3D Visualisation of Molten Glass using Raymarching Technique

Félix Chevalier
Manon Méhalin
Maxence Retier

Game Programming
Option GPU

September 2024 - June 2025

Tutor ISART DIGITAL:
Maël ADDOUM
m.addoum@isartdigital.com

Tutor enterprise:
Sylvain
CONTASSOT-VIVIER
sylvain.contassotvivier@loria.fr

Abstract

This project explores a real-time simulation of molten glass using raymarching techniques applied to implicit surfaces and 3D point clouds. A custom rendering pipeline was implemented in Vulkan, leveraging compute shaders for efficient parallel execution and enhanced control over memory access. Blending operations, Phong-based lighting, and reflection models were integrated to visually reproduce the semi-fluid, reflective nature of molten glass. A bespoke space partitioning algorithm was also developed to optimise point cloud traversal.
Preliminary results confirm the feasibility and visual realism of the approach, establishing a solid foundation for future integration of physical simulation and user interaction.
Beyond its technical contributions, this work illustrates how immersive technologies can support the transmission of traditional craftsmanship while promoting more sustainable practices.

Table of contents

1 Introduction **3**

2 The methodology **4**
2.1 Technical Scope and Approach 4
2.2 Project Breakdown 5
2.3 Space partitioning 11
2.4 Development Environment 14
2.5 Key Technical Choices 14
2.6 Performance Evaluation Method 15

3 Results and discussion **16**
3.1 Achieved results 16
3.2 Performance and Realism Evaluation 18
3.3 Limitations and Ongoing Work 19
3.4 General Discussion 20

4 Conclusions, future prospects **21**

5 Bibliographical references **22**

1 Introduction

Glassblowing is an art form with deep cultural and historical significance, particularly in France, where it embodies centuries of craftsmanship and tradition. However, preserving this heritage presents modern challenges, including the environmental impact of energy-intensive training methods and the need to adapt artisanal practices to digital technologies.

This work is conducted within a broader research initiative led by the LORIA (Laboratoire Lorrain de Recherche Informatique et ses Applications), which aims to develop an immersive augmented reality experience for virtual glassblowing. The present contribution focuses specifically on the real-time visual rendering of molten glass. No physical modelling of the material is performed ; the project is limited to graphical representation, to produce a visually convincing depiction of molten glass in an interactive context.

Real-time visualisation of molten glass remains a complex task due to its semi-fluid behaviour and optical properties. Achieving a balance between realism and performance is essential to enable integration into interactive environments.
To this end, a simulation framework based on raymarching techniques applied to 3D point clouds has been developed. Raymarching, an algorithm that advances rays incrementally to detect surfaces, offers a flexible approach to visualising deformable materials. In this context, it enables the dynamic visualisation of molten glass surfaces without relying on explicit geometry.

This method departs from traditional mesh-based rendering. Instead, implicit surfaces and raymarching are employed for their suitability with point cloud data, allowing dynamic deformation to be integrated with advanced visual effects. Although molten glass exhibits both fluid and solid characteristics depending on temperature and motion, these physical behaviours are not simulated. Rather, the rendering method approximates the visual result in real-time.

Beyond its technical scope, the project also addresses sustainability. By offering a virtual training tool, reliance on physical furnaces may be reduced, thereby lowering the environmental footprint associated with traditional training. This dual objective, technological innovation and ecological responsibility, forms the foundation of the work presented in the following sections.

2 The methodology

To address the challenge of representing molten glass in real-time, a structured and experimental methodology was adopted, drawing from graphics programming and real-time rendering techniques. This project does not aim to reproduce the physical behaviour of molten glass but focuses exclusively on its visual representation. The objective is to develop a convincing and responsive graphical system suitable for immersive environments. This required a balance between artistic fidelity, technical feasibility, and performance constraints.

2.1 Technical Scope and Approach

The visual rendering of molten glass presents several unique challenges : it must appear fluid, semi-transparent, and dynamically deformable. Traditional mesh-based rendering methods are generally inadequate for representing such materials, particularly when geometry evolves continuously. To address these limitations, a **raymarching-based rendering technique** was selected, enabling the direct rendering of **implicit surfaces** without predefined geometry.

Raymarching is a rendering approach in which rays are cast through a scene, and the algorithm incrementally advances along each ray until a surface is detected (see Fig.1). This technique is particularly effective for rendering mathematically defined surfaces such as metaballs, soft shape blends, or deformable volumes, features that are well suited for conveying the visual characteristics of molten glass.

