Combining Signed Distance Fields and Raymarching with Extended Position Based Dynamics for Rendering and Physics in Virtual Worlds

Antonio Villalta Isidro

(BSc) Computer Games Applications Development, 2024

School of Design and Informatics
Abertay University

Table of Contents

Table of Figures

Acknowledgements

I would like to thank my family and friends for their continued and unyielding support during not only the development of this project, but my entire time at Abertay University.

I have learnt so much over this past four years and would like to show my gratitude to all of the staff, who with all of their hard work, make Abertay the University it is.

Finally, a special thank you to my supervisor Mr Matthew Bett, who kept me on track during this project and guided me with helpful feedback.

Abstract

This dissertation investigated the implementation and evaluated the performance of two advanced techniques, Signed Distance Fields (SDF) and XPBD (Extended Position Based Dynamics), in the context of real-time rendering and physics simulation for video games. The primary objective was to assess the viability of integrating these techniques into modern game development pipelines to enhance visual quality and realism.

The research began with a comprehensive exploration of the theoretical foundations of SDF and XPBD, highlighting their principles, advantages, and potential applications in video game graphics and physics simulations. Through a series of practical implementations within the Unity game engine, the research examined the rendering performance and physics simulation capabilities of both techniques.

Results from extensive testing revealed critical insights into the strengths and limitations of SDF and XPBD. While SDF demonstrated remarkable precision and control over visual output, particularly in the creation of unique and striking visual effects, its performance suffered when rendering a large number of objects due to CPU and GPU resource demands. Conversely, XPBD showcased promising capabilities in simulating soft bodies with high flexibility and realism, albeit with significant strain on CPU resources, which led to suboptimal frame rates.

The research identified several areas for future research and optimization, including the exploration of optimisation strategies to enhance the performance of SDF and XPBD implementations and integration with existing rendering and physics systems. Furthermore, collaborative research efforts, user experience studies, and interdisciplinary applications were proposed to further explore the potential of these techniques beyond just video games.

Overall, this dissertation contributed valuable insights into the practical implications of utilizing SDF and XPBD in video game development, hopefully paving the way for future advancements in real-time rendering and physics simulation technologies.

Keywords:
Signed Distance Fields (SDF), Raymarching, Position Based Dynamics (PBD), Extended Position Based Dynamics (XPBD), Rendering, Physics simulation, Shader, Performance, Compute Shader.

Abbreviations, Symbols and Notation

SDF	Signed Distance Fields
PBD	Position Based Dynamics
XPBD	Extended Position Based Dynamics
FPS	Frames per Second
CPU	Central Processing Unit
GPU	Graphics Processing Unit
CS	Compute Shader

Chapter 1 Introduction

In the ever-evolving landscape of the video game industry, innovation reigns supreme. With each passing year, developers push the boundaries of technology and creativity, striving to create experiences that captivate and immerse players in worlds both fantastical and familiar (Lowood, 2009). At the heart of this pursuit lies a vast range of techniques and technologies, each playing a crucial role in shaping the games that players both play and cherish.

The games industry is a behemoth, with billions of players worldwide and a market that continues to expand at an exponential rate. From indie studios crafting intimate experiences to AAA juggernauts pushing the boundaries of graphical fidelity and realistic physics, the diversity and depth of modern video games are unparalleled. However, behind the scenes, a complex tapestry of techniques and technologies drives the creation of these digital worlds.

At the forefront of rendering technology lies rasterization, a tried-and-true method for transforming 3D models into pixel-perfect images (Lai and Chung, 2015). Utilized in the vast majority of modern games, rasterization leverages the computational power of GPUs to rapidly render complex scenes in real-time. Yet, while rasterization excels at rendering triangles and basic lighting effects, it struggles to capture the nuances of light and shadow that define realistic environments.

In addition to rendering, physics simulations play a pivotal role in creating believable interactions within the game world. Traditionally, force and impulse-based physics engines have dominated this domain, providing developers with tools to simulate rigid bodies, collisions, and physical forces (Bourg, 2004). These techniques support the dynamic nature of digital environments, imbuing them with a sense of realism and interactivity.

Within the realm of physics simulations, rigidbodies stand as a cornerstone of interactive dynamics, governing the movement and interaction of objects within the game world. From tumbling boulders to swaying trees, rigidbodies simulate the physical properties of objects with precision and accuracy. However, as games strive for greater realism and complexity, developers seek new methods to simulate softer, more deformable materials (Müller et al., 2007).

Introducing Raymarching and SDF, innovative techniques that challenge the conventions of traditional rendering. By tracing rays through volumetric data represented by SDFs (Hart, 1996), developers can create stunning visual effects and detailed environments with unparalleled flexibility. From volumetric fog to complex procedural landscapes, raymarching and SDFs offer a new frontier for visual storytelling and artistic expression in video games (Osher and Fedkiw, 2003).

Yet, it is important to understand that what truly sets a game apart is not merely its technical prowess, but its ability to evoke emotions, spark imaginations, and transport players to new worlds. A game's uniqueness lies in its ability to tell a compelling story, create memorable characters, and offer gameplay experiences that resonate with players long after the credits roll.

Moreover, what makes a game feel real is not just its graphical fidelity or physics simulation, but its attention to detail, its adherence to the rules of its own universe, and its ability to suspend disbelief. From the subtle animations of characters to the ambient sounds of a bustling city, every aspect of a game contributes to its sense of immersion and realism.

Nevertheless, the implementation of innovative techniques to enhance both graphical fidelity and achieve new levels of realism, certainly are important aspects of what can make a game feel unique and special.

Innovation, too, is a driving force in game development, pushing the boundaries of what is possible and redefining the medium itself. Whether through groundbreaking gameplay mechanics, revolutionary storytelling techniques, or bold artistic visions, innovative games inspire awe and admiration, leaving an indelible mark on the industry and its players.

Against this backdrop of technological advancement and creative exploration, this dissertation set out to explore the capabilities, challenges, and potential applications of raymarching, SDF, and XPBD. The aim being to dissect these technologies, analyse their strengths and limitations, and assess their potential impact on the gaming landscape.

"How do raymarching, SDF, and XPBD compare to traditional rendering and physics simulation methods in terms of visual fidelity, computational efficiency, and creative potential in video game development?"

“Are raymarching, SDF and XPBD viable technologies that can be used in modern games to improve graphical fidelity, realism and creative potential, or are traditional rendering and physics simulation methods still the better option for game developers?”

