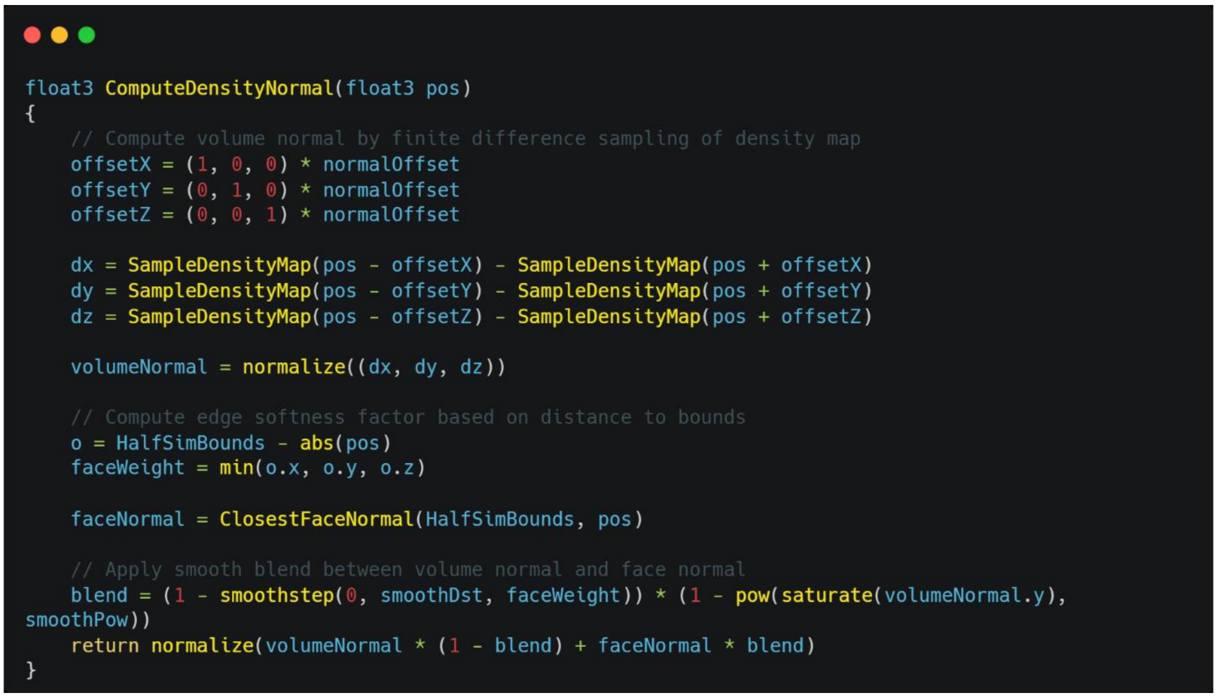
```
135.000              SphereRadi
0.100                ISO Level
[✓] Draw Fluid ?
2.500                Water Turb
3                    MaxRayBou
1.000  0.400  0.050  absorption(
0.100  0.200  0.800  scatteringC
0.800                Phase G
```

Sample Manager

```
Water Bath           Select a Sample
CONTROLS: WASD to move in space, CTRL to go down, SPACE to go up.
SPH simulation
500.000              Gravity
1000                 Number of Particles
15.000               Smoothing radius
50.000               Target density
25.000               Pressure multiplier
5000.000             Viscosity strength
```

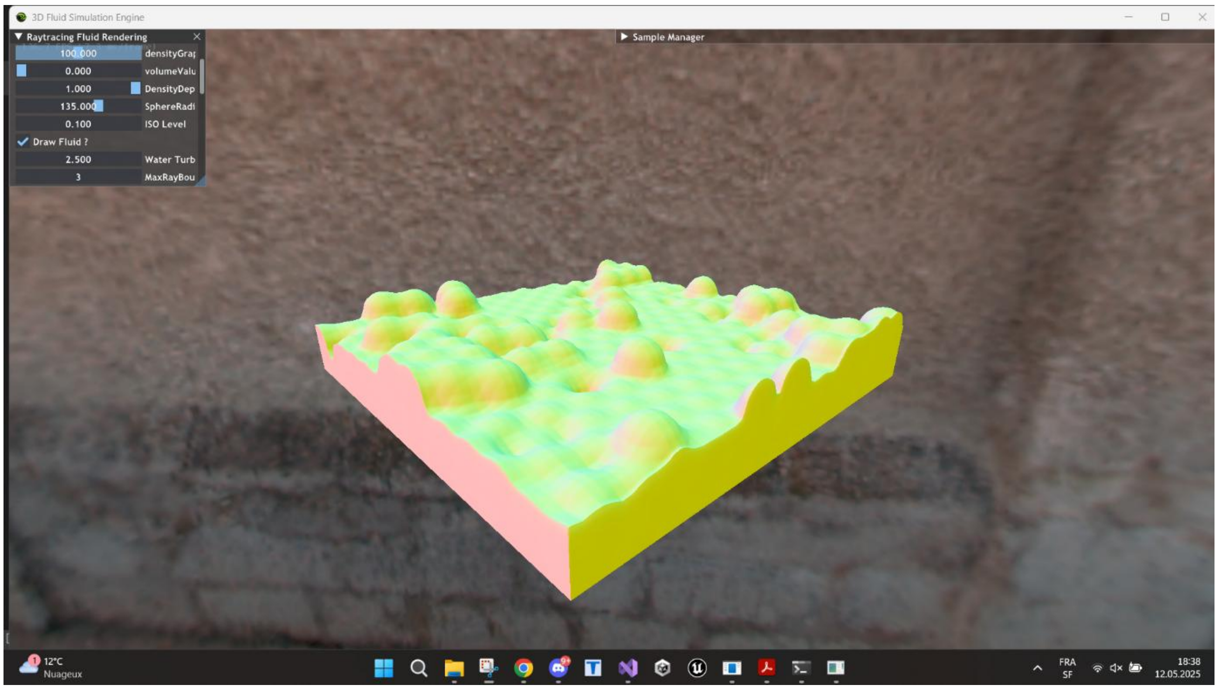
Figure 44 Visual result of the fluid surface reconstruction using ray marching in the intersection shader, with a uniform blue shading applied in the closesthit shader.

Surface normals are computed by sampling density values around the intersection point to estimate the gradient, producing smooth shading results. To reduce aliasing and artefacts near the fluid container's boundaries, a blending is applied between the computed volume normal and the container's box surface normal (see Figure 45 and 46).



```
float3 ComputeDensityNormal(float3 pos)
{
    // Compute volume normal by finite difference sampling of density map
    offsetX = (1, 0, 0) * normalOffset
    offsetY = (0, 1, 0) * normalOffset
    offsetZ = (0, 0, 1) * normalOffset

    dx = SampleDensityMap(pos - offsetX) - SampleDensityMap(pos + offsetX)
    dy = SampleDensityMap(pos - offsetY) - SampleDensityMap(pos + offsetY)
    dz = SampleDensityMap(pos - offsetZ) - SampleDensityMap(pos + offsetZ)

    volumeNormal = normalize((dx, dy, dz))

    // Compute edge softness factor based on distance to bounds
    o = HalfSimBounds - abs(pos)
    faceWeight = min(o.x, o.y, o.z)

    faceNormal = ClosestFaceNormal(HalfSimBounds, pos)

    // Apply smooth blend between volume normal and face normal
    blend = (1 - smoothstep(0, smoothDst, faceWeight)) * (1 - pow(saturate(volumeNormal.y),
smoothPow))
    return normalize(volumeNormal * (1 - blend) + faceNormal * blend)
}
```

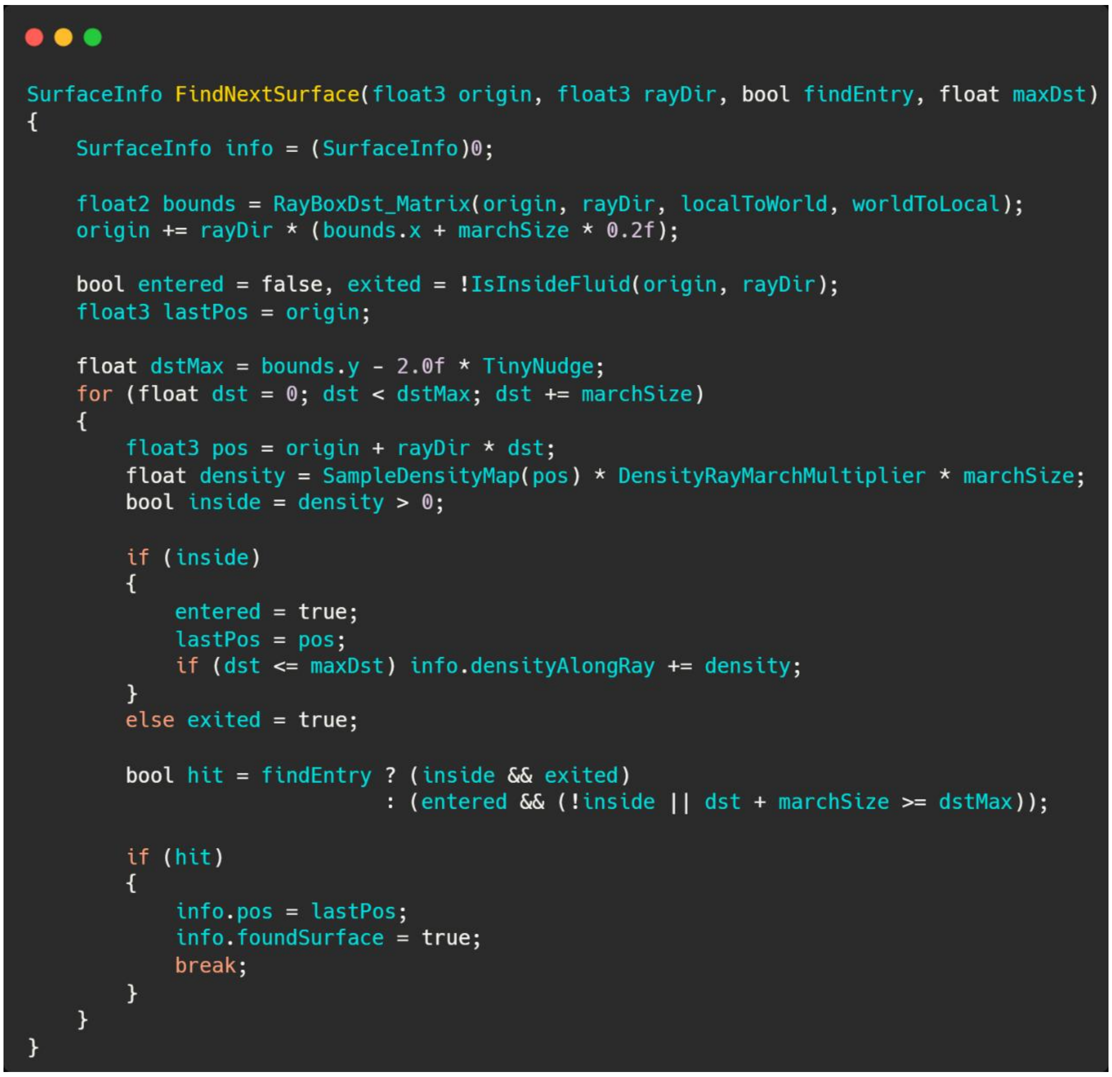
Figure 45 Pseudocode illustrating the normal calculation at the fluid surface hit point.



100.000	densityGra
0.000	volumeValu
1.000	DensityDep
135.000	SphereRadi
0.100	ISO Level
✓ Draw Fluid ?	
2.500	Water Turb
3	MaxRayBou

Figure 46 Normals of the Fluid surface calculated via interpolation. The 3 components of the normal vectors are assigned separately to a different color channel (RGB).

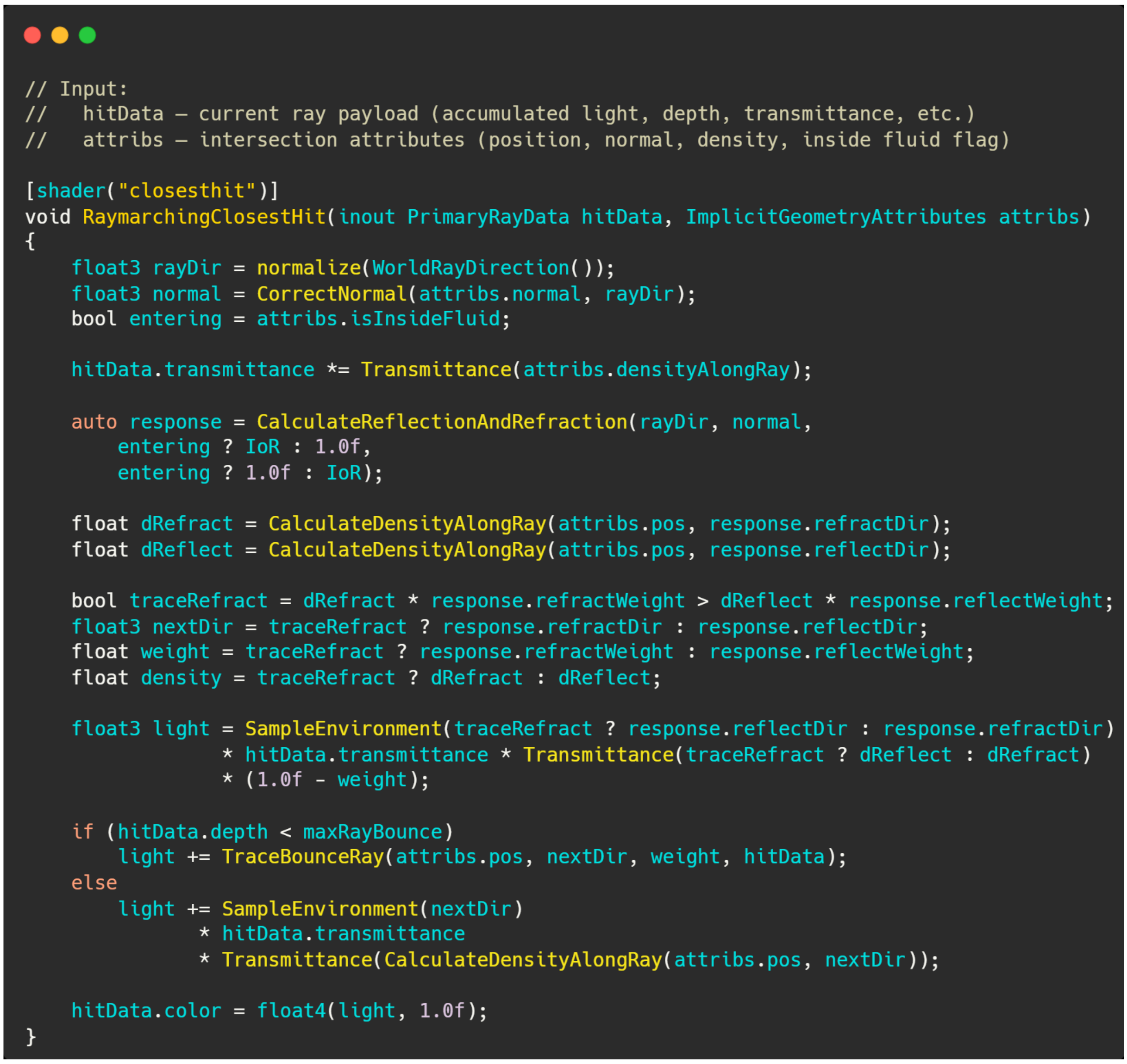
The next step involves extending the algorithm to perform volumetric traversal, enabling light to interact within the fluid medium through internal reflection and refraction. At each interface, rays are split according to Fresnel effects: a reflected ray and a refracted ray are recursively traced. A dedicated function, *FindNextSurface*, determines the next intersection by taking into account whether the ray starts inside or outside the fluid:



```
SurfaceInfo FindNextSurface(float3 origin, float3 rayDir, bool findEntry, float maxDst)
{
    SurfaceInfo info = (SurfaceInfo)0;

    float2 bounds = RayBoxDst_Matrix(origin, rayDir, localToWorld, worldToLocal);
    origin += rayDir * (bounds.x + marchSize * 0.2f);

    bool entered = false, exited = !IsInsideFluid(origin, rayDir);
    float3 lastPos = origin;

    float dstMax = bounds.y - 2.0f * TinyNudge;
    for (float dst = 0; dst < dstMax; dst += marchSize)
    {
        float3 pos = origin + rayDir * dst;
        float density = SampleDensityMap(pos) * DensityRayMarchMultiplier * marchSize;
        bool inside = density > 0;

        if (inside)
        {
            entered = true;
            lastPos = pos;
            if (dst <= maxDst) info.densityAlongRay += density;
        }
        else exited = true;

        bool hit = findEntry ? (inside && exited)
                             : (entered && (!inside || dst + marchSize >= dstMax));

        if (hit)
        {
            info.pos = lastPos;
            info.foundSurface = true;
            break;
        }
    }
}
```