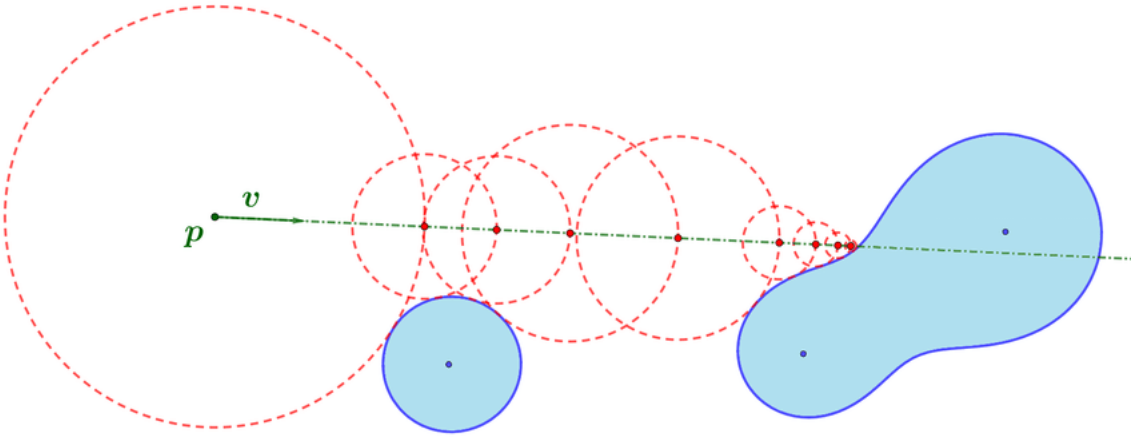

FIGURE 1 – Raymarching system

2.2 Project Breakdown

The technical development was structured into several incremental phases :

1. **Initial Raymarching Setup**

   A minimal raymarching loop was first implemented to render a basic torus shape using Signed Distance Functions (SDFs).

   A SDF is a mathematical function that, given a point in space, returns the shortest distance from that point to the surface of an object. The sign of the distance indicates whether the point is inside (negative value), on the surface (zero), or outside (positive value) of the object.
   Mathematically, for a point p and a surface S, an SDF f(p) satisfies :

   - f(p) $<$ 0 if p is inside the surface,
   - f(p) $=$ 0 if p lies exactly on the surface,
   - f(p) $>$ 0 if p is outside the surface.

   This concept is particularly powerful in the context of raymarching, where rendering relies on marching a ray through the scene and stopping when it is sufficiently close to a surface (i.e., when the SDF returns a value close to zero). The value returned by the SDF at each step tells the algorithm how far it can safely advance without hitting any geometry. This step enabled validation of the rendering pipeline and early experimentation with implicit surface definitions. See the mathematical function here to render a torus (see Fig.2) and a sphere.

```
float torusSDF(vec3 p, vec2 t)
{
vec2 q = vec2(length(p.xz) - t.x, p.y);
return length(q) - t.y;
}
```

```
float sphereSDF(vec3 p, vec3 center, float radius)
{
return length(p - center) - radius;
}
```

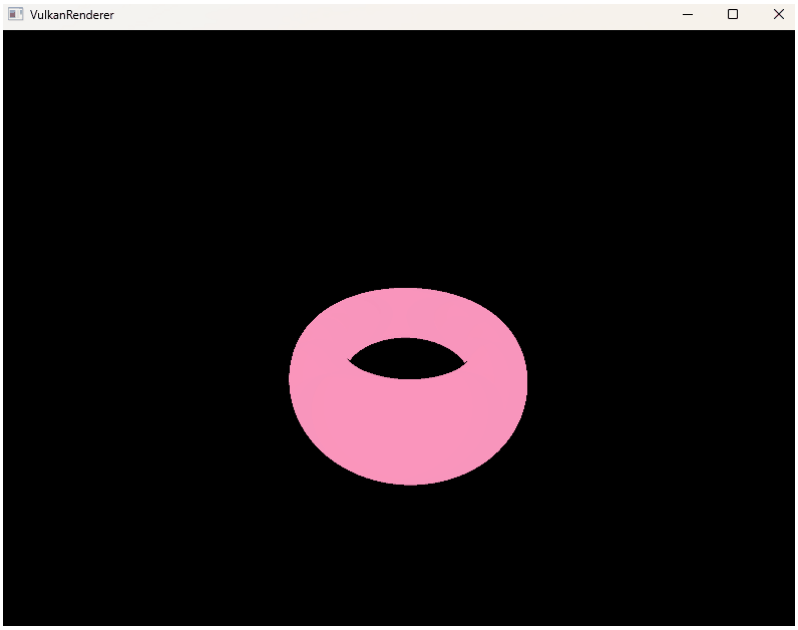

FIGURE 2 – Raymarching torus

2. **Implementation of Shape Blending**

   One of the main advantages of SDFs is that they allow the representation of implicit surfaces, surfaces defined not by vertices or meshes, but by equations. This makes them ideal for modelling smooth, continuous shapes like spheres, tori, or more complex forms created through blending (smooth unions), deformation, or procedural transformations.