By delving into this question, a deeper understanding of the evolving tools and techniques that shape the video games that millions enjoy, while also paving the way for future innovation and exploration in the field of interactive entertainment is expected.

Chapter 2 Literature Review

This chapter aims to provide an understanding of the ever-growing need for better visuals, physics, and performance in video games. There are many different techniques that have been used over the years for tackling both visuals and physics, with the main bottleneck being the performance restrictions that come with more realistic and accurate methods (Geer, 2006).

2.1 Performance in Video Games

In recent years, the video games industry has witnessed a shift towards highly immersive and visually impressive games. There is an expectation that new video games will have seamless transitions between areas and levels with fluid interactions with the world's elements, all while maintaining high frame rates and visual fidelity (Fischer et al., 2010). The heavy use of real time rendering and physics simulations, put an ever-increasing pressure on video game developers, who have to not only meet the expectations, but surpass them.

Rendering complex 3D scenes in real time is a significant challenge that has been taken for granted, particularly in achieving a balance between visual quality and performance (Watt and Policarpo, 2000). Traditional rendering techniques such as rasterization, while efficient, may struggle to cope with the demands that come with rendering intricate scenes with dynamic lighting and geometry with a high polygonal count (Lai and Chung, 2015).

When it comes to physics simulations, the two most common approaches are force, and impulse based, with them being often turned into a hybrid method to manage different situation like gravity and collisions. These methods are used to control the behaviour of elements in the game world, which must accurately simulate realistic interactions between them, while maintaining high computational efficiency (Bourg, 2004).

With games increasing ambitions in scope and scale, the challenge of maintaining satisfactory performance becomes more pronounced, requiring innovative approaches to address performance bottlenecks.

When it comes to performance expectations, players usually prioritise frame rate, with smooth and consistent frame rates considered essential for fluid gameplay and responsive controls. Regardless of the video game they play, their experience should be free from stuttering, lag and frame drops which disrupt immersion and hinders gameplay. Graphics quality is another critical aspect of expectations for players, with many seeking visually stunning environments, realistic character models, and immersive visual effects.

From the perspective of developers, meeting the performance expectations of players, creates both technical and creative challenges that require careful optimisation and resource management. Balancing visual fidelity with performance optimisation is a constant concern for developers, who must use innovative rendering techniques and hardware capabilities while ensuring compatibility with a wide range of gaming platforms and hardware configurations.

Developers utilise a variety of strategies to optimize performance, including efficient rendering pipelines, dynamic level-of-detail (LOD) systems, and resource streaming techniques to minimize load times and maximize frame rates. Moreover, developers prioritize code optimization and performance profiling to identify and address bottlenecks in CPU, GPU, and memory usage, to ensure smooth and responsive gameplay.

2.2 History of Rendering and Physics in Video Games

The evolution of rendering and physics in video games is a testament to the never-ending pursuit of realism and immersion in games. From the, now considered, rudimentary graphics and physics simulations of early arcade games to the photorealistic environments and complex physics engines of modern AAA titles, game development has been marked by continuous innovation and technological advancements.

The origins of rendering and physics can be traced to the 1970s and 1980s (Lowood, 2009). During this era, games were primarily rendered using 2D sprites and tile-based graphics, with limited capabilities for simulating realistic physics. Classic games such as Pong (Atari, 1972) and Space Invaders (Taito, 1978) showcase simple graphics using rasterization and do not display any complex physics simulations.

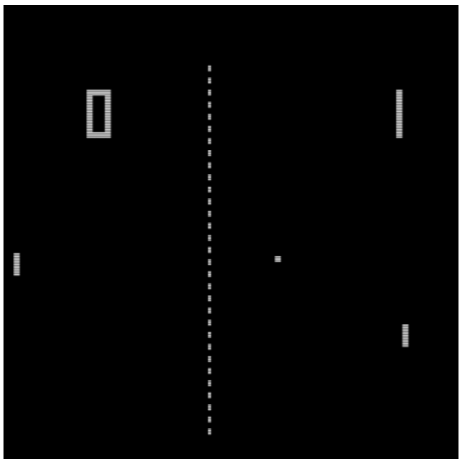


Fig. 2.1. Screenshot of Pong

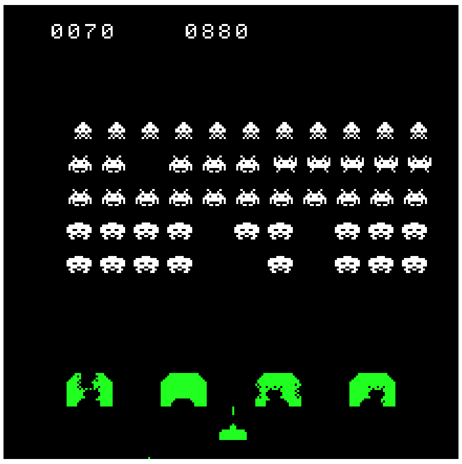


Fig. 2.2. Screenshot of Space Invaders

The introduction of Personal Computers in the 1980s (Beaudry, Doms and Lewis, 2010) brought a new era of innovation. Titles such as Donkey Kong (Nintendo R&D1, 1981) and Super Mario Bros. (Nintendo R&D4, 1985) introduced more advanced rendering techniques, including scrolling backgrounds and parallax scrolling, to create the illusion of depth and movements in 2D environments. However, physics simulations remained simplistic, with games relying on basic collision detection using impulse based algorithms and simplified motion physics with a crude gravity system using forces.

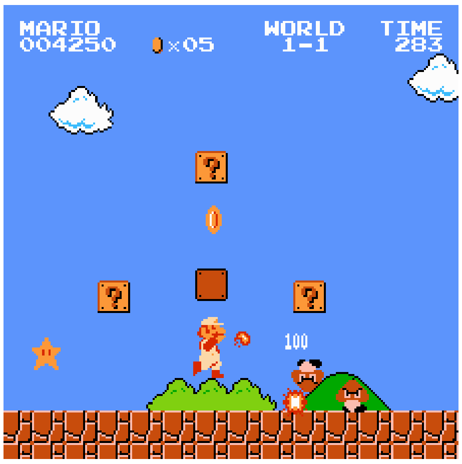


Fig. 2.3. Screenshot of Super Mario Bros.