Figure X - Pseudocode illustrating the volumetric ray marching algorithm implemented in a function called FindNextSurface used by the custom intersection shader.

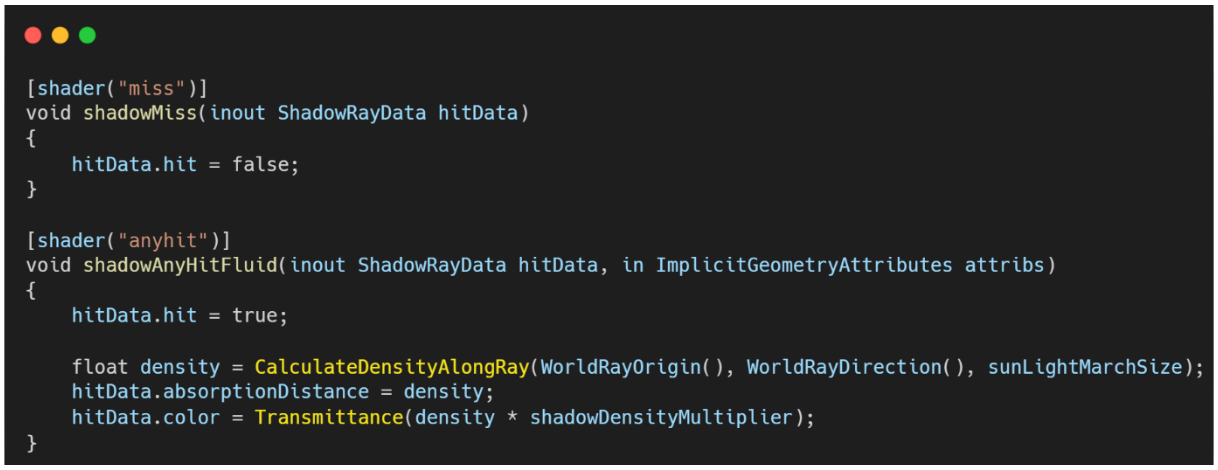
After a surface hit is reported, the closest hit shader executes a shading routine governed by the Fresnel coefficients. First, the transmitted ray is partially attenuated to simulate absorption within the medium. Then, reflection and refraction weights are computed, and a recursive ray is traced along the dominant direction. If the maximum recursion depth is reached, the shader samples the scene cubemap as an approximation for the outgoing ray. Otherwise, recursion continues by spawning a new ray with updated parameters.



```
// Input:
//   hitData – current ray payload (accumulated light, depth, transmittance, etc.)
//   attribs – intersection attributes (position, normal, density, inside fluid flag)

[shader("closesthit")]
void RaymarchingClosestHit(inout PrimaryRayData hitData, ImplicitGeometryAttributes attribs)
{
    float3 rayDir = normalize(WorldRayDirection());
    float3 normal = CorrectNormal(attribs.normal, rayDir);
    bool entering = attribs.isInsideFluid;

    hitData.transmittance *= Transmittance(attribs.densityAlongRay);

    auto response = CalculateReflectionAndRefraction(rayDir, normal,
        entering ? IoR : 1.0f,
        entering ? 1.0f : IoR);

    float dRefract = CalculateDensityAlongRay(attribs.pos, response.refractDir);
    float dReflect = CalculateDensityAlongRay(attribs.pos, response.reflectDir);

    bool traceRefract = dRefract * response.refractWeight > dReflect * response.reflectWeight;
    float3 nextDir = traceRefract ? response.refractDir : response.reflectDir;
    float weight = traceRefract ? response.refractWeight : response.reflectWeight;
    float density = traceRefract ? dRefract : dReflect;

    float3 light = SampleEnvironment(traceRefract ? response.reflectDir : response.refractDir)
                 * hitData.transmittance * Transmittance(traceRefract ? dReflect : dRefract)
                 * (1.0f - weight);

    if (hitData.depth < maxRayBounce)
        light += TraceBounceRay(attribs.pos, nextDir, weight, hitData);
    else
        light += SampleEnvironment(nextDir)
               * hitData.transmittance
               * Transmittance(CalculateDensityAlongRay(attribs.pos, nextDir));

    hitData.color = float4(light, 1.0f);
}
```

Figure 47 Pseudocode for the closest hit shader implementing recursive ray tracing using Fresnel-based reflection and refraction.

For shadows, a separate hit group is defined with a dedicated any-hit and miss shader, allowing shadow rays to pass through low-density regions while being blocked by dense areas, which contributes to soft volumetric shadows.



```
[shader("miss")]
void shadowMiss(inout ShadowRayData hitData)
{
    hitData.hit = false;
}

[shader("anyhit")]
void shadowAnyHitFluid(inout ShadowRayData hitData, in ImplicitGeometryAttributes attribs)
{
    hitData.hit = true;

    float density = CalculateDensityAlongRay(WorldRayOrigin(), WorldRayDirection(), sunLightMarchSize);
    hitData.absorptionDistance = density;
    hitData.color = Transmittance(density * shadowDensityMultiplier);
}
```

Figure 48 Structure of the shadow hit group used to handle volumetric shadows in the ray marching pipeline

The final result demonstrates a fluid surface with both external and internal interactions of light, capturing key optical phenomena such as transparency, colored absorption, and smooth refraction (See figure 49).

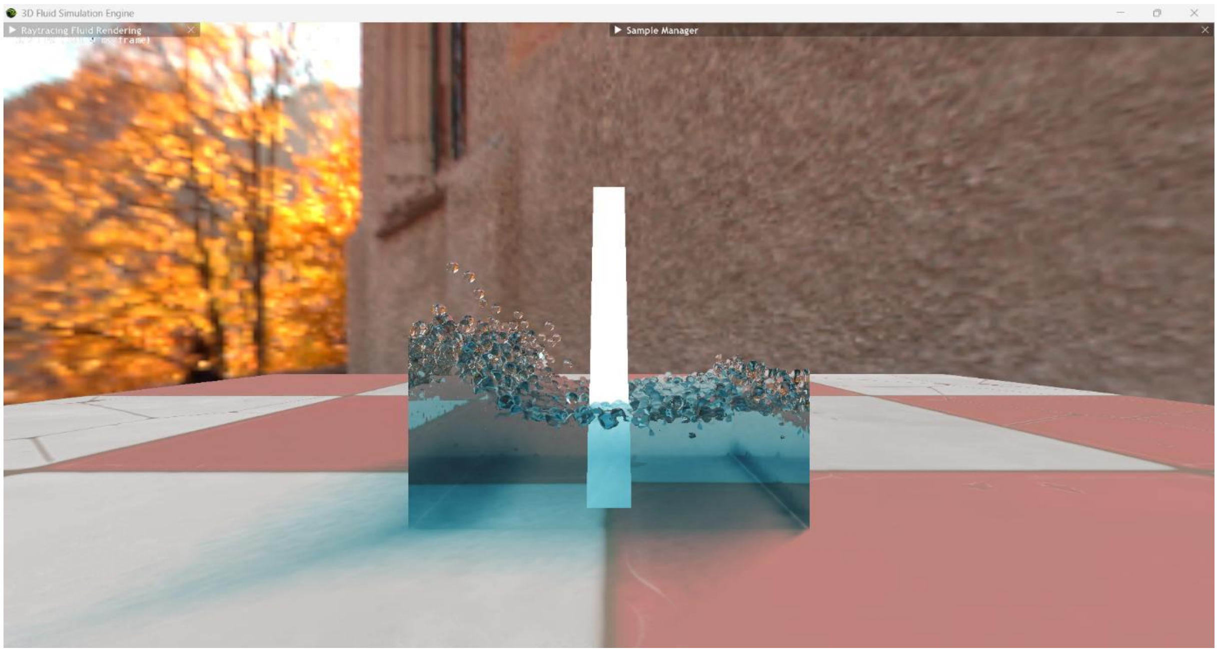


Figure 49 Final visual output of the fluid rendering using ray marching in the DXR pipeline.

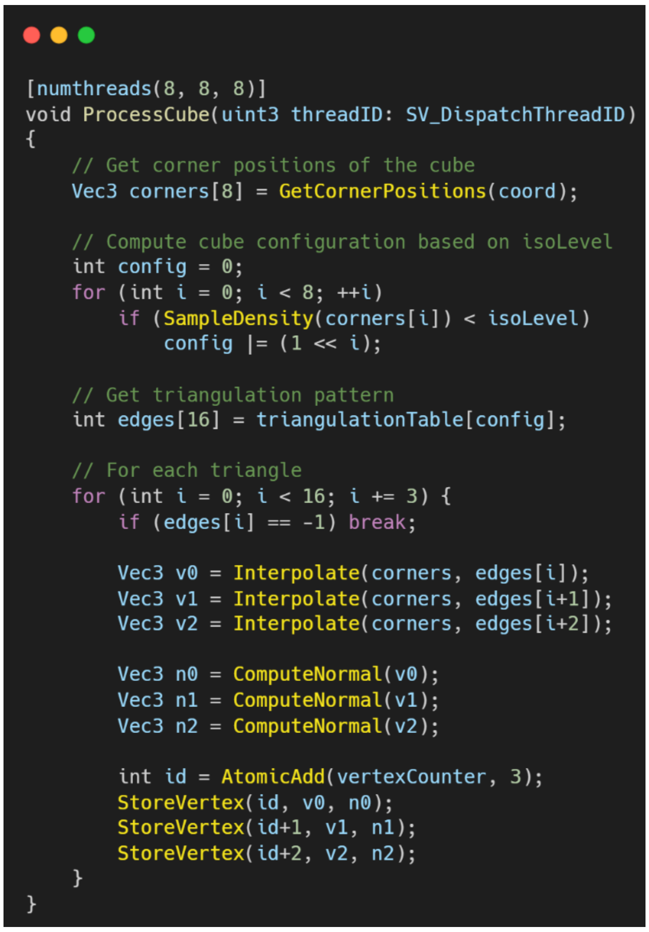
4.2.5 Marching Cubes

Compared to ray marching, the Marching Cubes algorithm requires fewer custom ray tracing shaders, as it produces explicit triangle geometry that can be processed directly by DXR's built-in intersection logic. However, one challenge lies in the dynamic nature of the output: the number of vertices generated by the algorithm varies from frame to frame depending on the fluid's surface topology.

Since vertex buffers in DirectX 12 cannot be dynamically resized at runtime, a conservative upper bound must be pre-allocated. This upper bound is computed based on the resolution of the voxel grid, assuming a worst-case scenario of one triangle (three vertices) per voxel:

$$Max\ vertices = 3 \cdot (density\ map\ resolution)^3$$

The mesh is then generated within a compute shader, which executes the Marching Cubes logic in parallel across the voxel grid. For each voxel, the algorithm evaluates the density at the eight corners to determine the cube configuration, retrieves the corresponding triangulation pattern, and interpolates the vertex positions along active edges. The computed vertices and normals are stored in a buffer using atomic operations to avoid race conditions. A concise version of this logic is presented as pseudocode in Figure 50.