   To simulate the smooth merging of surfaces, as we observe in real molten or soft materials, we use a smooth minimum function. One commonly used variant is the polynomial smooth min, defined as :

```
float smoothMin(float a, float b, float k)
{
    float h = max(k - abs(a - b), 0.0) / k;
    return min(a, b) - h * h * h * k * (1.0 / 6.0);
}
```

   - **a** and **b** are the distance values from the two SDFs.
   - **k** is the blending factor : the higher the value, the softer the blend.

This function smoothly interpolates between the two distance fields when they are close, producing a rounded transition rather than a hard intersection. Visually, this results in a fluid, organic fusion of shapes — ideal for simulating viscous materials such as glass in a molten state. (See Fig.3)

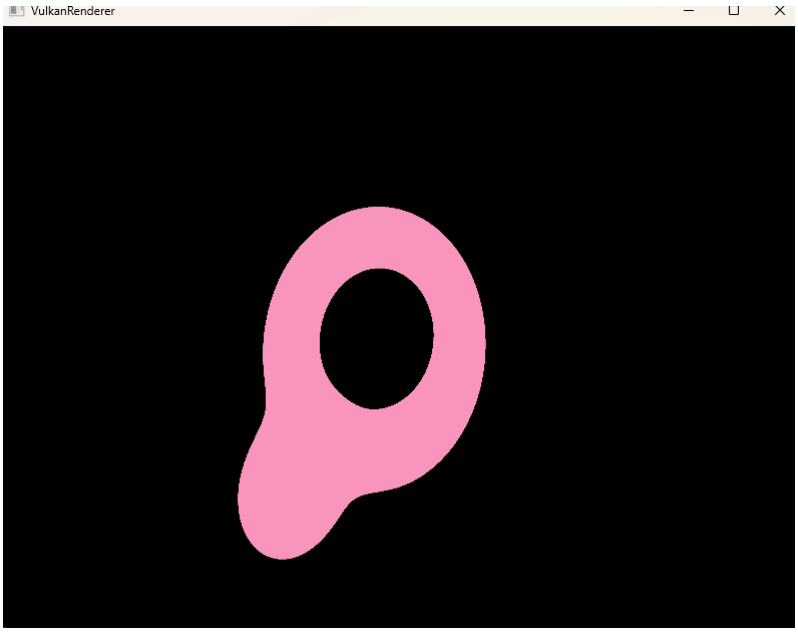

FIGURE 3 – Raymarching blending

3. **Lighting and Reflection Enhancements**

   Realistic shading is essential for depicting transparent and reflective materials such as glass. Phong-based lighting, Fresnel effects, and surface normals derived from SDF gradients were incorporated to enhance visual realism (see Fig.4).

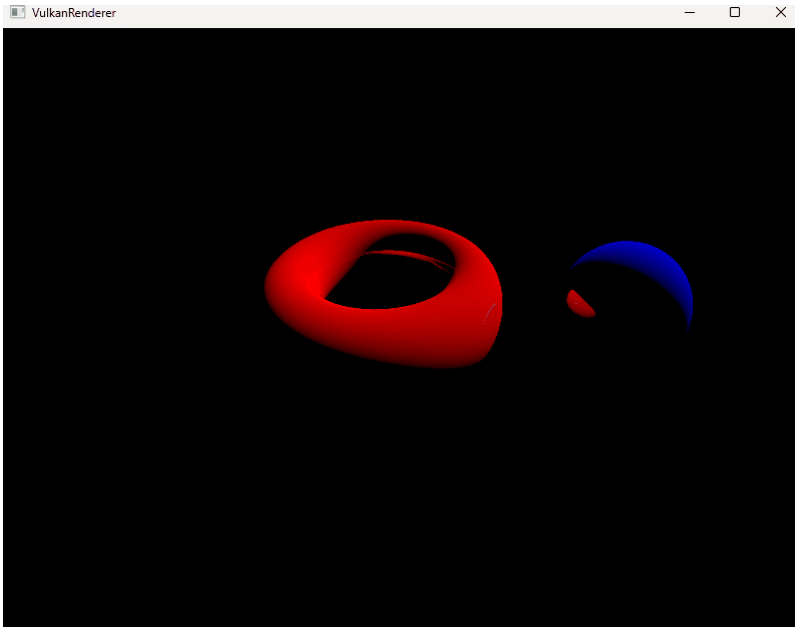

FIGURE 4 – Raymarching Phong light

4. **Transition to Compute Shaders**

   The initial implementation relied on fragment shaders, but performance and flexibility limitations led to a migration to compute shaders. This transition provided enhanced control over parallel execution and memory access, enabling better performance and supporting future extensions involving point cloud data.