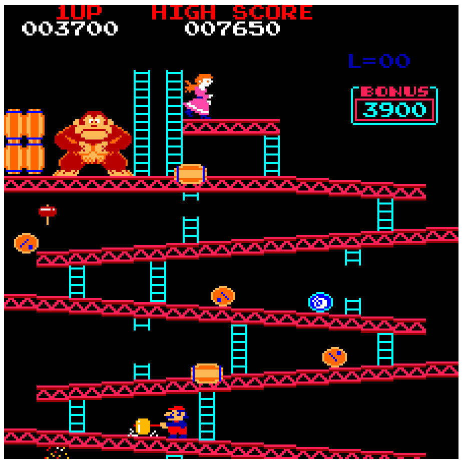


Fig. 2.4. Screenshot of Donkey Kong

In the 1990s, a revolution in rendering took place, with the widespread adoption of 3D graphics in games, thanks to advancements in hardware, such as dedicated graphics processing units (GPUs) (Dally, Keckler and Kirk, 2021), which enabled developers to render 3D elements in real time. Before this, Wolfenstein 3D (id Software, 1992) and Doom (id Software, 1993) pioneered the use of raytracing to render 3D worlds (Hart, 1996), albeit with limited textures and lighting.

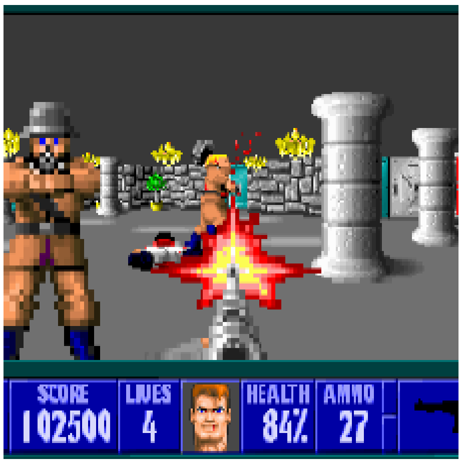


Fig. 2.5. Screenshot of Wolfenstein 3D

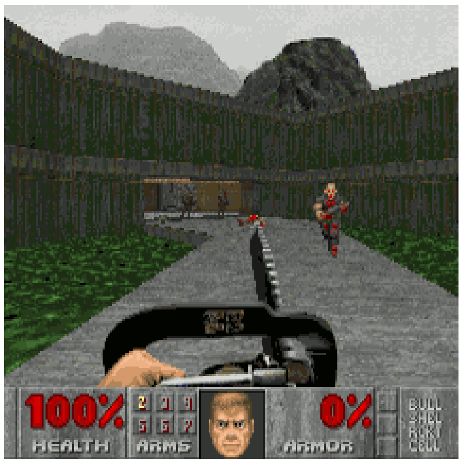


Fig. 2.6. Screenshot of Doom

The release of the Nintendo 64 and Sony PlayStation in the mid-1990s, brought with them the era of polygonal 3D graphics, allowing developers to create fully realized 3D environments with texture mapping, lighting, and shading. Games like Super Mario 64 (Nintendo EAD, 1996) and Tomb Raider (Core Design, 1996) highlighted the potential of 3D rendering in delivering large worlds and complex character animations. Meanwhile, advancements in physics simulations led to the adoption of more sophisticated algorithms for collision detection, the combination of simplified rigid-body simulation and kinematic constraints, friction, and momentum.

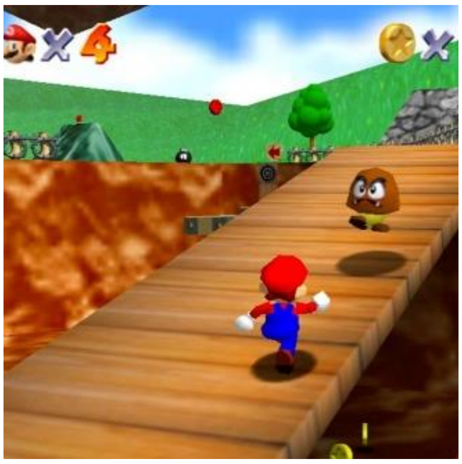


Fig. 2.7. Screenshot of Super Mario 64

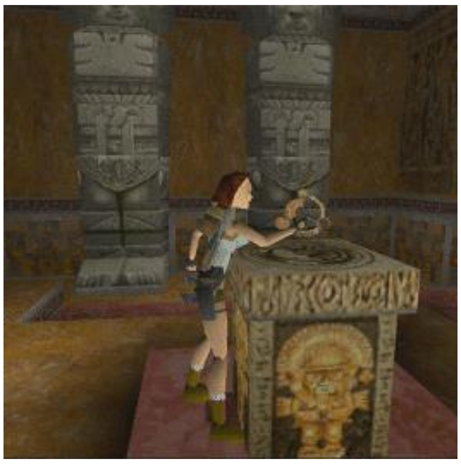


Fig. 2.8. Screenshot of Tomb Raider

The early 2000s saw the introduction of programmable GPUs (Dally, Keckler and Kirk, 2021) and advanced rendering APIs such as DirectX (Microsoft, 1995) and OpenGL (Silicon Graphics and Khronos Group, 1992), which allowed developers to gain a new level of flexibility when it came to implementing complex visual effects and post-processing filters. Games like Half-Life 2 (Valve, 2004) and Crysis (Crytek, 2007) pushed the boundaries of graphical fidelity with dynamic lighting and realistic physics interactions improving and perfecting past utilised techniques.

Fig. 2.9. Screenshot of Half Life 2

Fig. 2.10. Screenshot of Crysis

At the same time, physics simulations underwent major advancements, with the adoption of middleware solutions like Havok (Telekinesys Research Limited, 2000) and PhysX (Nvidia, 2004), enabling developers to integrate sophisticated physics engines into their games. Titles like Grand Theft Auto IV (Rockstar North, 2008) and Red Dead Redemption (Rockstar San Diego, 2010) demonstrated the integration of advanced physics simulations for realistic vehicle handling and environmental destruction.

Fig. 2.11. Screenshot of Red Dead Redemption

Fig. 2.12. Screenshot of Grand Theft Auto IV

In recent years, the pursuit of photorealism and cinematic immersion has driven further innovation in rendering and physics technologies. Real-time ray tracing, enabled by modern GPUs, has revolutionized lighting and reflections in games, allowing for incredible visual realism and fidelity. Games like Metro Exodus (4A Games, 2019) and Cyberpunk 2077 (CD Projekt Red, 2020) make use of raytracing for accurate lighting and shadows that create realistic environments.