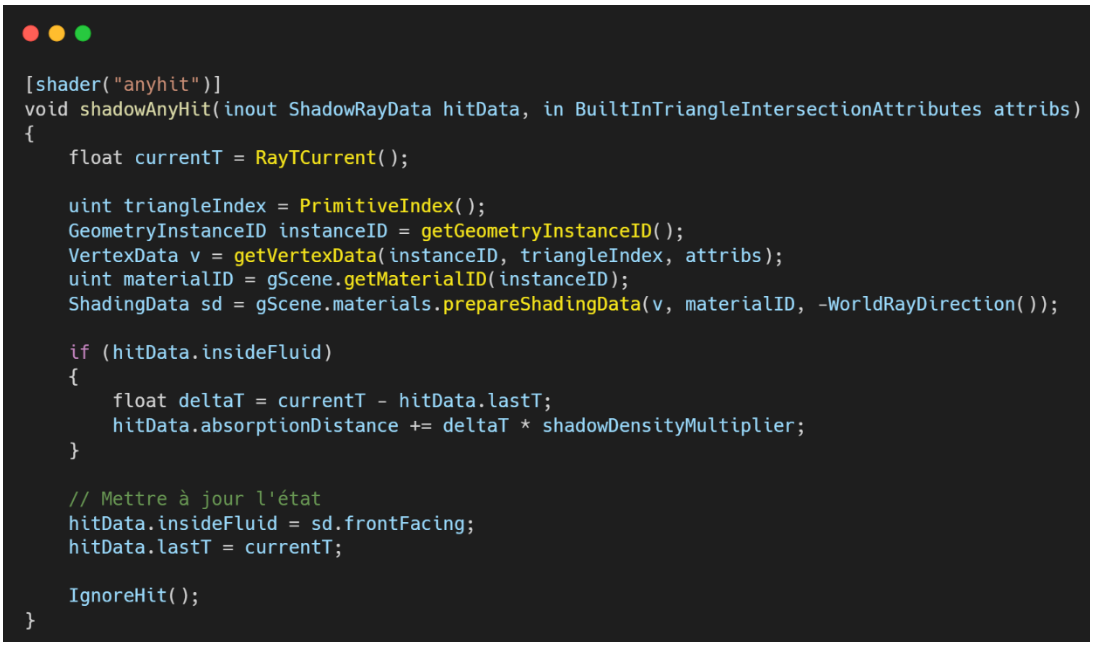
```
[numthreads(8, 8, 8)]
void ProcessCube(uint3 threadID: SV_DispatchThreadID)
{
    // Get corner positions of the cube
    Vec3 corners[8] = GetCornerPositions(coord);

    // Compute cube configuration based on isoLevel
    int config = 0;
    for (int i = 0; i < 8; ++i)
        if (SampleDensity(corners[i]) < isoLevel)
            config |= (1 << i);

    // Get triangulation pattern
    int edges[16] = triangulationTable[config];

    // For each triangle
    for (int i = 0; i < 16; i += 3) {
        if (edges[i] == -1) break;

        Vec3 v0 = Interpolate(corners, edges[i]);
        Vec3 v1 = Interpolate(corners, edges[i+1]);
        Vec3 v2 = Interpolate(corners, edges[i+2]);

        Vec3 n0 = ComputeNormal(v0);
        Vec3 n1 = ComputeNormal(v1);
        Vec3 n2 = ComputeNormal(v2);

        int id = AtomicAdd(vertexCounter, 3);
        StoreVertex(id, v0, n0);
        StoreVertex(id+1, v1, n1);
        StoreVertex(id+2, v2, n2);
    }
}
```

Figure 50 Pseudocode of the Marching Cubes compute shader.

Once the mesh has been generated and transferred to the CPU, the vertex buffer must be uploaded back to the GPU for acceleration structure construction. This step is necessary to build the Bottom-Level Acceleration Structure (BLAS) required by DXR’s intersection shaders. Unfortunately, unlike rasterization pipelines, this process cannot be optimized using indirect draw calls and must be repeated every frame.

Beyond this geometry construction step, the closest hit shader used for shading the surface reuses the dielectric material logic previously developed for the ray marching pipeline. This modularity allows the Marching Cubes method to be integrated more quickly, since it leverages existing shading code for reflection, refraction, and absorption within dielectric media.

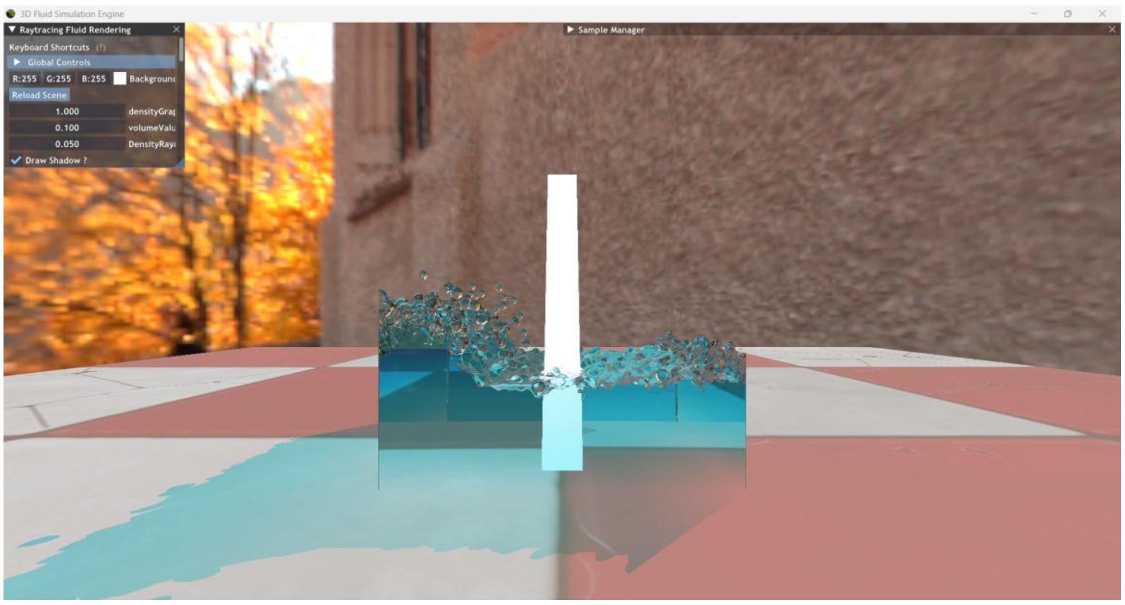
The main difference lies in the shadow computation. Unlike ray marching, which can accumulate absorption directly along volumetric paths, Marching Cubes requires an explicit any-hit shader to simulate the attenuation of light as it travels through the mesh. This shader integrates absorption based on the length of the shadow ray's path through the fluid. The corresponding pseudocode is illustrated in Figure 51.



```
[shader("anyhit")]
void shadowAnyHit(inout ShadowRayData hitData, in BuiltInTriangleIntersectionAttributes attribs)
{
    float currentT = RayTCurrent();

    uint triangleIndex = PrimitiveIndex();
    GeometryInstanceID instanceID = getGeometryInstanceID();
    VertexData v = getVertexData(instanceID, triangleIndex, attribs);
    uint materialID = gScene.getMaterialID(instanceID);
    ShadingData sd = gScene.materials.prepareShadingData(v, materialID, -WorldRayDirection());

    if (hitData.insideFluid)
    {
        float deltaT = currentT - hitData.lastT;
        hitData.absorptionDistance += deltaT * shadowDensityMultiplier;
    }

    // Mettre à jour l'état
    hitData.insideFluid = sd.frontFacing;
    hitData.lastT = currentT;

    IgnoreHit();
}
```

Figure 51 Shadow Any Hit shader for marching cube fluid.

The final visual result is shown in Figure 52. While the output appears clean and geometrically sharp, it lacks the soft transitions and density-based variations of volumetric ray marching. Additionally, shadows tend to appear flat and uniform, as the geometric mesh no longer carries information about gradual changes in density.


3D Fluid Simulation Engine
Raytracing Fluid Rendering
Keyboard Shortcuts (?)
Global Controls
R:255 G:255 B:255 Backgrounc
Reload Scene
Sample Manager

1.000 densityGrap
0.100 volumeValu
0.050 DensityRay
✓ Draw Shadow ?

Figure 52 Final visual output of the fluid rendering using marching cubes in the DXR pipeline.

4.3 Project summary

Figure 53 presents a visual comparison between the two rendering techniques explored in this project: ray marching (left) and Marching Cubes (right). The ray marched rendering produces a significantly more realistic appearance, particularly in terms of internal density variation and optical depth within the fluid volume.

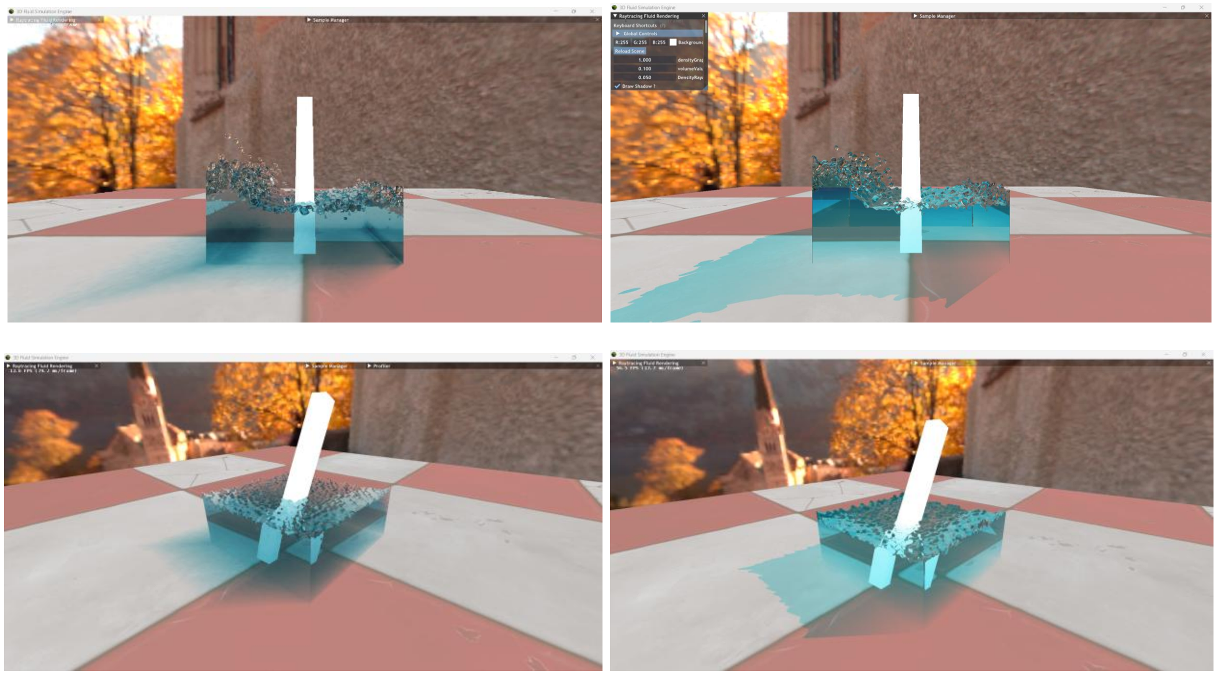


Figure 53 Comparison between ray marching (left) and Marching Cubes (right) rendering results.

A more detailed optical analysis of both implementations is provided in Appendix A, where the visual output of this Media Project is compared to selected industrial fluid rendering examples as well as real-world reference photographs. This comparative study highlights both the visual plausibility and physical consistency of the techniques developed.

From a technical standpoint, the core objectives of the project have been successfully achieved. Both ray marching and Marching Cubes were fully integrated into a real-time DXR-based ray tracing pipeline, and all targeted optical effects—including reflection, refraction, light absorption (via Beer's Law), and Fresnel reflection—were correctly implemented and validated within each method.

These results confirm the feasibility of rendering implicit, dynamic fluids with complex optical behavior in a real-time ray tracing context, while also illustrating the trade-offs inherent to each technique in terms of visual fidelity, performance, and implementation complexity.

5. Quantitative Analysis

This chapter evaluates the performance of two rendering techniques, ray marching and marching cubes, applied to a real-time SPH simulation using a ray tracing pipeline. Tests were run on a 1920×1055 viewport with 20,000 particles. The analysis considers frame time (in milliseconds), geometry memory usage (megabytes), and the cost of BLAS and TLAS updates (in milliseconds), under varying ray bounce counts (3–5) and density map voxels count (64^3, 128^3, 256^3).

Each test spans 10 seconds of simulation, allowing the fluid to evolve from an airborne state to a stable one. This setup covers diverse conditions, from uniform fluid volumes to splash-heavy dynamics, helping assess how screen coverage and fluid complexity affect ray traversal and performance.

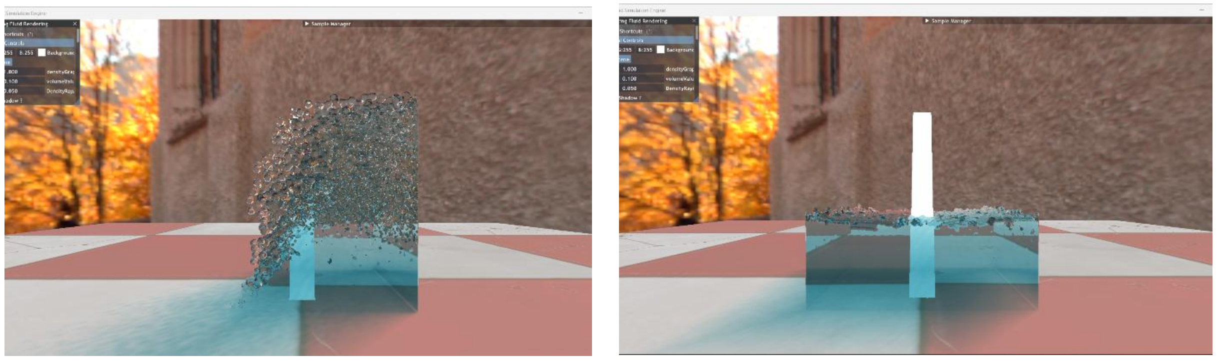


Figure 54 Rendering of the SPH simulation (using ray marching) after 2 seconds (left) and 8 seconds (right).

5.1 Ray marching Analysis

To ensure consistent visual quality, the ray marching step size was scaled proportionally from Seb Lague's implementation. Since the simulation domain is roughly 2.6 times larger, the step size was increased from 0.02 to 0.05 and kept constant across tests to isolate the effects of bounce count and voxel resolution.

5.1.1 Frame Analysis

Table 1 reports the average execution time (in milliseconds) of each stage in the rendering pipeline, measured across full frame. The tested configurations vary by the maximum number of ray bounces (3, 4, or 5) and the resolution of the density map used (64^3, 128^3, and 256^3 voxels).

	Max ray bounce: 3 **Density map: 64^3 voxels**	**Max ray bounce: 4** **Density map: 128^3 voxels**	**Max ray bounce: 5** **Density map: 256^3 voxels**
SPH Update	3.21 ms	3.28 ms	3.17 ms
Compute Density Map	0.33 ms	1.54 ms	10.64 ms
BLAS build	0.01 ms	0.01 ms	0.01 ms
TLAS build	0.04 ms	0.04 ms	0.04 ms
Ray tracing **(Ray marching in intersection shader)**	32.33 ms	58.71 ms	90.17 ms
Total Frame	35.92 ms	63.53 ms	103.88 ms

Table 1. Average execution time per stage (in ms) under varying ray bounce counts and density map resolutions using the ray marching algorithm.

Although not central to this study, the SPH Update step is included to reflect total frame cost in a realistic simulation and remains constant across settings. The density map generation step, converting SPH particles to a voxel grid, scales rapidly with resolution due to the cubic growth in voxel count, from 0.33 ms at 64^3 to 10.64 ms at 256^3.

BLAS build time is negligible (0.01 ms), as the scene contains only a single custom primitive. TLAS build incurs a small cost (0.04 ms), reflecting matrix transformation overhead.