5. **Upgrading raymarching rendering**

   To enhance the realism of the molten glass material, we implemented a transparency model within our raymarching shader. This feature allows the simulation of light transmission through semi-transparent materials, which is essential for mimicking the visual behaviour of real glass.

   Transparency is controlled through a Material structure that contains parameters such as reflectivity, transparency, and the index of refraction. During the raymarching process, when a ray intersects an object, the material properties are retrieved using the sceneSDF function. While current examples use opaque spheres, the framework is fully capable of handling transparent objects.

   Additionally, light attenuation is applied at each bounce to simulate absorption, which adds visual depth and realism. The final colour of the pixel results from a weighted combination of direct lighting, reflected light, and refracted light, depending on the material's optical properties.

   When a ray hits a surface, it can reflect depending on the surface's reflectivity. The reflected direction is calculated using the standard reflection formula :

```
vec3 reflectDir = reflect(ray.direction, normal);
```

   If the material's reflectivity is high, the ray is bounced in the reflected direction and continues its march, contributing to mirror-like highlights and surface detail.

6. **Transparency and Refraction**

   Transparency is controlled by a coefficient stored in the material. When transparency is non-zero, the shader considers refraction, which simulates light bending as it passes through the surface, according to Snell's Law. The refracted direction is computed with :

```
vec3 refractDir = refract(ray.direction, normal, eta);
```

   Where eta is the ratio of indices of refraction (IOR) between the two media, the IOR defines how much the light bends ; typical values range from 1.0 (air) to 1.5 (glass).

   If a valid refracted ray exists and the material is more transparent than reflective, the ray continues inside the object. Otherwise, it reflects. This choice is determined dynamically at each bounce :

```
bool useRefract = transparency > reflectivity
                && length(refractDir) > 0.001;
```

7. **Light Attenuation and Final Color**

   To simulate absorption of light in transparent materials, an attenuation factor is applied after each bounce. This causes light to gradually lose intensity and adopt the material's colour, mimicking how real glass absorbs and tints light :

```
attenuation *= mix(vec3(1.0), material.color, 0.1);
```

   The final colour of the pixel is computed by combining :
   - Direct light (diffuse),
   - Reflected light (from mirrored surfaces),
   - Refracted light (transmitted through transparent areas),

   all weighted by the material's properties.

FIGURE 5 – Reflection and refraction

2.3 Space partitioning

Distance evaluations for multiple spheres were identified as a computational bottleneck in the raymarching process. To improve performance when processing large point clouds, a custom space partitioning algorithm was developed, prioritising balanced subdivision and traversal efficiency over conventional methods such as k-d trees [1].

The algorithm begins by computing an Axis-Aligned Bounding Box (AABB) around the point cloud, followed by recursive axis-aligned subdivisions along X, Y, and Z axes based on median point positions. This ensures an even distribution of points across subdivisions and minimises the depth of the resulting binary tree (see Fig.6).

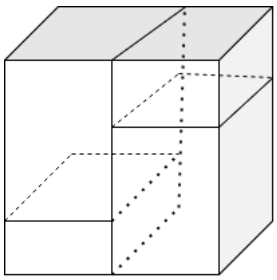

FIGURE 6 – Space representation of the point cloud bounding boxes after 2 cuts

1. **Binary tree**

   Each subdivision is represented by a node in a binary tree. The root corresponds to the bounding volume of the entire point cloud, while child nodes contain recursively defined subspaces (see Fig.7).