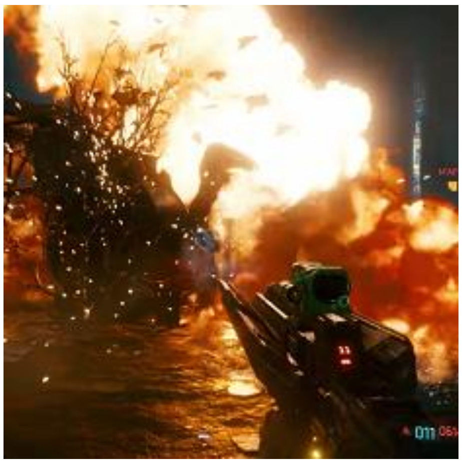


Fig. 2.13. Screenshot of Cyberpunk 2077

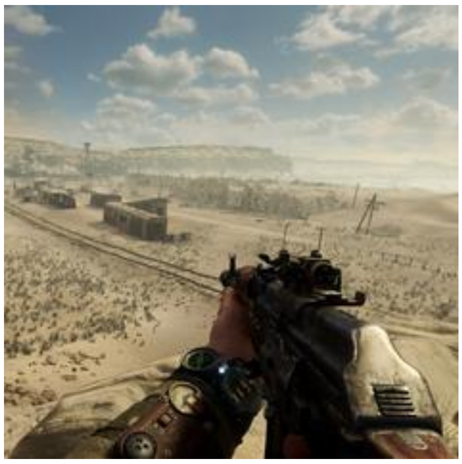


Fig. 2.14. Screenshot of Metro Exodus

Physics engines have evolved to support complex soft body dynamics, fluid simulations and volumetric effects, allowing for a new level of realism and interactivity in games. Titles such as Half-Life: Alyx (Valve, 2020) and Death Stranding (Kojima Productions, 2019) showcase the integration of advanced physics simulations for realistic object interactions, environmental effects, and dynamic animations.

Fig. 2.15. Screenshot of Death Stranding

Fig. 2.16. Screenshot of Half Life: Alyx

2.3 SDF, Raymarching and XPBD

As hardware optimisations reach their maximum potential, techniques such as rasterization are showing their limitations, making developers explore cutting-edge solutions to revolutionize games once again.

Rasterization stands as the standard rendering method, owing to its efficiency, hardware acceleration, long-standing development, and widespread adoption. Its proficiency in rendering triangles, alongside dedicated GPU support tailored for accelerating rasterization calculations, cements its dominance. Over decades of refinement, rasterization has matured into a well-understood and reliable technique, integrated as the default option across most game engines and tools.

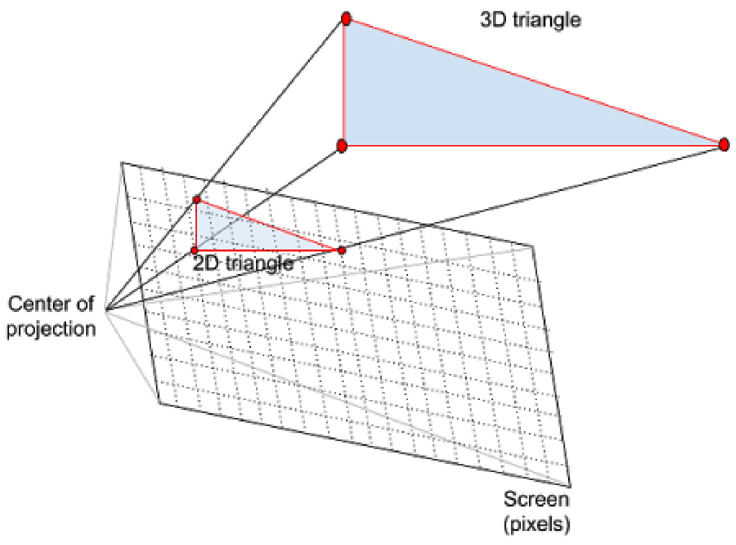


Fig. 2.17. Rasterization diagram

Nonetheless, rasterization presents inherent limitations, notably aliasing, restricted realism, challenges with transparency, and scalability constraints. Aliasing artifacts manifest as jagged edges along diagonal lines or curves on pixel grids, they are common in rasterization and necessitate computationally intensive anti-aliasing methods.

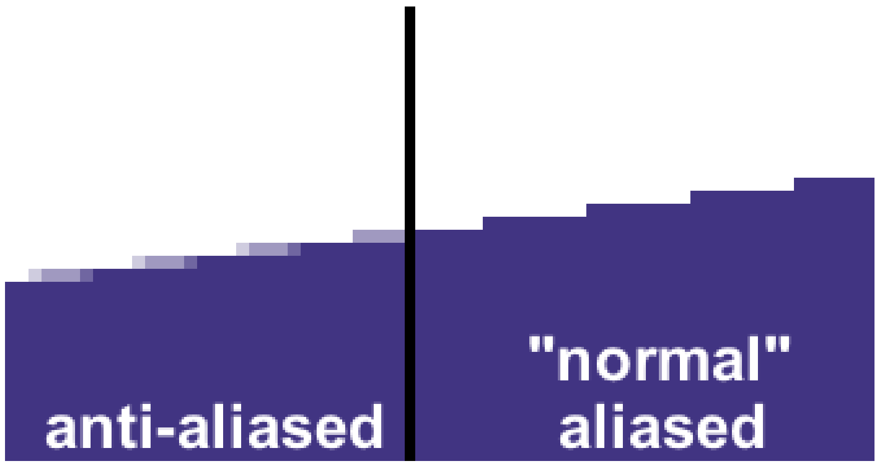


Fig. 2.18. Example of anti-aliasing and aliasing

Moreover, rasterization struggles to accurately reproduce intricate lighting effects such as soft shadows, caustics (light reflecting off reflective surfaces), and global illumination (indirect light bouncing around a scene), often requiring supplementary techniques like ray tracing. Transparent or translucent objects introduce additional challenges as well, demanding specialized approaches like alpha blending, which adds complexity to the rendering process.

Rendering using Raymarching and SDF represent a unique approach to rendering 3D scenes in real-time. Unlike traditional rasterization-based rendering techniques, which rely on polygonal meshes and complex shader pipelines, raymarching with SDF operates directly on volumetric data represented by mathematical distance functions.

Raymarching, also known as sphere tracing, is a rendering technique that involves tracing rays through a scene and iteratively marching along their paths until they intersect with objects or surfaces (Hart, 1996).