The ray tracing pass dominates the frame cost, increasing sharply with both bounce count and density map resolution. From 32.33 ms at 3 bounces and 64^3, the cost rises to 90.17 ms at 5 bounces and 256^3. This confirms that recursive ray traversal in dense volumetric data is the main performance bottleneck in a rendering using ray marching.

5.1.2 Ray tracing Pass Analysis

While Table 1 focused on average frame times, Table 2 provides a statistical breakdown (min, max, mean, std dev) of the ray tracing pass for the under different test configurations.

	Density map: 64^3 voxels	**Density map: 128^3 voxels**	**Density map: 256^3 voxels**
Max ray bounces: 3	Min: 23.88 Max: 57.19 Mean: 32.33 Std dev: 9.12	Min: 30.07 Max: 79.00 Mean: 40.10 Std dev: 12.75	Min: 43.32 Max: 113.86 Mean: 57.25 Std dev: 18.12
Max ray bounces: 4	Min: 29.57 Max: 114.86 Mean: 46.66 Std dev: 22.15	Min: 35.35 Max: 144.41 Mean: 58.71 Std dev: 31.73	Min: 54.88 Max: 199.58 Mean: 79.69 Std dev: 37.01
Max ray bounces: 5	Min: 29.22 Max: 216.51 Mean: 57.88 Std dev: 46.13	Min: 36.63 Max: 277.02 Mean: 76.45 Std dev: 59.25	Min: 50.88 Max: 341.63 Mean: 90.17 Std dev: 64.01

Table 2. Minimum, maximum, mean, and standard deviation (in milliseconds) of the ray tracing pass across different configurations using the ray marching algorithm.

The minimum times correspond to visually stable frames, e.g., after settling, where rays encounter minimal density changes and refraction, leading to efficient traversal. In these cases, 4 of the 9 configurations reach real-time performance (less than or equal to 33.33 ms), with 2 near that threshold.

In contrast, maximum values arise during dynamic fluid states with splashes, where many small droplets trigger costly recursive refractions. These cases show extreme slowdowns, reaching up to 341.63 ms. High standard deviation values in high-resolution or high-bounce scenarios further highlight performance instability as fluid complexity increases.

The data reveals a shift in the limiting factor based on configuration. At low resolution (64^3), increasing bounce count from 3 to 5 causes a 79% rise in average ray tracing time (32.33 ms to 57.88 ms), indicating a bounce-bound regime. At high resolution (256^3), increasing bounce count has less impact (+57%), while simply increasing resolution from 64^3 to 256^3 at 3 bounces raises cost by 77%, making sampling the main bottleneck.

5.1.3 Summary

Overall, the analysis shows that ray marching performance is limited more by the algorithm than the DXR pipeline. Real-time performance is only feasible in stable scenes with compact fluid volumes. In dynamic conditions, recursive refractions from dispersed particles severely impact performance. Low-resolution volumes are limited by bounce depth, while high-resolution volumes are limited by sampling cost. This dual limitation makes ray marching difficult to scale reliably for real-time rendering of complex fluid scenes.

5.2 Marching Cubes Analysis

Real-time applications require a fixed vertex buffer size, as dynamic resizing is not viable during rendering. To accommodate the varying topology generated by the Marching Cubes algorithm, a conservative upper bound on vertex count is pre-allocated based on the voxel grid resolution (assuming at most one triangle per voxel):

$$Max\ vertices = 3 \cdot (density\ map\ resolution)^3$$

5.2.1 Frame Analysis

Table 3 compares execution times (in milliseconds) across pipeline stages, highlighting how performance scales with density map resolution and ray recursion depth.

	Max ray bounce: 3 **Density map: 64^3 voxels**	**Max ray bounce: 4** **Density map: 128^3 voxels**	**Max ray bounce: 5** **Density map: 256^3 voxels**
SPH Update	3.23 ms	3.19 ms	3.25 ms
Compute Density Map	0.35 ms	1.51 ms	11.38 ms
Compute Marching Cubes	0.26 ms	1.80 ms	131.01 ms
Upload Vertices	0.45 ms	3.01 ms	218.19 ms
BLAS build	1.11 ms	8.94 ms	67.80 ms
TLAS build	0.04 ms	0.04 ms	0.04 ms
Ray tracing	2.37 ms	4.90 ms	14.58 ms
Total Frame	7.81	23.29	446.11

Table 3. Average execution time per stage (in ms) under varying ray bounce counts and density map resolutions using the marching cubes algorithm.

As in the ray marching pipeline, the SPH Update stage is included for completeness, although it is not part of the rendering process. Its cost remains stable across all configurations (~3.2 ms), unaffected by resolution or bounce count.

Similarly, the Compute Density Map stage, responsible for voxelizing particle data, shows the expected cubic scaling with resolution.

The Marching Cubes stage scales cubically with resolution, from 0.26 ms at 64^3 voxels to 131.01 ms at 256^3 voxels, due to the volumetric nature of the grid. Geometry upload to the GPU becomes a major bottleneck at high resolutions, reaching 218.19 ms at 256^3 voxels. The BLAS build stage also scales with triangle count, peaking at 67.80 ms, whereas TLAS remains negligible due to the use of a single instance.

Interestingly, the ray tracing pass remains relatively efficient across all configurations, averaging 14.58 ms at the highest settings (256^3, 5 bounces), thanks to hardware acceleration. Overall, performance in this pipeline is primarily limited by memory bandwidth and geometry complexity, not by ray recursion.

5.2.2 Geometry Memory Analysis

Table 4 presents the evolution of geometry memory consumption as a function of the density map resolution.

	Density map: 64^3 voxels	**Density map: 128^3 voxels**	**Density map: 256^3 voxels**
Vertex Count	750,183	6,145,191	49,744,167
Vertex Buffer Memory	22.89 MB	187.54 MB	1480 MB
BLAS Memory	13.39 MB	109.68 MB	887.83 MB
TLAS Memory	0,00212 MB	0,00212 MB	0,00212 MB
Total Geometry Memory	78.94 MB	415.98 MB	3070 MB

Table 4. Geometry Memory (in vertex count and mb) varying density map resolutions.

The vertex count and memory usage increase cubically with resolution, from 750k vertices / 22.89 MB at 64^3 to nearly 50 million / 1.48 GB at 256^3. While this growth is expected, the BLAS memory footprint becomes a critical concern, expanding from 13.39 MB to 887.83 MB. TLAS memory remains negligible. Combined, total geometry-related memory usage reaches approximately 3 GB at 256^3, revealing a clear scalability limit for real-time rendering with dense meshes.

5.2.3 Ray tracing Pass Analysis

Table 5 provides a statistical breakdown (min, max, mean, std dev) of the ray tracing pass for the under different test configurations.

	Density map resolution = 64^3	Density map resolution = 128^3	Density map resolution = 256^3
Max nbr of ray bounce = 3	Min: 1.92 Max: 8.92 Mean: 2.36 Std dev: 1.07	Min: 2.30 Max: 12.11 Mean: 3.22 Std dev: 2.13	Min: 3.00 Max: 16.18 Mean: 6.13 Std dev: 4.35
Max nbr of ray bounce = 4	Min: 2.16 Max: 16.27 Mean: 2.93 Std dev: 2.06	Min: 2.65 Max: 22.71 Mean: 4.90 Std dev: 4.58	Min: 3.59 Max: 29.97 Mean: 9.77 Std dev: 8.17
Max nbr of ray bounce = 5	Min: 2.32 Max: 29.35 Mean: 3.49 Std dev: 3.57	Min: 3.15 Max: 40.94 Mean: 6.44 Std dev: 8.26	Min: 3.99 Max: 78.05 Mean: 14.58 Std dev: 14.96

Table 5. Minimum, maximum, mean and standard deviation (milliseconds) of the ray tracing pass across different configurations using the marching cubes algorithm.

The minimum ray tracing times correspond to visually stable frames, such as after the fluid has settled. In these cases, the mesh remains compact and smooth, with relatively few triangles and limited refraction, allowing for fast traversal.

In contrast, maximum values occur during dynamic events like splashes, where the fluid fragments into many small surfaces. This leads to a dense and irregular mesh, significantly increasing the number of intersection tests and recursive refractions, and causing major slowdowns.

The standard deviation rises with resolution, reflecting increased variability as mesh complexity grows. Overall, performance in the ray tracing stage is primarily driven by geometry complexity, which depends on both fluid motion and voxel resolution.

5.2.4 Summary

This analysis confirms that in the marching cubes rendering, the main performance bottleneck during ray tracing comes from the complexity of the generated mesh, not from the recursion depth. When the fluid is stable and compact, performance remains efficient. However, during splashes, the fluid breaks into many small fragments, leading to a dramatic increase in triangle count and costly recursive refractions. The density map resolution directly controls the mesh complexity, causing both the mean cost and variability of the ray tracing pass to increase. As a result, geometry complexity, not the number of ray bounces, is the dominant factor limiting performance in this approach.

5.3 Comparative Analysis

Table 6 summarizes the average frame time for both rendering approaches, ray marching and marching cubes, across increasing quality settings

	3 ray bounces, DensityMapSize 64^3px	4 ray bounces DensityMapSize 128^3px	5 ray bounces DensityMapSize 256^3px
Ray marching	SPH: 3.21ms Density Map: 0.33ms BLAS build: 0.01ms TLAS build: 0.04ms Ray tracing: 32.33ms Total: 35.92ms	SPH: 3.28ms Density Map: 1.54ms BLAS build: 0.01ms TLAS build: 0.04ms Ray tracing: 58.71ms Total: 63.53ms	SPH: 3.07ms Density Map: 10.64ms BLAS build: 0.01ms TLAS build: 0.04ms Ray tracing: 90.17ms Total: 103.88ms
Marching Cubes	SPH: 3.23ms Density Map: 0.35ms MC algo: 0.26ms Upload Vertices: 0.45ms BLAS build: 1.11ms TLAS build: 0.04ms Ray tracing: 2.37ms Total: 7.81ms	SPH: 3.09ms Density Map: 1.51ms MC algo 1.80ms: Upload Vertices 3.01ms: BLAS build: 8.94ms TLAS build: 0.04ms Ray tracing: 4.9ms Total: 23.29ms	SPH: 3.11ms Density Map: 11.38ms MC algo: 131.01 Upload Vertices: 218.19 BLAS build: 67.80ms TLAS build: 0.04ms Ray tracing: 14.58ms Total: 446.11ms

Table 6. Average execution time per stage (in ms) under varying ray bounce counts and density map resolutions using the ray marching and the marching cubes algorithm.

In the two lower-quality configurations, marching cubes outperforms ray marching, with speedups of 78.26% and 63.34%, respectively. This advantage comes from its use of triangle meshes, which are efficiently handled by modern GPUs through RT core acceleration. Triangle-ray intersections are highly optimized, resulting in fast traversal even with moderately complex geometry.

However, at the highest quality level, where the voxel resolution and visual complexity are greatest, ray marching becomes 76.71% faster. Marching cubes suffers from excessive geometry generation and GPU memory pressure, particularly due to the cost of uploading tens of millions of vertices and building acceleration structures. Ray marching, by contrast, avoids these stages by directly sampling the voxel grid during rendering.

Despite its performance advantage in some cases, marching cubes produces a less visually accurate representation of fluid, especially during dynamic events like splashes or surface breakup. The triangulated mesh cannot fully capture the continuous, semi-transparent nature of fluid volumes, which ray marching handles more naturally through volumetric integration and soft density gradients. In scenes where visual realism is critical, such as underwater effects or translucent volumes, ray marching offers a more convincing appearance, albeit at higher cost.

In summary, marching cubes is better suited for real-time rendering with compact or moderately dynamic fluids, offering consistent performance. Ray marching, while more expensive, is better at reproducing realistic volumetric effects, and becomes preferable at high resolutions where mesh-based methods struggle. However, its performance remains highly dependent on the visible size of the fluid in the frame, leading to greater variability in dynamic scenes.

6. Conclusion and Further Work

This thesis investigated the core challenges of integrating a dynamic implicit fluid, simulated via Smoothed Particle Hydrodynamics (SPH), into a real-time raytracing pipeline using Microsoft's DXR API. Through a combination of literature review, expert interviews, practical implementation, and quantitative performance analysis, it became evident that representing fluid volumes in a raytraced context presents a multidimensional set of constraints, involving data structure design, memory bandwidth, rendering cost, and visual fidelity.

From an academic standpoint, the state of the art revealed a strong dichotomy between surface-based and volume-based fluid rendering techniques. While traditional mesh extraction methods like Marching Cubes remain dominant in raster-based pipelines due to their hardware efficiency and integration with existing rendering engines, volume rendering methods such as ray marching offer more physically plausible visuals, especially for translucent or gaseous effects. However, both approaches encounter significant limitations when translated into a real-time raytracing context, an environment less forgiving in terms of performance and highly dependent on the efficient use of GPU acceleration structures.

Expert interviews confirmed these theoretical insights: modern real-time graphics pipelines must constantly balance between performance, scalability, and visual expressiveness. According to the professionals consulted, volume rendering remains largely experimental in production raytraced pipelines, while triangle-based representations are still favored for their maturity and GPU-friendly nature.

To explore this trade-off in practice, a custom DXR rendering prototype was developed, capable of visualizing an animated SPH simulation using both ray marching and marching cubes. This case study made it possible to directly compare both techniques under identical simulation and rendering conditions. The results showed that marching cubes offers significantly more stable and predictable performance at lower resolutions, benefiting from RT core acceleration. However, its mesh-based representation fails to capture the soft, continuous appearance of fluids, especially in dynamic states. Ray marching, in contrast, produced more realistic results through volumetric integration but showed high performance variability, particularly when the fluid occupied a large portion of the screen.

Quantitative analysis confirmed that each approach presents a different performance bottleneck: marching cubes is geometry-bound, particularly at high resolution due to vertex upload and BLAS construction, while ray marching is sampling-bound and highly sensitive to bounce count and voxel resolution. These findings point to a clear trade-off between visual fidelity and temporal stability, with no one-size-fits-all solution.

In conclusion, integrating dynamic implicit fluids into real-time raytracing remains an open challenge, particularly due to the inherent tension between mesh-based and volume-based representations. While DXR offers powerful low-level control and hardware acceleration, it also imposes strict performance constraints. However, the actual cost of raytraced rendering is not fixed, it heavily depends on the fluid's spatial extent and dynamism. When the fluid remains compact or settled, performance remains manageable, but during turbulent phases with widespread droplets, the cost can rise drastically due to increased sampling or geometric complexity.