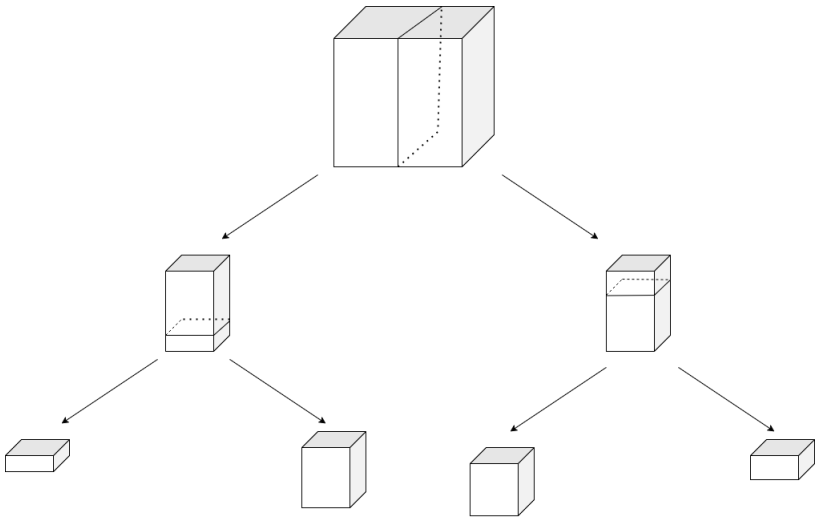

FIGURE 7 – Binary tree

Unique indices are assigned to each node using the Morton order. The root node is indexed as 1, and each child inherits an index extended by 0 or 1 depending on its position relative to the parent. This allows the binary tree to be stored in a linear buffer, which is then sent to the compute shader.

2. **Tree exploration in compute shader**

   The common method for exploring all nodes in a tree involves recursive functions. However, due to limitations of GPU architectures, recursion is not supported. Tree traversal is therefore implemented using a stack, represented by an array of indices. The process begins with the root node, and a while loop is used to perform computations. During each iteration, child nodes are added to the stack only if they are relevant for future checks. To reduce unnecessary computations, nodes whose bounding boxes are not intersected by the current ray are excluded from traversal.

   Visual effects, such as blending, are also adapted to the tree structure. Applying the smoothMin function to all spheres in the scene would be prohibitively inefficient. Instead, the binary tree allows each leaf node to contain a limited number of nearby points. Blending is thus restricted to the spheres contained within the same leaf node. Although this approach may introduce minor approximations, the visual results remain consistent, as nearby points are generally grouped within the same sub-region.

The lighting and reflection systems operate identically with or without tree-based traversal. The primary requirement is to correctly identify the nearest sphere in order to retrieve its associated data. Once the nearest geometry is determined, surface normals are computed and standard shading operations are applied. The binary tree structure contributes to the optimisation of lighting calculations, particularly for Phong-based effects, by reducing the number of distance evaluations needed.