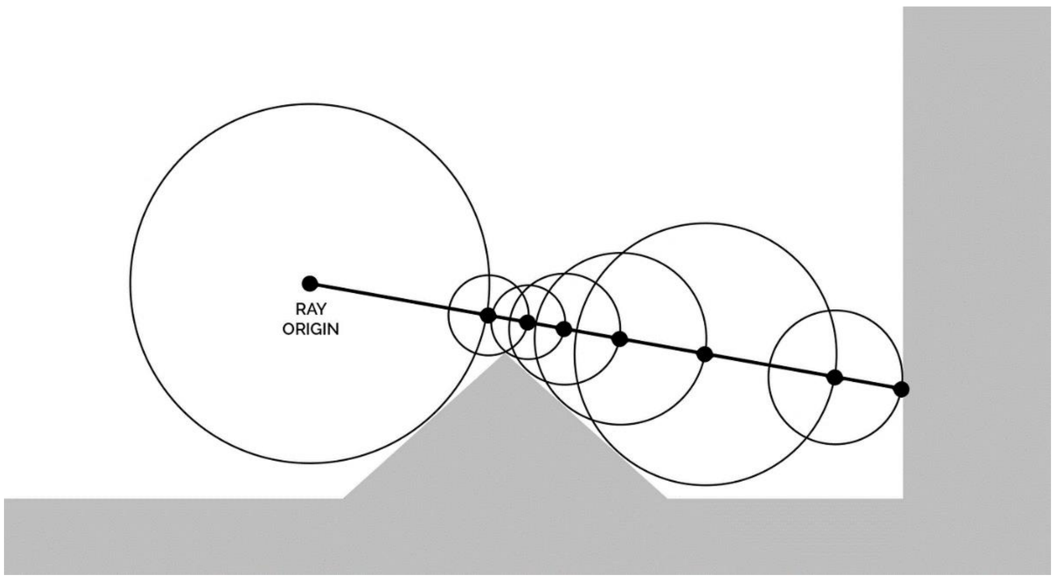


Fig. 2.19. Raymarching diagram

The concept of raymarching dates back to the early days of computer graphics research, with seminal works by John Hart in the 1980s and 1990s laying the groundwork for its development. Hart's pioneering research on distance fields and implicit surfaces paved the way for the formulation of raymarching as a viable rendering technique, enabling the rendering of objects defined by mathematical equations rather than discrete geometric primitives.

One of the key advantages of raymarching with SDF is its ability to render complex geometric shapes and intricate scenes with minimal computational overhead. By making use of the inherent properties of distance fields, such as smoothness and continuity, developers can create highly detailed and visually appealing environments without the need for traditional mesh-based geometry. Additionally, raymarching with SDF facilitates the implementation of advanced rendering effects such as ambient occlusion, soft shadows, and global illumination in a more performant manner.

SDFs play a fundamental role in raymarching, serving as mathematical representations of objects and surfaces in 3D space. An SDF assigns a signed distance value to every point in space, indicating the distance to the nearest surface along the ray direction. Positive values indicate points outside the surface, negative values indicate points inside the surface, and zero values indicate points on the surface itself (Osher and Fedkiw, 2003).

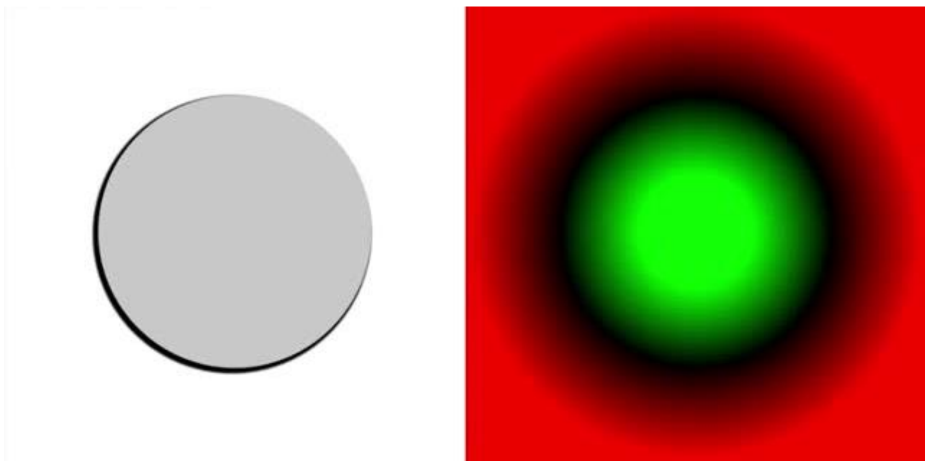


Fig. 2.20. Visualization of SDF

Over the years, SDFs have become increasingly prevalent in computer graphics, finding applications in a wide range of fields, including rendering, modelling, animation, and simulation. The adoption of SDFs has been driven by advancements in hardware capabilities, rendering algorithms, and software tools, as well as the growing demand for realistic and immersive virtual environments.

The two fundamental approaches used in physics are force based and position based. Similarly to rasterization, these methods have become standard in game development, owing to extensive research and their widespread use. While both approaches hold relevance and complement each other, they are not without limitations.

Force based physics produces significant computational overheads by computing the effects of forces on an object's motion every frame. Moreover, inherent numerical instability issues can lead to the accumulation of small errors over time, potentially resulting in unrealistic behaviour or crashes.

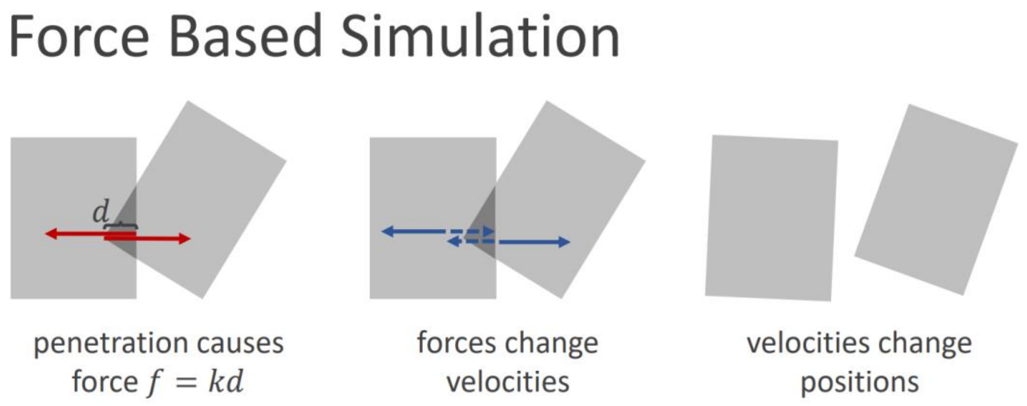

Force Based Simulation
d
penetration causes force $f = kd$
forces change velocities
velocities change positions

Fig. 2.21. Force based physics pipeline

On the other hand, impulse based physics exhibit limitations in accurately simulating complex collisions, particularly those involving multiple contact points or irregularly shaped objects. Additionally, elastic collisions may not be adequately represented, restricting the use cases of this method in scenarios involving soft bodies. It also requires the fine-tuning of parameters to achieve the desired physical behaviours, which is a cumbersome and intricate process.