Future work could focus on optimized variants of both rendering techniques. For ray marching, methods like Sparse Voxel Surfaces (SVS) could significantly reduce sampling overhead while preserving volumetric appearance. For marching cubes, decomposing the fluid into multiple sub-meshes could prevent oversized vertex buffers and reduce the cost of BLAS updates by limiting them to localized regions. Hybrid representations, adaptive sampling strategies, or screen-space approximations also offer promising directions to mitigate current limitations. This study provides a grounded technical foundation for further research in physically plausible fluid rendering within real-time raytracing environments.

Sources

Bibliography

Crane, K., Lentine, M., & Fedkiw, R. (2007). Real-time simulation and rendering of 3D fluids. In H. Nguyen (Ed.), *GPU Gems 3* (Chap. 30). Addison-Wesley. https://dl.acm.org/citation.cfm?id=1407436

Haines, E., & Akenine-Möller, T. (Eds.). (2019). *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*. Apress. https://doi.org/10.1007/978-1-4842-4427-2

Shirley, P., Parker, S., Jarosz, W., & Wyman, C. (2021). The Schlick Fresnel approximation. In T. Akenine-Möller, E. Haines, N. Hoffman, A. Patney, & M. Pharr (Eds.), *Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX* (pp. 109–114). Apress. https://doi.org/10.1007/978-1-4842-7185-8_9

Filmography

A Bug's Life. (1998). [Film]. Pixar Animation Studios / Walt Disney Pictures.

Monster House. (2006). [Film]. ImageMovers / Columbia Pictures.

Avatar: The Way of Water. (2022). [Film]. Lightstorm Entertainment / 20th Century Studios.

Ludography

CD Projekt RED. (2020). Cyberpunk 2077 [Video game]. CD-Projekt.

DICE. (2018). Battlefield V [Video game]. Electronic Arts.

Game Science. (2024). Black Myth: Wukong [Video game]. Game Science.

Scientific Articles

Appel, A. (1968). *Some Techniques for Shading Machine Renderings of Solids*. AFIPS Spring Joint Computer Conference. https://doi.org/10.1145/1468075.1468082

Ihmsen, M., Orthmann, J., Solenthaler, B., Kolb, A., & Teschner, M. (2012). *Unified spray, foam and air bubbles for particle-based fluids*. Computer Graphics Forum, 31(2), 965–974. https://doi.org/10.1111/j.1467-8659.2012.03089.x

Kajiya, J. T. (1986). The rendering equation. *ACM SIGGRAPH Computer Graphics*, *20*(4), 143-150. https://doi.org/10.1145/15886.15902

Lorensen, W. E., & Cline, H. E. (1987). Marching cubes : A high resolution 3D surface construction algorithm. *ACM SIGGRAPH Computer Graphics*, *21*(4), 163-169. https://doi.org/10.1145/37402.37422

Whitted, T. (1979). An improved illumination model for shaded display. *ACM SIGGRAPH Computer Graphics*, *13*(2), 14.
https://doi.org/10.1145/965103.807419

Xu, Y., Xu, Y., Xiong, Y., Yin, D., Ban, X., Wang, X., Chang, J., & Zhang, J. J. (2022). Anisotropic screen space rendering for particle-based fluid simulation. *Computers & Graphics*, *110*, 118-124.
https://doi.org/10.1016/j.cag.2022.12.007

Webography

BELSPO. (n.d.). *que peut-on observer et comment Comment les rayonnements interagissent avec les objets sur Terre ?* [Article].
https://eo.belspo.be/fr/introduction-quest-ce-que-la-teledetection

Brucks, R. (2016, November 16). *Shader Bits HLSL and UE4 Development Blog: Creating a Volumetric Ray Marcher.* [Blog Post]
https://shaderbits.com/blog/creating-volumetric-ray-marcher

Burnes, A. (2023, April 11). *Cyberpunk 2077: Technology preview of new ray tracing overdrive mode out now*. NVIDIA.
https://www.nvidia.com/en-us/geforce/news/cyberpunk-2077-ray-tracing-overdrive-update-launches-april-11/

Caulfield, B. (2022, March 23). *What is Path tracing?* [Blog Post].
https://blogs.nvidia.com/blog/what-is-path-tracing/

demofox. (2017). *The blog at the bottom of the sea: Raytracing Reflection, Refraction, Fresnel, Total Internal Reflection, and Beer's Law* [Blog Post].
https://blog.demofox.org/2017/01/09/raytracing-reflection-refraction-fresnel-total-internal-reflection-and-beers-law/

Fleck, B. (2008). *Real-time rendering of water in computer graphics* Otto-von-Guericke University Magdeburg [Bachelor's thesis].
https://www.cg.tuwien.ac.at/courses/Seminar/WS2007/arbeit_fleck.pdf

Green, S. (2010). *Screen space fluid rendering for games* [Conference presentation slides]. NVIDIA.
https://developer.download.nvidia.com/presentations/2010/gdc/Direct3D_Effects.pdf

Hong, S., & Beets, K. (2020, June 8). *What is ray tracing and how is it enabling real-time 3D graphics?* EDN Asia [Article].
https://www.ednasia.com/what-is-ray-tracing-and-how-is-it-enabling-real-time-3d-graphics/

JangaFX. (2025). *LiquiGen – Real-time fluid simulation and rendering software*.
https://jangafx.com/software/liquigen

Lague, S. (2024, December 6). *Coding Adventure: Rendering Fluid* [Video]. YouTube.
https://youtu.be/kOkfC5fLfgE?si=fONsW_Tz3tmC83Jq

Lague, S. (2023, October 8). Coding Adventure: Simulatin Fluid [Video]. YouTube.
https://youtu.be/rSKMYc1CQHE?si=1X32-SDvfQy9AABU

Legouge, M. (2024, July 24). *Ray tracing : tout comprendre à cette technologie de rendu graphique en temps réel.* Frandroid [Article].
https://www.frandroid.com/dossiers/660693_dossier-ray-tracing-explication

Lehmann, M. (2022). *FluidX3D* [Computer software]. GitHub.
https://github.com/ProjectPhysX/FluidX3D

Moroz, M. (2025). *Zibra Liquids: A look at real-time ray tracing in Unity* [Interview]. 80.lv.
https://80.lv/articles/zibra-liquids-a-look-at-real-time-ray-tracing-in-unity

NVIDIA. (2020, March 23). *"Reflections" – A Star Wars UE4 Real-Time Ray Tracing Cinematic Demo | By Epic, ILMxLAB, and NVIDIA* [Video]. YouTube.
https://www.youtube.com/watch?v=lMSuGoYcT3s

Puchka, O. (2024, May 6) *Zibra Liquid Glossary (Common Zibra Liquid terms)* [Blog Post].
https://www.zibra.ai/blog-posts/zibra-liquid-glossary

Singh.Puja. (2021, August 26). *Explained! Refraction of light, How does refraction work with examples. SmartClass4Kids* [Article].
https://smartclass4kids.com/refraction-of-light/

Stampf. (2022, March). *Ray traced reflection brightness compared to Path Tracing* [Forum post with image]. Unreal Engine Forums.
https://forums.unrealengine.com/t/ray-traced-reflection-brightness-compared-to-path-tracing/509942

The Conversation. (2023, June 19). *Pourquoi la science des fluides est au cœur des défis du 21e siècle* [Article].
https://theconversation.com/pourquoi-la-science-des-fluides-est-au-coeur-des-defis-du-21e-siecle-204203

Wallisc. (2020, December 8). *Making of Moana (the Shadertoy*) [Blog Post].
https://wallisc.github.io/rendering/2020/12/08/Making-Of-Moana-the-shadertoy.html

https://jangafx.com/software/liquigen

Figures

Appendices

Appendix A: Optical Properties Analysis

This chapter evaluates the realism of the fluid's optical properties, reflection, refraction, absorption and shadows, by comparing them to real-world references and other fluid rendering projects. It also compares the results of the two rendering methods used: volumetric ray marching and geometric marching cubes. Each optical effect is isolated using screenshots taken under controlled visual conditions, with the following constant parameters:

Global parameters:

- 20,000 particles
- 4 bounces
- Density map size = 150x150x150px
- Viewport = 1920x1055px

Ray marching parameters:

- Surface detection march size = 0.02m
- Density detection march size = 0.2m

Reflection

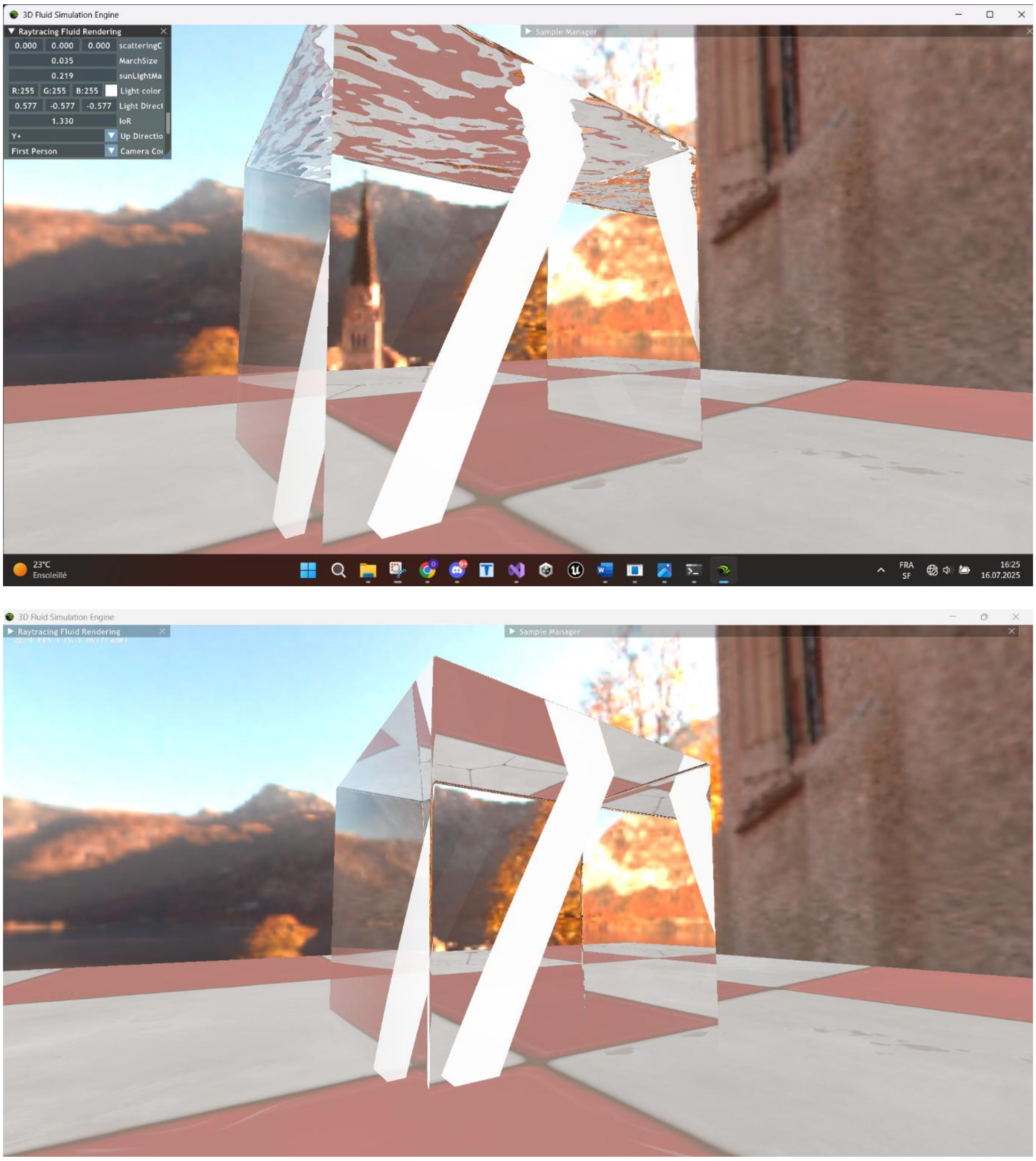
To evaluate the accuracy of the reflection simulation, we first isolate this effect by disabling light absorption (rgb = (0, 0, 0)). This ensures that reflections on the surface and within the fluid volume can be clearly observed without interference from attenuation.

Comparison between the two rendering methods.

Figure 55 shows the reflections generated by both rendering techniques for an identical scene. We can observe the reflection of the checkered floor on the fluid surface, as well as internal reflections (such as the white stick visible through the side of the fluid volume).

In the ray marching version, the surface appears more irregular due to local variations in the density gradient, which introduces slight noise in the surface normals.

In the marching cubes version, the surface is geometrically smooth, producing more stable and uniform reflections.



```
0.000   0.000   0.000   scatteringC
        0.035           MarchSize
        0.219           sunLightMa
R:255   G:255   B:255   Light color
0.577   -0.577  -0.577  Light Direct
        1.330           IoR
Y+                      Up Directio
First Person            Camera Coi
```

Figure 55 Surface and internal reflections in the Media Project. Top: Ray marching. Bottom: Marching Cubes.

Comparison with real-world optics.

To validate the physical plausibility of these reflections, Figure 56 presents a laser beam passing through water and bouncing at the air–water interface. This illustrates total internal reflection, similar to what is observed in the simulation, where light reflects off the inner walls of the fluid volume.

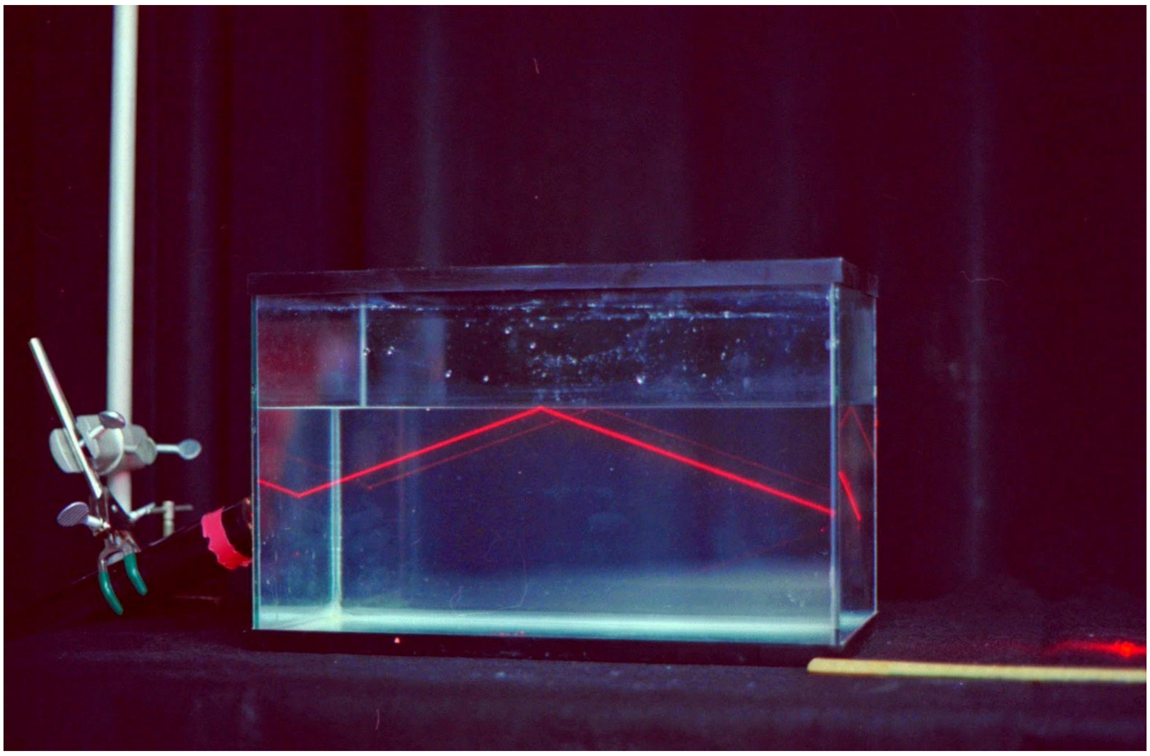


Figure 56 Laser reflection inside water.