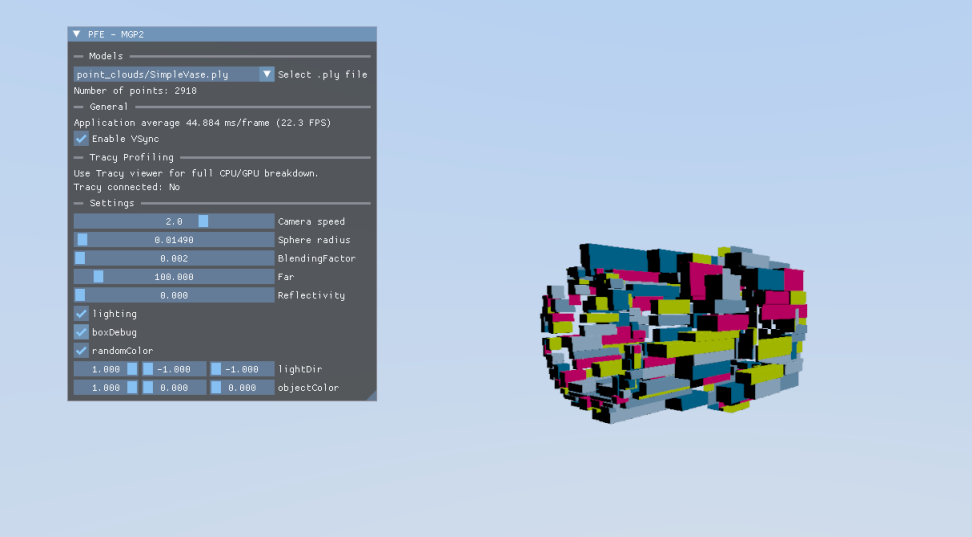

FIGURE 8 – Smallest sub-boxes showed for debugging purposes

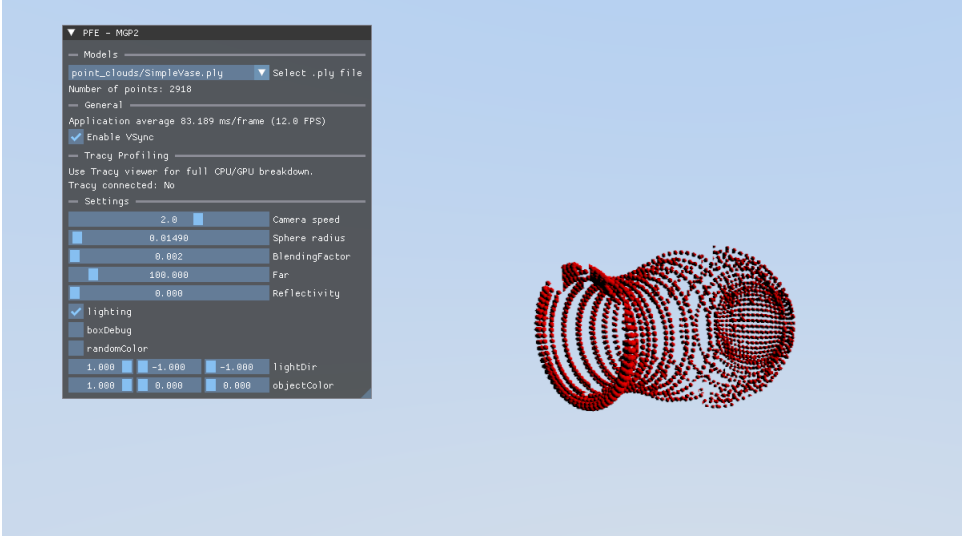

FIGURE 9 – The same view rendered with lighting and blending effects

2.4 Development Environment

The system has been implemented using Vulkan, a low-level graphics API offering fine-grained control over GPU resources and pipeline stages. Vulkan was preferred over OpenGL for its performance, multithreading support, and native compatibility with compute shaders.

The rendering logic is written in C++, while GPU operations, such as raymarching, lighting, and memory management, are implemented in GLSL, with dedicated modules for fragment and compute shaders. Version control was managed through Git, and the codebase was structured into modular components for maintainability and scalability.

2.5 Key Technical Choices

Several key decisions shaped the development approach :

- Adoption of raymarching : Enabled the rendering of smooth, deformable shapes without relying on mesh geometry.

- Use of implicit surfaces and blending techniques : Provided a way to visually represent molten glass in a fluid-like state through continuous transitions between forms.

- Migration to compute shaders : Offered improved performance and flexibility by decoupling rendering from the rasterisation pipeline.

- Preparation for volumetric and point-based rendering : Established a foundation for future integration of 3D data structures, while remaining within the scope of real-time visualisation, not physical simulation.

2.6 Performance Evaluation Method

To assess the performance of the rendering system, a dedicated profiling methodology was established using the Tracy profiler. This tool enabled detailed monitoring of execution timings, memory usage, and GPU/CPU workload distribution, providing valuable insights into the internal behaviour of the application.

1. **Comparison between fragment shader and compute shader pipelines**
   Performance was measured for both a traditional rasterisation-based pipeline using fragment shaders and a compute-shader-based rendering pipeline. This comparison aimed to evaluate differences in control, parallelisation efficiency, and resource usage.

2. **Impact of the binary tree structure**
   Tests were conducted with and without the custom space partitioning algorithm based on a binary tree to determine its influence on distance computations and traversal efficiency within point cloud datasets.

3. **Metric collection**
   The profiling focused on collecting quantitative data such as frames per second (FPS), GPU utilisation, and CPU utilisation. To ensure consistency across tests, the same scene and parameters were used in all configurations, isolating the impact of the rendering pipeline itself.

4. **Multi-system testing**
   To ensure generalizability and better understand performance variations across hardware, tests were conducted on multiple machines with different specifications. This approach provided a broader basis for comparison and highlighted the scalability of the system.

It should be noted that the use of ImGui for interface and debugging purposes introduces a dependency on the traditional graphics pipeline. As a consequence, the rendering system does not operate in a fully compute-only configuration. This factor was taken into account during performance measurements and may influence the interpretation of the results.

3 Results and discussion

3.1 Achieved results

At the current stage of the project, a fully functional raymarching rendering loop has been implemented, enabling real-time visualisation of implicit surfaces. Basic geometric shapes, such as spheres, are rendered using Signed Distance Functions (SDFs), and blending techniques have been developed to produce smooth transitions between them, effectively mimicking the visual behaviour of molten glass.