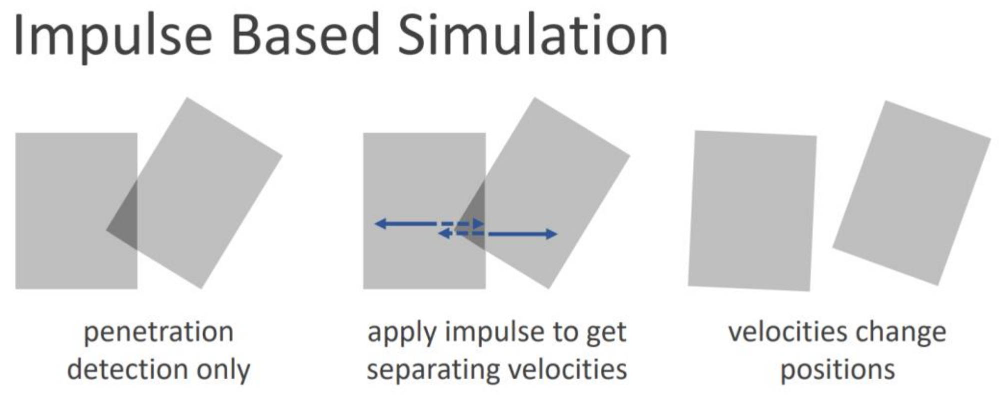

Impulse Based Simulation
penetration detection only
apply impulse to get separating velocities
velocities change positions

Fig. 2.22. Impulse based physics pipeline

Lastly, both force-based and impulse-based approaches encounter challenges in contact resolution between objects, and often exhibit poor stability at higher object velocities.

PBD and its improved counterpart, XPBD represent a shift in physics simulation for video games, offering a versatile and efficient framework for simulating complex interactions between objects in real-time. Unlike traditional rigid body dynamics engines, which rely on discrete time-stepping methods to update object positions and velocities, Position Based Dynamics operate on continuous constraints that handle the behaviour of objects over time (Müller et al., 2007).

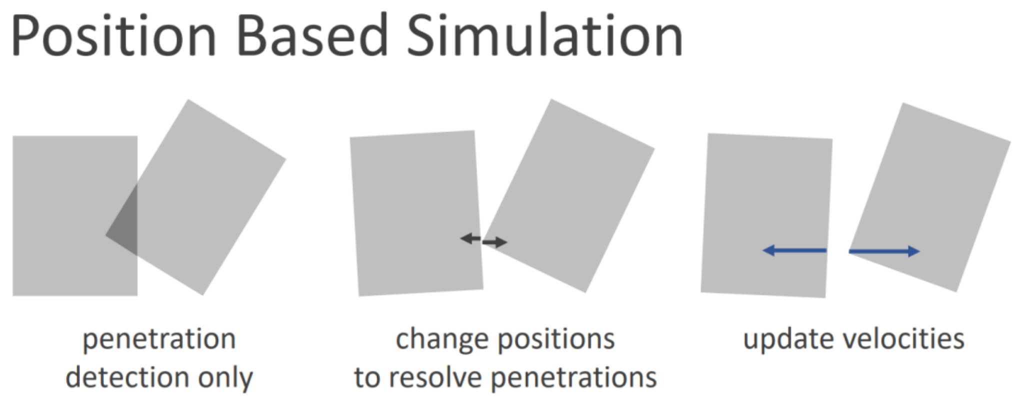

Position Based Simulation
penetration detection only
change positions to resolve penetrations
update velocities

Fig. 2.23. PBD physics pipeline

PBD is a physics simulation technique that originated from the field of computer animation, particularly in the context of simulating deformable objects and soft bodies (Müller et al., 2007). PBD was first introduced in academic research in the early 2000s as an alternative to traditional methods such as Finite Element Analysis (FEA) and Mass-Spring Systems (MSS) for simulating dynamic behaviour in computer-generated animations.

The fundamental principle of PBD lies in the use of position constraints to enforce physical constraints and simulate dynamic behaviour. By iteratively updating the positions of particles or vertices in a simulated object to satisfy these constraints, PBD achieves stable and physically plausible simulations.

XPBD extends the principles of PBD to manage a broader range of physical interactions and constraints, including collisions, friction, and constraints between multiple objects. XPBD introduces enhancements to the basic PBD framework, such as improved collision handling algorithms, adaptive time-stepping, and enhanced constraint solvers, to enable more accurate and efficient simulations (Macklin, Müller and Chentanez, 2016).

One of the key innovations of XPBD is its ability to handle collisions between objects of arbitrary shapes and sizes without the need for explicit collision detection routines. Instead, XPBD uses implicit constraints derived from the geometry of the objects to resolve collisions and maintain physical realism (Müller et al., 2020).

At the core of XPBD is the concept of position-based iteration, where object positions are iteratively adjusted to satisfy a set of constraints derived from physical laws and user-defined parameters. This iterative approach allows for stable and accurate simulation of a wide range of phenomena, including collisions, deformations, and soft body dynamics, without the need for complex solver algorithms or high computational costs.

Since their inception, PBD and XPBD have undergone significant refinement and optimization, driven by advancements in computational algorithms, hardware acceleration, and real-time rendering techniques.

The integration of raymarching with SDF and XPBD into video game development opens up a wide array of possibilities for creating immersive, interactive, and visually stunning gaming experiences. By combining the power of distance functions and position-based dynamics, developers can push the boundaries of realism and interactivity in virtual environments, from highly detailed and procedurally generated landscapes to unique 3D effects and dynamic physics interactions.

Furthermore, the efficiency and scalability of raymarching with SDF and XPBD make them well-suited for a wide range of gaming platforms, from high-end PCs and consoles to mobile devices and virtual reality headsets. As hardware capabilities continue to evolve and software optimizations improve, it is expected to see further advancements in rendering and physics simulation techniques, with raymarching with SDF and XPBD, perhaps, leading the way towards the next generation of video game experiences.