Additionally, Figure 57 shows natural reflections observed from underwater. The reflected image of the sky and surroundings is visible at the water's surface, comparable to the reflective patterns produced by the simulation.

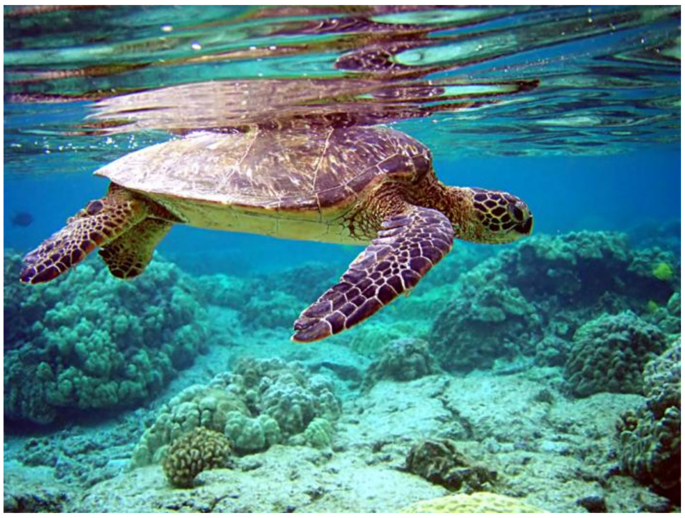


Figure 57 Environmental reflections visible from underwater.

Comparison with another rendering project.

In Figure 58, we compare the internal reflections produced by our simulation with those from FluidX3D (Dr. Moritz Lehmann, 2023) which uses the marching cubes algorithm. In both cases, light reflects inside the fluid volume, particularly along the lower surface and the sidewalls, demonstrating similar optical behaviour.

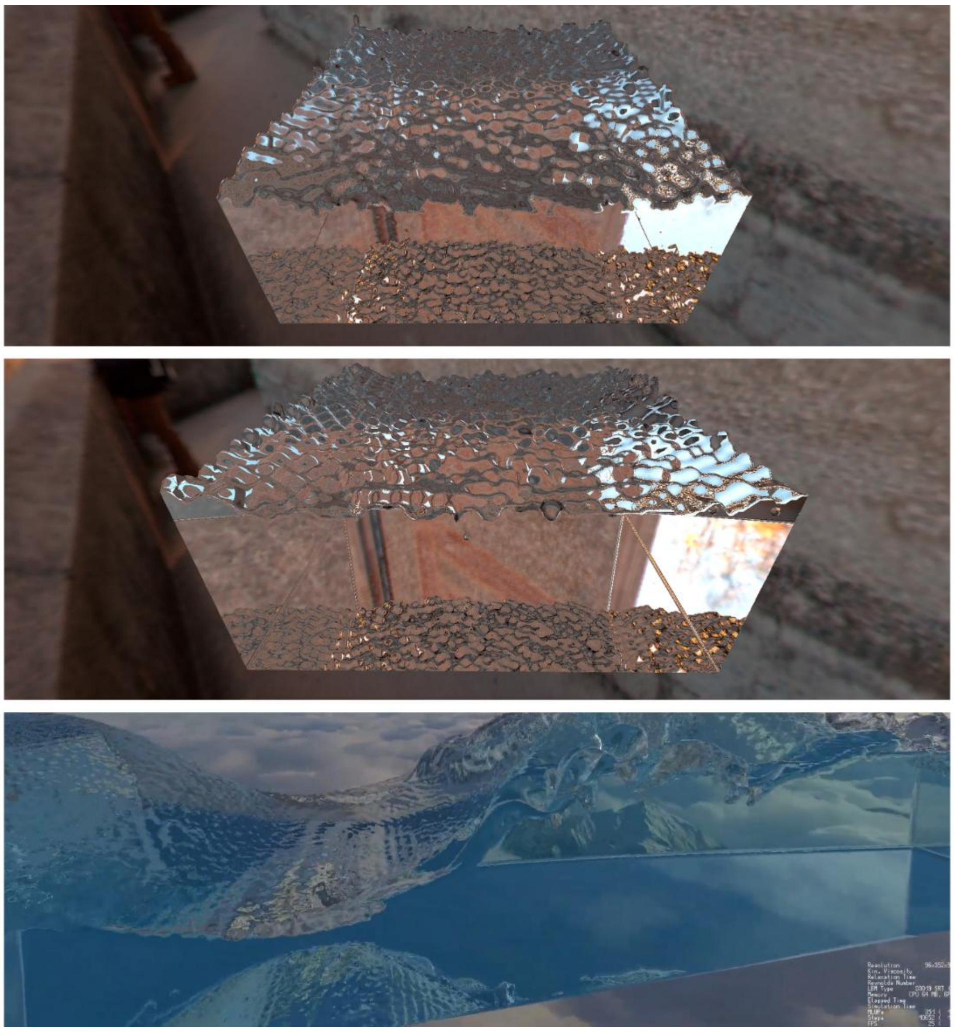


Figure 58 Comparison of internal reflections: Media Project (Top: Ray marching. Center: Marching Cubes) vs. FluidX3D (Marching Cubes).

Summary.

Both rendering methods successfully reproduce key reflection phenomena. Ray marching offers a more dynamic and detailed surface due to volumetric variation, while marching cubes provide a cleaner and more stable appearance. The reflections observed are consistent with physical expectations and match those seen in real fluids and advanced simulation projects.

Refraction

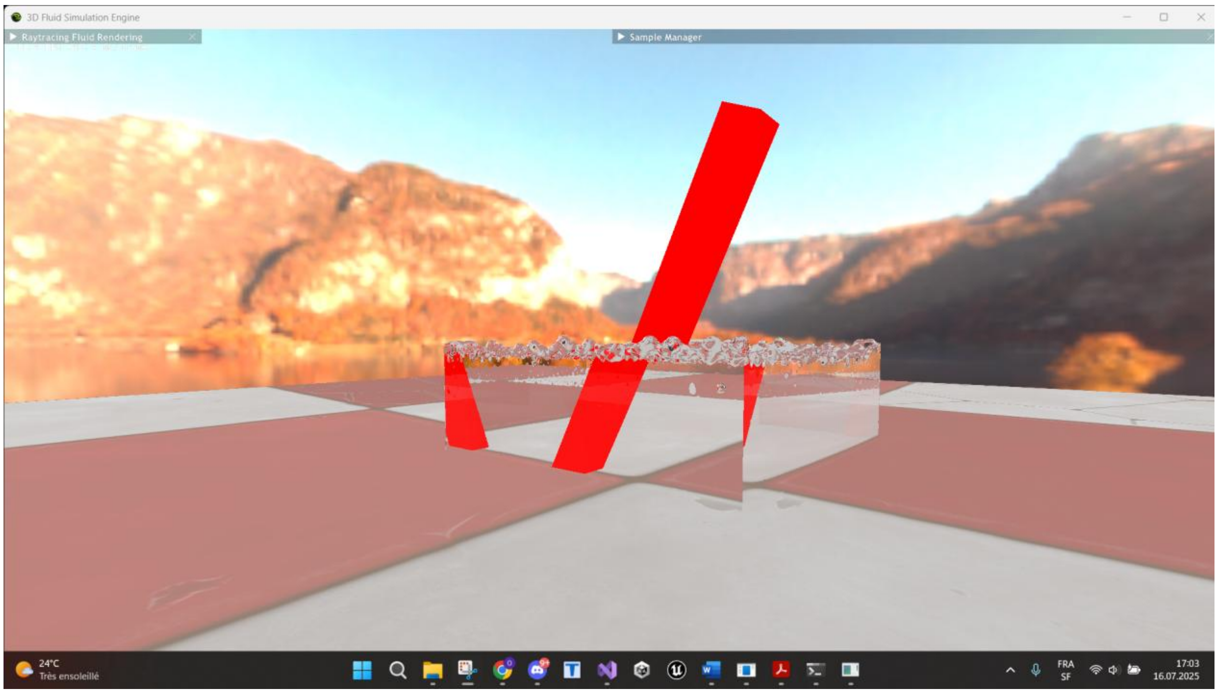
To evaluate the accuracy of refraction through the fluid, the camera was positioned at a low angle to emphasize the visual deformation at the fluid interface. A textured checkered floor and a red rectangular stick were placed partially inside the fluid volume to reveal how light rays bend when passing from one medium to another.

Comparison between the two rendering methods.

Figure 59 shows the same scene rendered using the ray marching (top) and marching cubes (bottom) methods. In both images, we can observe the refraction of the checkered floor behind the fluid and the bending of the red stick at the interface between air and fluid.

In the ray marching version, the volume's density variation results in slightly smoother gradients and subtler distortions.

In the marching cubes version, the geometry provides a cleaner cut at the interface, making the deformation appear sharper.



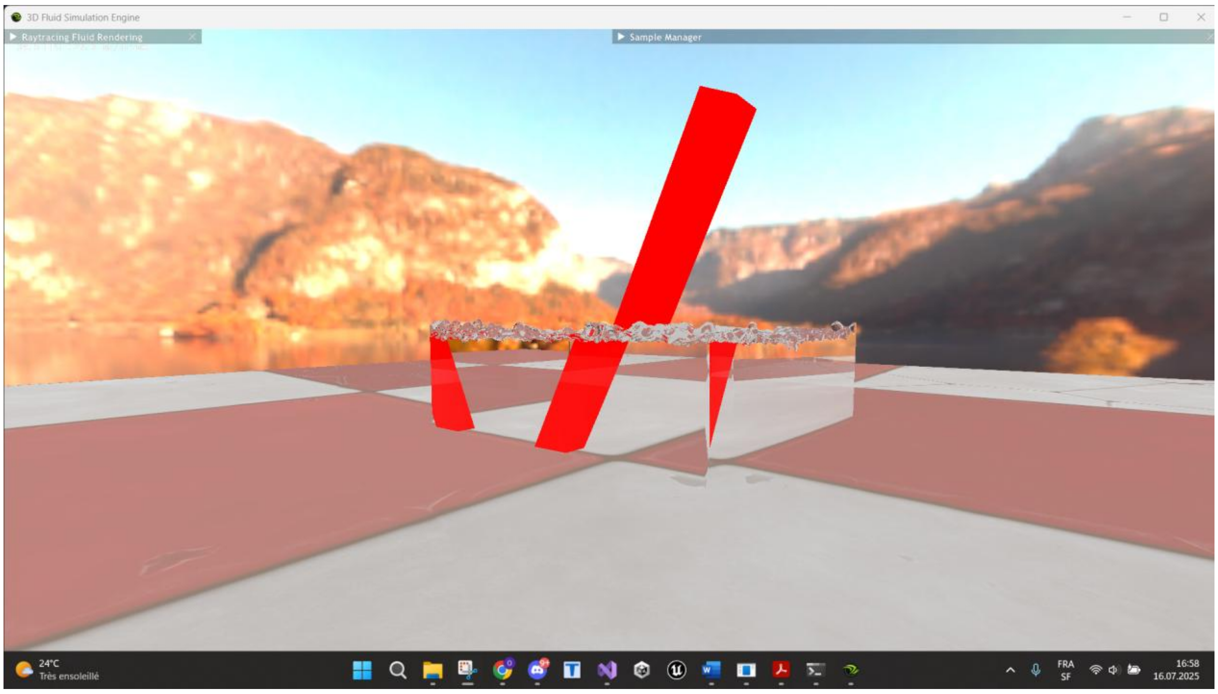


Figure 59 Refraction in the Media Project. Top: Ray marching. Bottom: Marching Cubes.

Comparison with real-world optics.

Figure 60 shows a photograph of a real straw partially submerged in water. The visual deformation is consistent with the renderings: the straw appears "broken" due to the change in the speed of light at the boundary between air and water, a classic illustration of Snell's law.

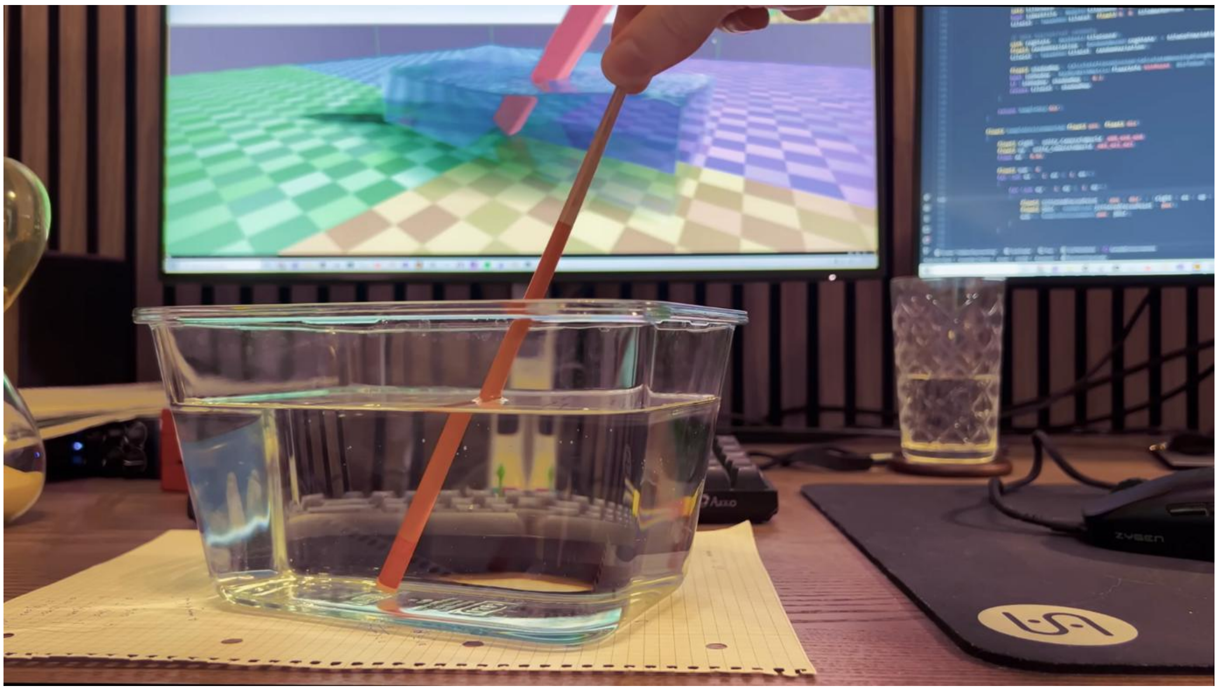


Figure 60 Real-life refraction of a straw in water.