In addition to shape blending, dynamic lighting and reflection models have been integrated. Surface normals are computed from the gradient of the SDFs, and lighting effects, such as specular highlights and environmental reflections, are applied to enhance the appearance of a transparent, reflective material. These enhancements contribute to a visually convincing representation of molten glass.

A significant milestone was the transition from a traditional fragment shader pipeline to compute shaders. This change provided improved control over execution, memory management, and thread distribution, resulting in better performance and greater scalability for future extensions. A custom binary tree structure was also implemented to optimise point cloud traversal, laying the foundation for more efficient distance computations in the compute shader pipeline.

A comparison between the fragment and compute shader approaches is presented in Table 1. Results show a clear gain in rendering performance when using the compute pipeline, especially as the number of rendered primitives increases.

Shader	Nb spheres	GPU Draw	GPU delay to exec	CPU setup	CPU DrawFrame
Fragment shader	1000	56,74 ms (~17 FPS)	13,59 ms	32,56 µs	142,17 µs (~0,85 %)
	1500	375,87 ms (~3 FPS)	8,44 ms	26,93 µs	177,41 µs (~1,06 %)
Compute shader	2000	54,64 ms (~18 FPS)	5,04 ms	14,13 µs	145,97 µs (~0,88 %)
	3000	238,09 ms (~4 FPS)	5,87 ms	28,37 µs	144,51 µs (~0,87 %)

TABLE 1 – Comparison between fragment shader and compute shader

GPU Draw : time spent rendering on the GPU.
GPU delay to exec : latency between command submission and actual GPU execution.
CPU setup : CPU time required to prepare the draw calls.
CPU DrawFrame : total CPU time per frame, with the approximate cost in percentage of the frame duration.

Following this baseline comparison, additional tests were conducted to assess the effect of lighting and to evaluate the compute shader performance after integrating the binary tree. These tests were performed using 2000 and 3000 spheres, rendered both with and without lighting. As shown in Table 2, the values for GPU delay, CPU setup, and CPU DrawFrame remain stable across configurations, confirming that the addition of the binary tree introduces no measurable overhead.

In contrast, the GPU Draw time increases significantly when lighting is enabled. This effect is particularly notable with higher object counts, where the GPU render time rises sharply while CPU metrics remain largely unchanged. The results confirm that the raymarching stage, especially when combined with lighting and reflection computations, is the primary performance bottleneck. This observation is further illustrated by the accompanying performance graph, which highlights the impact of lighting on GPU workload.

Nb spheres	**Lighting**	**GPU Draw**	**GPU delay to exec**	**CPU setup**	**CPU DrawFrame**
~2000	ON	32,32 ms (~31 FPS)	3,05 ms	13,04 µs	142,37 µs (~0,85 %)
	OFF	13,18 ms (~76 FPS)	4,07 ms	19,28 µs	147,08 µs (~0,88 %)
~3000	ON	120,68 ms (~8 FPS)	4,42 ms	28,09 µs	144,23 µs (~0,87 %)
	OFF	14,28 ms (~70 FPS)	5,17 ms	12,88 µs	361,23 µs (~2,16 %)

TABLE 2 – Comparison between fragment shader and compute shader

GPU Draw : time spent rendering on the GPU.
GPU delay to exec : latency between command submission and actual GPU execution.
CPU setup : CPU time required to prepare the draw calls.
CPU DrawFrame : total CPU time per frame, with the approximate cost in percentage of the frame duration.

3.2 Performance and Realism Evaluation

The integration of compute shaders and a binary space partitioning structure has led to significant improvements in execution efficiency. As demonstrated in the previous tables, frame rendering times were reduced, and GPU workload was more evenly distributed. These gains are primarily attributed to the decoupling of rendering from the rasterisation pipeline, as well as to the reduction of unnecessary distance evaluations made possible by spatial subdivision.

While CPU usage remained low and stable across all test cases, the GPU draw time increased considerably when lighting was enabled. This confirms that the primary performance bottleneck lies in the raymarching process itself, particularly when combined with shading and reflection computations. These findings are consistent across multiple hardware configurations.

From a visual perspective, the implemented blending techniques, surface normals derived from SDF gradients, and Phong-based lighting collectively contribute to a convincing graphical representation of molten glass. The transitions between forms and reflections offer strong visual cues related to material curvature and transparency. It is important to note, however, that these effects are purely visual ; no physically-based simulation has been implemented.

Overall, the current rendering pipeline demonstrates a favourable compromise between realism and computational performance, providing a strong foundation for future extensions in interactive environments.

3.3 Limitations and Ongoing Work

Although the system has reached a functionally complete stage, several key features remain under development or require refinement to reach the intended level of interactivity and realism.