Chapter 3 Methodology

This section will go into detail into how both methods, raymarching, and SDF, and XPBD were implemented and how they could be combined to achieve the proposed results.

3.1 Unity Engine

Unity is a versatile and widely used game engine, known for its user-friendly interface and robust feature set, it provides a comprehensive suite of tools for game development, including rendering, physics simulation, audio, and scripting. It is intuitive workflow and cross-platform capabilities, has made it a popular choice for both indie developers and AAA studios.

3.1.1 Why a Game Engine

The project was developed inside of a game engine, Unity, this decision was taken as it would alleviate from making from scratch basic functionality needed for the project, like a rendering pipeline and basic physics. In addition, since this project aimed to create an innovative technique for future games, it was coherent to try to implement it within a game engine.

3.1.2 Implementation Plan

To facilitate the implementation of both raymarching with SDF and XPBD, the project featured three distinct developmental phases, each dedicated to specific rendering and physics techniques.

The first phase focused on the integration of raymarching and SDF, requiring the creation of a dedicated scene for its implementation. Here, the emphasis lied on developing the necessary algorithms and shaders to enable the accurate rendering of scenes using raymarching and SDF.

Next, the project transitioned to the implementation of PBD, serving as the foundation for the integration of XPBD. This phase involved the creation of a separate development suite dedicated to implementing PBD algorithms and simulating the physics interactions within the virtual environment.

Finally, the project finished in a comprehensive scene where the seamless combination of both raymarching with SDF and XPBD techniques was showcased. This final developmental stage integrated the rendering and physics components developed in the previous phases, providing a platform that demonstrated the synergistic effects of combining these advanced techniques in a unified environment.

3.2 Raymarching and Signed Distance Fields

When talking about implementing raymarching and SDF, the rendering pipeline becomes centre stage, as it would need to be modified, or tweaked to fit with the desired outcome.

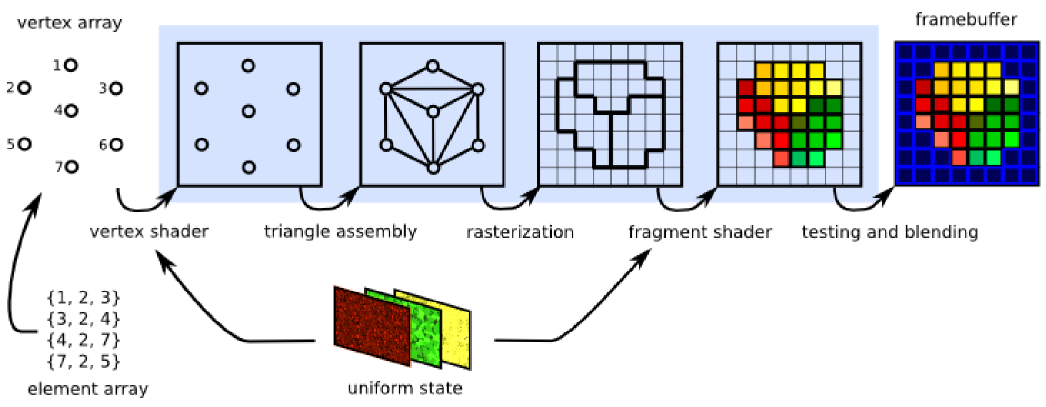


Fig. 3.1. Rendering pipeline

3.2.1 Possible Implementations

When developing a visual technique, the first inclination might be to create a shader to do it, with shaders being essential components in computer graphics, executed by the GPU to manipulate the appearance and behaviour of rendered objects in real-time. Written in specialized shading languages like HLSL (High Level Shader Language), GLSL (OpenGL Shader Language), or Cg (C for Graphics), shaders operate within the graphics pipeline to control aspects like colour, texture, lighting, and geometry.

The initial approach was to employ Unity's Standard Surface Shader script as the primary method. However, it swiftly became evident that this approach posed limitations. Basic shaders, inherent to Unity's Standard Surface Shader script, are confined to material usage, primarily designated for rendering individual objects within a scene. This constraint diverged from the project's objective, which necessitated for the application of raymarching and SDF across the entire virtual environment. Despite these challenges, this first approach validated the viability of the method, showcasing its functionality. Nonetheless, this implementation allowed for code reuse, easing subsequent iterations.

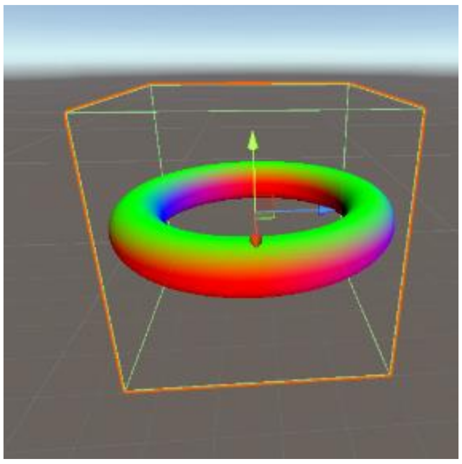


Fig. 3.2. Raymarched Torus inside a cube

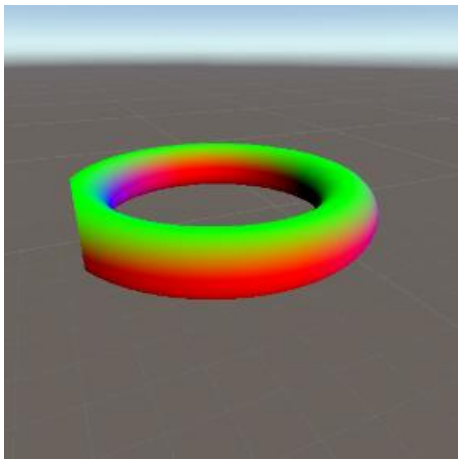


Fig. 3.3. Raymarched Torus cut due to cube

The resolution came from using Unity's rendering techniques to manage real-time rendering operations effectively. By using this functionality, manipulation, or deletion of the render texture, which represents the current scene frame projected onto the screen, became feasible, with the additional capability of generating entirely new render textures. To perform this process efficiently, a CS was developed to perform this task.

3.2.2 Compute Shaders

Compute shaders are a revolutionary aspect of modern graphics programming, allowing the use of the GPU's computational power for non-graphical tasks. Unlike traditional shaders, compute shaders focus on general-purpose computation, enabling parallel processing across GPU cores. This capability accelerates complex algorithms, simulations, and data processing tasks, making compute shaders indispensable in fields like real-time physics simulation.