Comparison with another rendering project.

Finally, Figure 61 shows a similar refraction effect produced in Sebastian Lague's Unity-based fluid rendering project. The stick appears displaced and broken at the fluid surface in a way that is visually consistent with both the real photograph and the Media Project renderings.

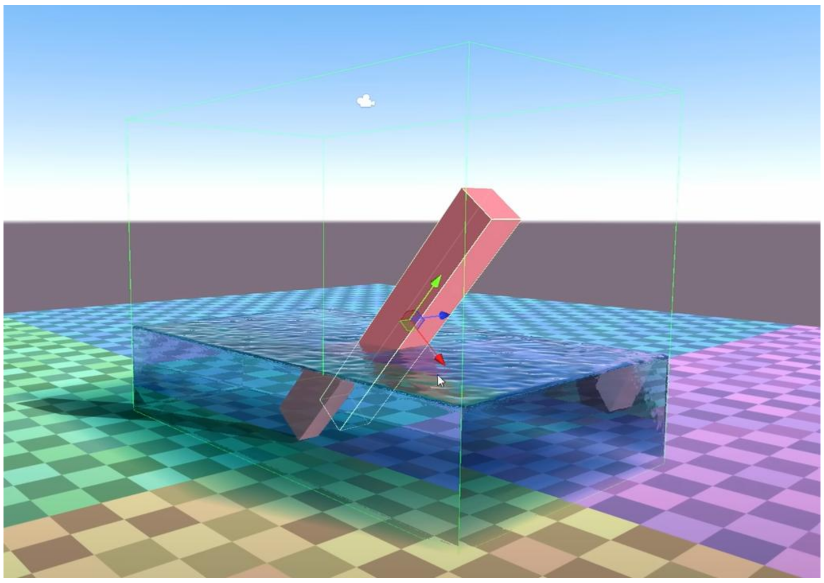


Figure 61 Refraction in Sebastian Lague's Unity project.

Summary

Both rendering methods successfully reproduce the expected optical deformation at the fluid surface. The displacement of the red stick and distortion of the floor pattern align well with real-world observations and external simulation references, confirming the physical plausibility of the implemented refraction model.

Fresnel Effect

The following screenshots were taken with two specific viewing angles: a shallow angle of incidence to emphasize surface reflections, and a near-vertical angle to observe light transmission through the fluid. This setup allows the Fresnel effect to be clearly evaluated.

Comparison between the two rendering methods

Figure 62 presents the Fresnel reflections from both techniques. Both demonstrate a progressive increase in reflection near the fluid surface edge. However, some clear differences emerge:

In the ray marching version, the fluid surface is affected by density noise, resulting in small surface irregularities. These microvariations modulate the Fresnel reflectivity across the image, producing a dynamic and detailed effect. However, this can sometimes result in unstable visual noise.

In the marching cubes version, the smooth surface leads to a more uniform Fresnel reflection, with clean and stable transitions from reflection to transparency. The reflection highlights the fluid's contour more clearly but at the cost of fine surface detail.

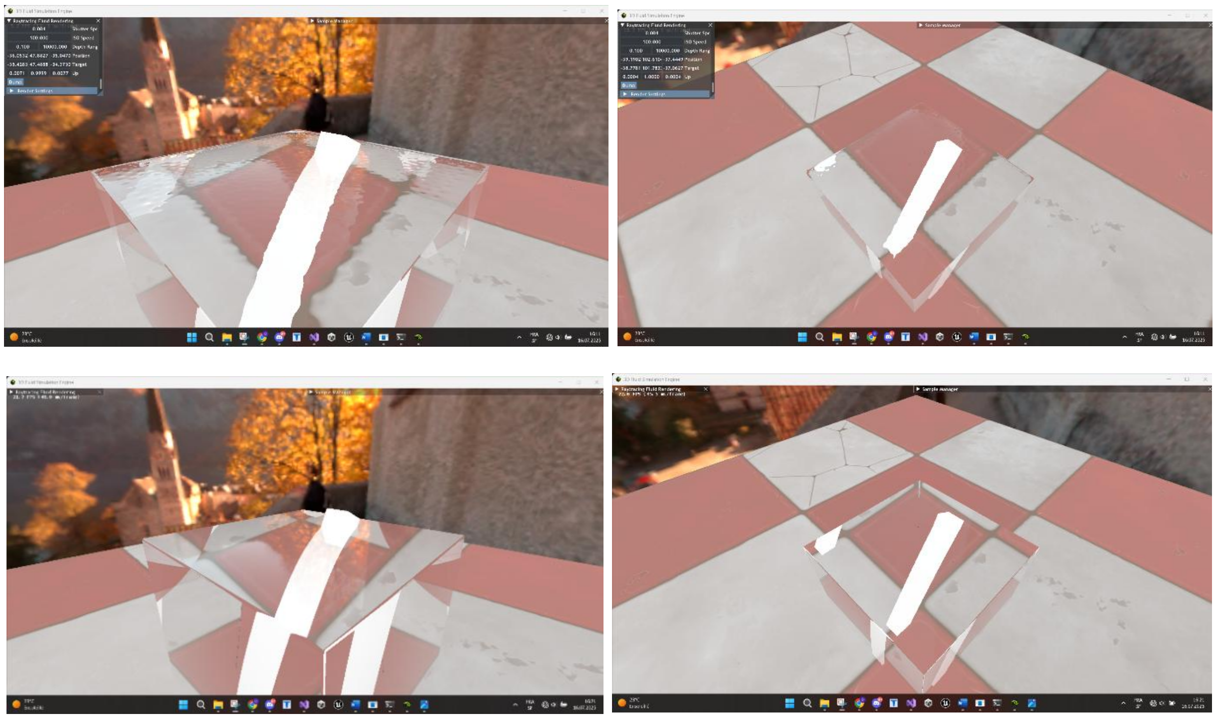


Figure 62 Fresnel reflections in the Media Project. Top: Ray marching. Bottom: Marching Cubes

Comparison with real-world optics

Figure 63 shows a real-life reference image of a water surface viewed at different angles. Just like in the simulated versions, we can observe strong reflections at shallow angles and high transparency near perpendicular views. This visual consistency confirms that the Fresnel response in the Media Project matches real physical behaviour.

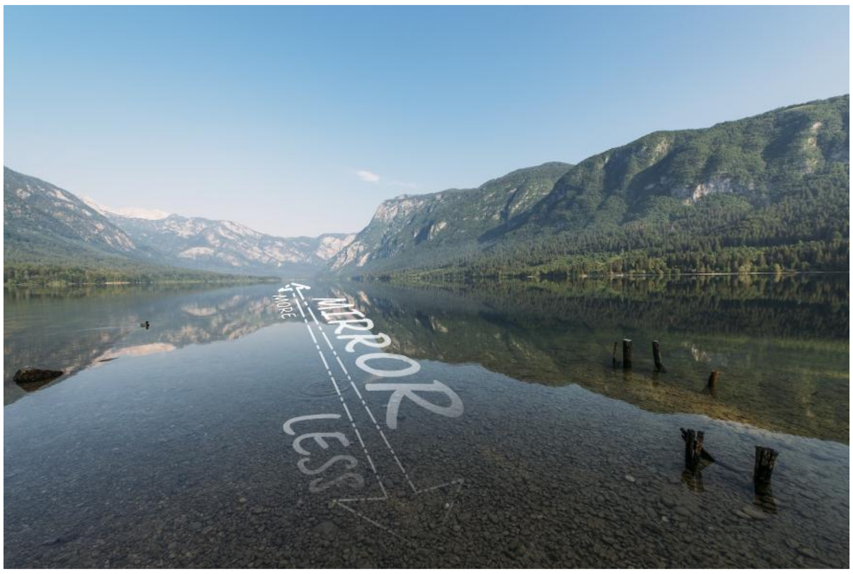


Figure 63 Real-world Fresnel reflection on water.

Comparison with another rendering project

Figure 64 compares the same effect rendered using the WebGL fluid rendering by Evan Wallace (2010). This project also simulates Fresnel reflection based on view angle. The similarity in the angular reflection pattern validates the implementation used in the Media Project. Evan Wallace's rendering also uses a smooth geometry, which produces results visually closer to the marching cubes approach.

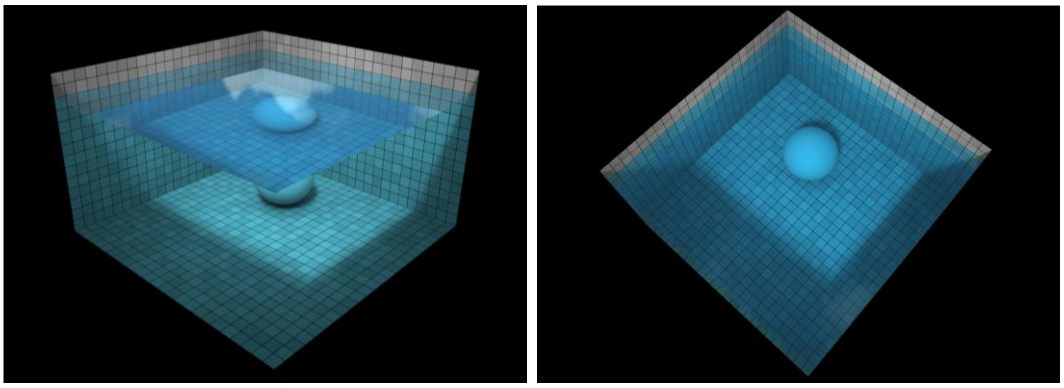


Figure 64 Fresnel reflection in Evan Wallace's WebGL fluid rendering.

Summary

The Fresnel effect is successfully reproduced across both rendering methods. Ray marching emphasizes micro-variations in surface reflection due to volumetric noise, while marching cubes deliver a cleaner and more stable angular transition. Both results align with physical expectations and with external rendering references, demonstrating the visual validity of the implemented Fresnel model.

Absorption

To analyse light absorption, screenshots were taken with the camera facing the fluid, using a white vertical stick to reveal how light intensity decreases through the volume. The absorption coefficient was set to rgb(2.19, 0.75, 0.55) to produce a visible blueish tint.

This effect is not compared to real-world images, as the absorption model used here is a common graphics approximation that ignores microscopic properties such as molecular composition or suspended particles. Instead, comparisons focus on results from other fluid rendering projects.

Comparison between the two rendering methods

Figure 65 shows the results obtained with both the ray marching and marching cubes methods.

In the ray marching version (top), the absorption appears gradual and volumetric. The white stick fades smoothly as it passes through deeper parts of the fluid, and background reflections remain visible, albeit slightly tinted by the fluid.

In the marching cubes version (bottom), absorption is also visible, but an unintended darkening of background reflections is noticeable. The distant wall behind the fluid appears unnaturally dim, especially compared to the ray marching result.

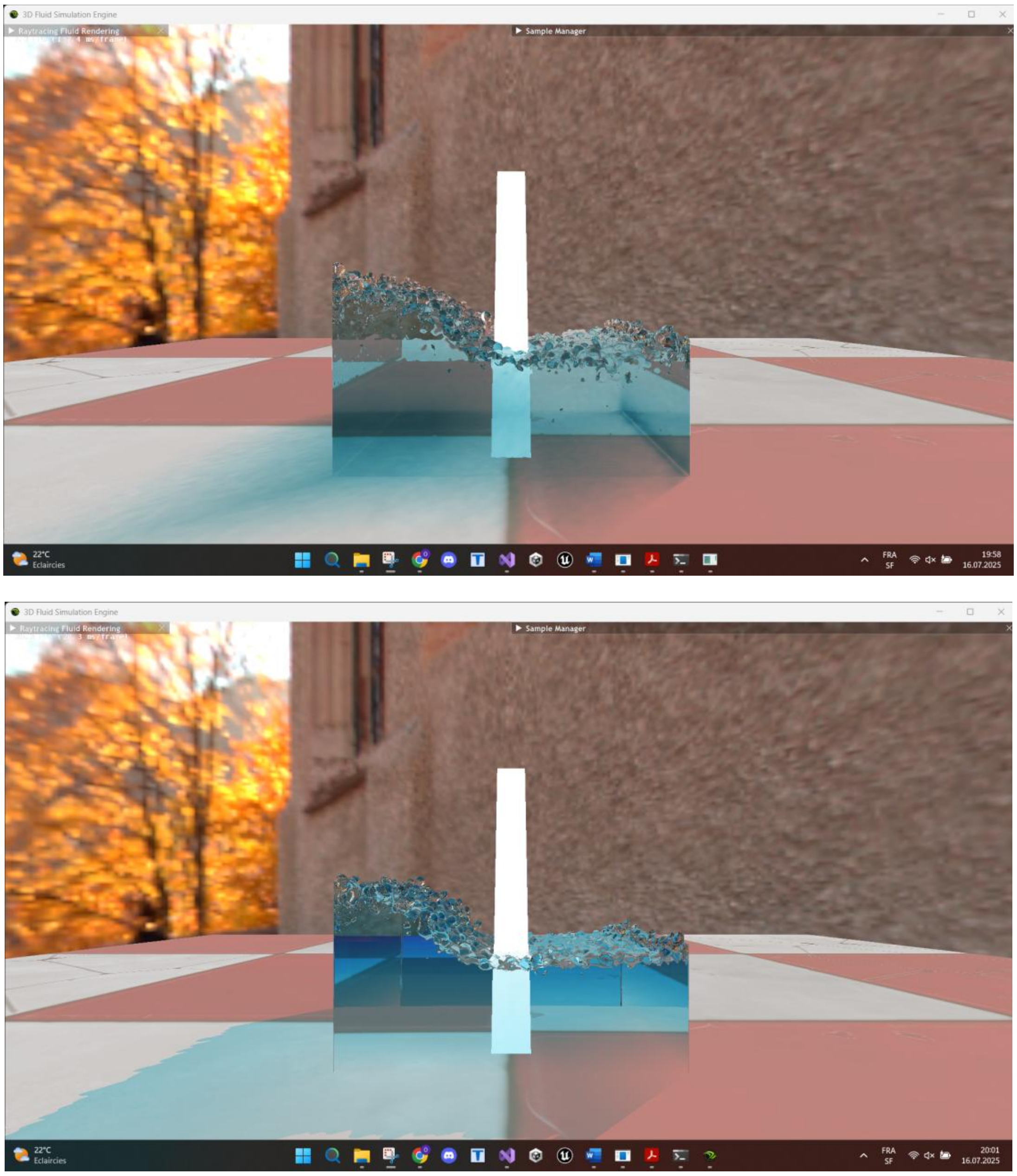

3D Fluid Simulation Engine
Raytracing Fluid Rendering
Sample Manager
22°C
Eclaircies
FRA
SF
19:58
16.07.2025
3D Fluid Simulation Engine
Raytracing Fluid Rendering
Sample Manager
22°C
Eclaircies
FRA
SF
20:01
16.07.2025

Figure 65 Light absorption with coefficient rgb(2.19, 0.75, 0.55) in the Media Project.