First, the current implementation of tree traversal in the compute shader is functional but requires updates to resolve known issues, particularly related to the consistency of node access and traversal order. The tree creation is on the CPU side and it is not optimal in terms of performance and memory usage.

Secondly, lighting remains a performance bottleneck. While basic Phong shading is implemented, recursive lighting effects such as multiple reflections are not yet fully supported. Enabling more accurate and layered lighting computations is expected to improve realism, but requires careful control of performance cost.

Another technical limitation lies in the frame rendering strategy. The system currently operates without double or triple buffering, which affects frame coherence and responsiveness under load. A buffered rendering pipeline is planned to improve visual stability and latency.

Several advanced features are also under consideration. The transparency model, although partially implemented, does not yet support full refraction and light propagation through complex materials. Supporting larger point clouds remains a performance challenge mainly in memory management. Additionally, the integration of a cube map for environment-based reflections is planned to enhance spatial coherence and visual immersion.

Finally, the introduction of a time-resolved buffer structure is being explored to enable real-time deformation and stream updates to the point cloud, thereby transforming the viewer from a static renderer into a dynamic simulation context.

3.4 General Discussion

The results obtained confirm that the raymarching-based approach provides a flexible and expressive framework for the real-time visualisation of molten glass. By combining implicit surface rendering, compute shaders, and spatial partitioning, the system can depict continuous, deformable volumes with convincing visual fidelity.

The visual feedback, achieved through blending operations, dynamic shading, and surface-level lighting, effectively reproduces the perceptual properties of molten glass without resorting to complex mesh structures or physical modelling. This focus on graphical realism, rather than physical accuracy, has enabled real-time performance even on moderately equipped systems.

The architecture remains extensible. The successful integration of the binary tree structure demonstrates the feasibility of incorporating spatial data structures within compute shaders, paving the way for more advanced interactions with point cloud data. The modular design also facilitates the addition of future features such as transparency, cube map reflections, and point cloud deformation.

More broadly, the project demonstrates how modern graphics techniques can support traditional artistic domains by offering tools for virtual training, experimentation, and cultural dissemination. The current implementation provides a robust base for future developments, including interactivity, adaptive streaming, and ultimately, physically-informed graphical simulation.

4 Conclusions, future prospects

This work was carried out as part of a broader research initiative led by the LORIA, aiming to create an immersive augmented reality experience for glassblowing. The contribution presented here focused exclusively on the real-time visual rendering of molten glass. No physical modelling of the material was performed ; the project remained within the scope of graphical representation, to produce a visually convincing and interactive depiction of molten glass.

To meet this objective, a raymarching-based rendering pipeline was developed, relying on implicit surfaces and compute shaders to depict smooth, deformable volumes in real time. Several techniques were implemented to enhance visual realism, including shape blending, Phong lighting, Fresnel effects, and a transparency model. In parallel, a custom space partitioning algorithm was introduced to optimise point cloud traversal through a binary tree structure.

Performance evaluations confirmed the scalability of the compute pipeline and demonstrated the visual plausibility of the molten glass rendering, without introducing overhead from the space partitioning system. These results validate the feasibility of using raymarching in conjunction with point cloud data for interactive and expressive visual applications.

This implementation provides a robust and extensible foundation for further development. Immediate technical priorities include :

- optimizing tree traversal within the compute shader ;
- optimizing recursive lighting and reflections ;
- introducing double or triple buffering to store the binary tree and the point cloud in different buffers to push the memory limits
- finalising the transparency model with full support for refraction ;
- integrating environment-based cube maps ;
- execute the space partitioning on the GPU side to get better performances.

More broadly, the project sets the stage for the eventual integration of physically-based simulation models. While the current system focuses on visual fidelity, the long-term vision is to transform the viewer into a real-time simulation framework, where user-driven deformations and realistic material behaviours can be computed interactively.

5 Bibliographical references

[1] Bruce Naylor. Constructing good partitioning trees. *Graphics Interface*, September 2001.

[2] Floney Yang. Raymarching collision, November 2024.

[3] Hecomi. Raymarching collision project (github), November 2024.

[4] Sakri Koskimies. Raymarching engine (github), November 2024.

[5] Adrian Biagioli. Raymarching presentation, November 2024.

[6] Nabil Mansour. SDF raymarching, November 2024.

[7] Wojciech Grega, Adam Pilat, and Andrzej Tutaj. Modelling of the glass melting process for real-time implementation. *International Journal of Modeling and Optimization*, 5(6) :366–373, 2015. Number : 6.