Comparison with other rendering projects.

Figure 66 shows the result from Sebastian Lague's Unity-based fluid simulation, which also uses a ray marching approach. The white light fades as it passes through the fluid, and the background reflections remain properly lit. The behaviour is visually consistent with the Media Project's ray marching version.

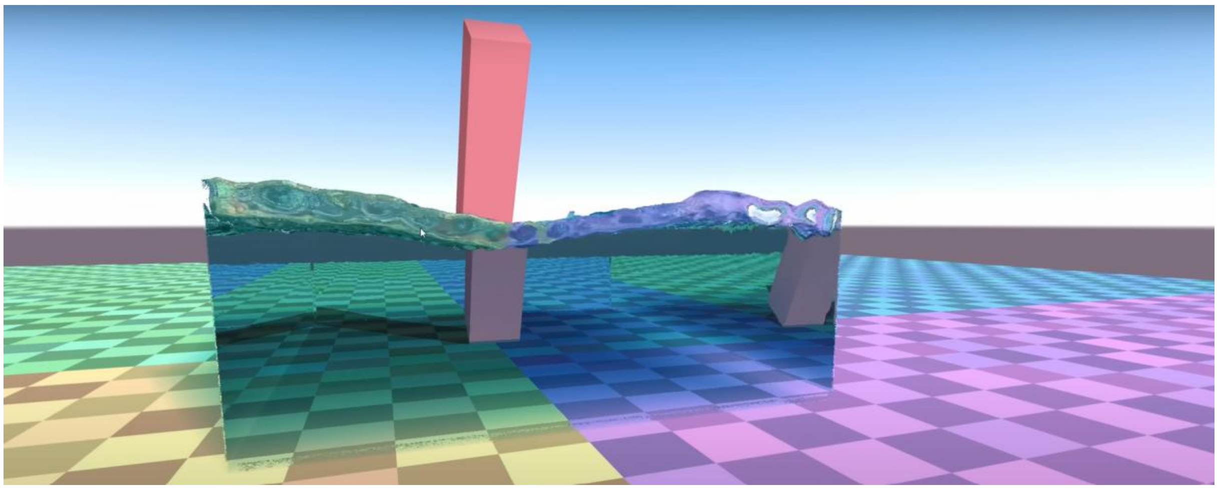


Figure 66 Light absorption in Sebastian Lague's Unity fluid renderer.

Figure 67 presents absorption in FluidX3D, which uses marching cubes. As in Lague's project, reflections remain bright, regardless of the fluid depth. This highlights the issue in the Media Project's marching cubes rendering, where the reflected environment appears too dim, a behaviour not observed in either external reference.

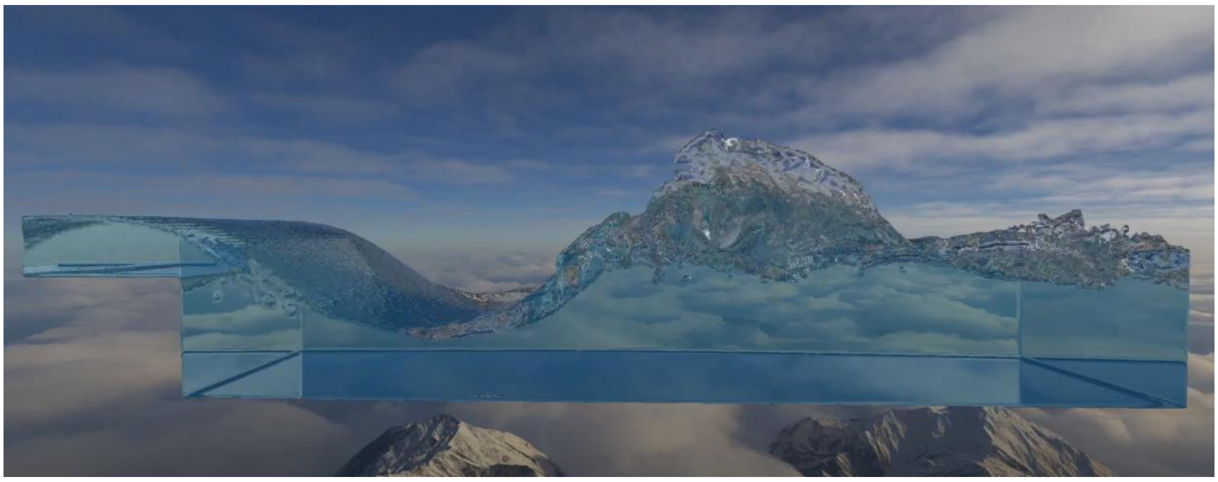


Figure 67 Light absorption in FluidX3D.

Summary

Both rendering methods simulate volumetric light attenuation, with ray marching producing smooth and physically consistent fading, visually close to that of Sebastian Lague's renderer. However, the marching cubes version displays unnaturally dark reflections, which are absent from both FluidX3D and Lague's project. This discrepancy is most likely caused by an implementation error, possibly the incorrect application of absorption to reflected rays.

Shadows

To analyse shadow behaviour, top-down screenshots were taken to observe volumetric light attenuation as sunlight passes through the fluid and reaches the floor. As in the light absorption analysis, no real-world photographs are used for comparison, since the current absorption implementation does not fully reflect the physical principles governing real fluid behaviour. Instead, we focus on comparing the Media Project with other graphics-based simulations.

Comparison between the two rendering methods

Figure 68 shows the shadows cast through the fluid in both ray marching (top) and marching cubes (bottom).

In the ray marching version, the shadow is soft and volumetric. Light is gradually absorbed as it travels through the fluid, producing smooth gradients on the ground. The shadow shape reflects the internal structure and density of the fluid.

In the marching cubes version, the shadow is far more uniform and unrealistic. Because the method does not sample density continuously along the light ray, the result appears as a flat projection with sharp edges and little internal variation. This lacks the depth and softness expected from volumetric shadows.

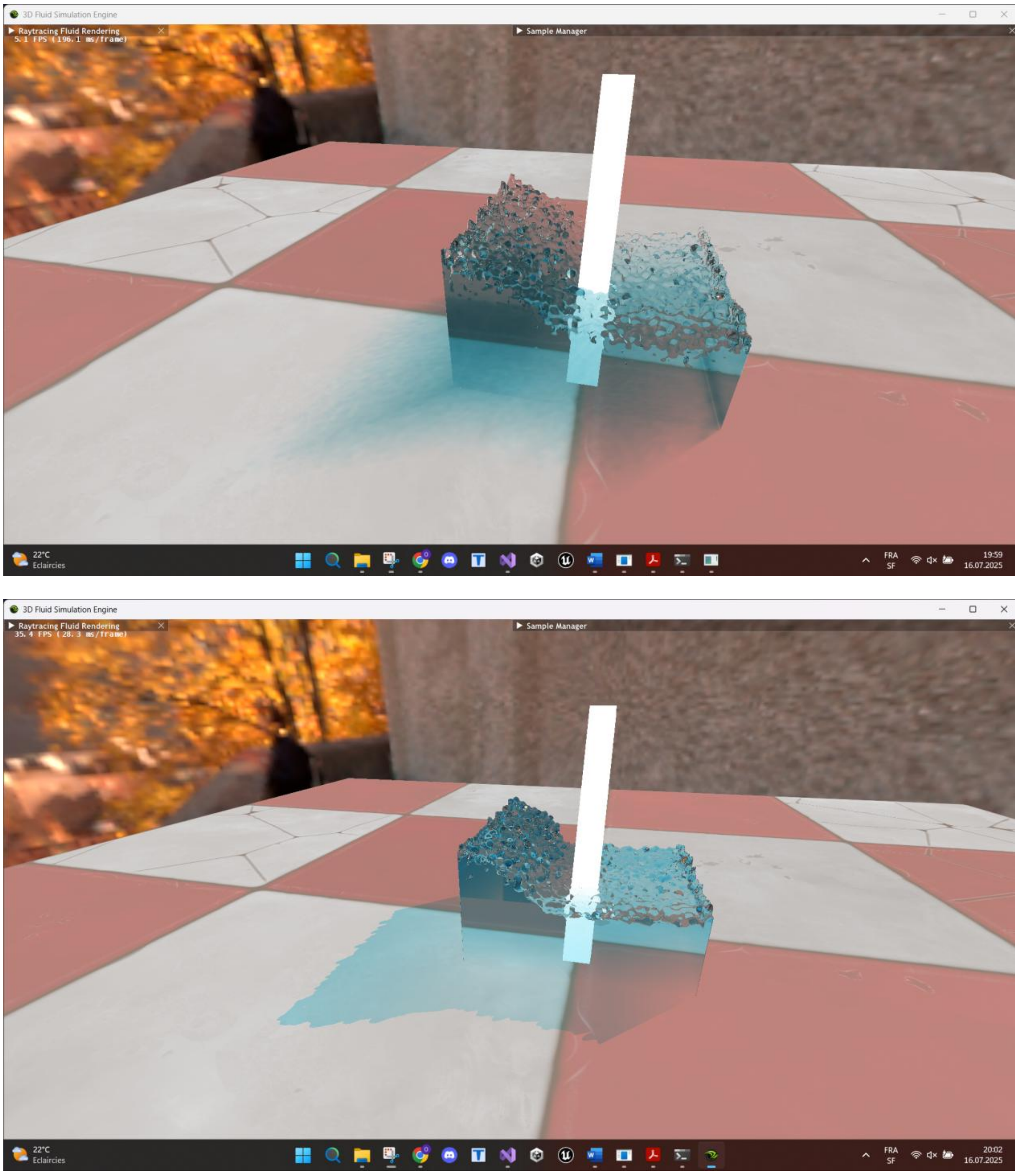


Figure 68 Shadows in the Media Project. Top: Ray marching. Bottom Marching Cubes.

Comparison with other rendering projects.

Figure 69 shows the shadow result from Sebastian Lague's Unity fluid simulation, which also uses a ray marching-based technique. The resemblance to the Media Project's ray marching version is striking: both exhibit smooth, depth-aware volumetric shadows with natural falloff and light scattering.

In contrast, no direct comparison is made with external marching cubes projects, either because shadows are missing, or because they are computed with fully realistic lighting models (e.g., full global illumination), making them too different to compare meaningfully. Nevertheless, when compared to Sebastian Lague's raymarched result, it is clear that the Media Project's marching cubes implementation fails to produce convincing volumetric shadows.

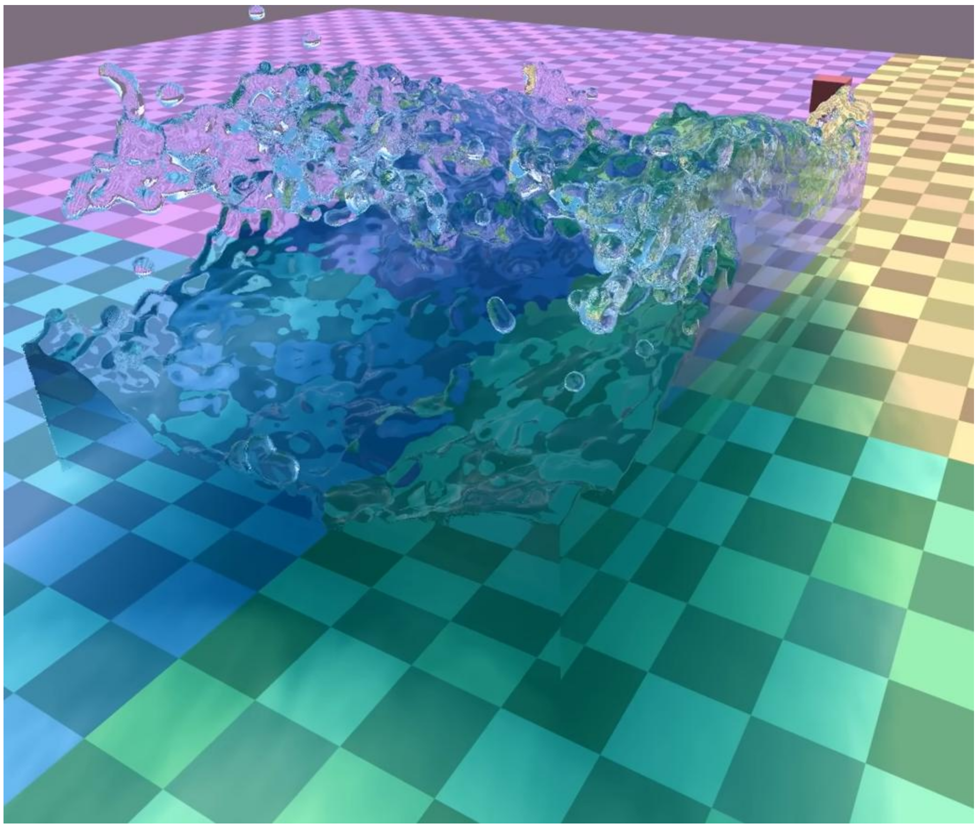


Figure 69 Volumetric shadow in Sebastian Lague's Unity fluid renderer.

Summary

The ray marching method accurately simulates volumetric shadowing, producing soft and realistic attenuation on the ground. In contrast, the marching cubes method fails to reproduce this effect, as it does not integrate density along the light path. This leads to overly uniform and non-physical shadow shapes. As a result, only the ray marching implementation delivers shadows that visually reflect the fluid's volume.