Since there was data being sent over to the GPU, it must be added to a compute buffer, in order to be transferred properly from the CPU. For this purpose, the scene consisted of custom objects which contained all of the necessary data that needed to be passed onto the GPU. These objects contained information about their position, scale, colour, shape, operation, etc.

The data, however, needed to be organized and packaged before it can be sent, for this purpose, the data was sorted according to the operation the shape would perform, as they needed to be rendered in a specific order so not to influence other types of SDF operations.

With the data gathered and properly sorted, it could now be attached onto a compute buffer and sent over to the GPU for the CS to use.

3.2.3 SDFs

Within the CS, the entire logic behind the raymarching and SDF existed. Beginning of course, with the raymarching process, which unlike traditional rasterization methods, which determine the colour of pixels by projecting geometry onto a two-dimensional screen, it traced rays through a volume of space to determine the appearance of each pixel.

This process involved casting a ray from the camera’s viewpoint into the scene and iteratively advancing along the ray to determine the intersection points with objects in the scene.

At each step along the ray, the algorithm evaluated an SDF to determine the distance to the nearest object surface. The SDF represents the distance from a point in space to the surface of an object, with positive distances indicating points outside the object and negative distances indicating points inside the object.

By iteratively evaluating the SDF and advancing along the ray, raymarching could accurately determine the intersection points with objects in the scene.

Once an intersection point was found, the algorithm calculated the surface normal at that point, which was then used to determine the shading of the pixel.

This shading calculation involved factors such as lighting, reflections, and shadows, depending on the lighting data gathered from the game scene.

Finally, the colour of the pixel was determined based on the shading calculation, and the process was repeated for each pixel in the image, which was precisely why using a CS with access to multiple threads made this technique much more performant.

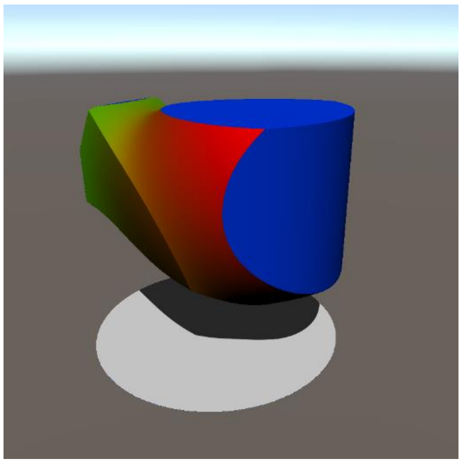


Fig. 3.4. SDF Blend operation inside cylindrical mask

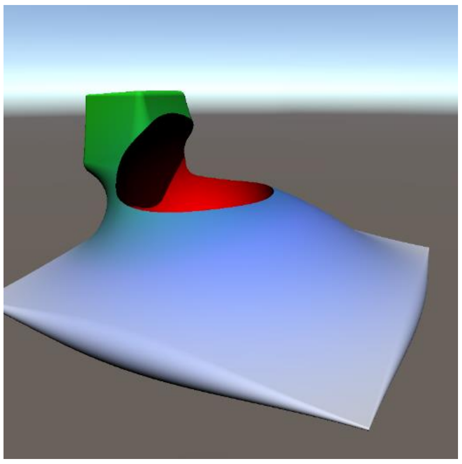


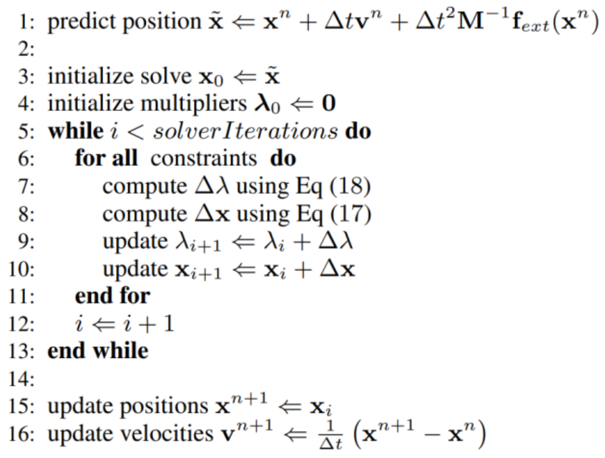
Fig. 3.5. SDF Blend operation with a spherical cut

3.3 Extended Position Based Dynamics

To effectively implement PBD and subsequently enhance its capabilities through XPBD, it was imperative to start with the creation of a basic demonstration of the fundamental algorithm. Adopting this approach to implementation, facilitated a systematic and methodical progression through each facet of the method, thereby enhancing manageability throughout the developmental process.

3.3.1 PBD and XPBD

The foundational PBD algorithm encompasses four key phases: pre-solve, constraint resolution, collision detection, and post-solve (Müller et al., 2007). These phases are executed iteratively within a loop of substeps per frame to ensure optimal precision in results (Müller et al., 2020).



```
predict position $\tilde{\mathbf{x}} \Leftarrow \mathbf{x}^n + \Delta t \mathbf{v}^n + \Delta t^2 \mathbf{M}^{-1} \mathbf{f}_{ext}(\mathbf{x}^n)$

initialize solve $\mathbf{x}_0 \Leftarrow \tilde{\mathbf{x}}$
initialize multipliers $\boldsymbol{\lambda}_0 \Leftarrow \mathbf{0}$
while $i < solverIterations$ do
    for all constraints do
        compute $\Delta\lambda$ using Eq (18)
        compute $\Delta\mathbf{x}$ using Eq (17)
        update $\lambda_{i+1} \Leftarrow \lambda_i + \Delta\lambda$
        update $\mathbf{x}_{i+1} \Leftarrow \mathbf{x}_i + \Delta\mathbf{x}$
    end for
    $i \Leftarrow i + 1$
end while

update positions $\mathbf{x}^{n+1} \Leftarrow \mathbf{x}_i$
update velocities $\mathbf{v}^{n+1} \Leftarrow \frac{1}{\Delta t}\left(\mathbf{x}^{n+1} - \mathbf{x}^n\right)$
```

Fig. 3.6. PBD pseudocode loop

Initially, the focus was directed towards the pre-solve step, which facilitated unconstrained integration and served as an important starting point for implementation.

PBD operates by employing particles which are the vertices of a given mesh, serving as recipients of simulated forces (Müller et al., 2007). Consequently, it becomes imperative to store essential attributes such as position, previous position, velocity, and mass for these particles within arrays to enhance computational efficiency.

In the context of simulating a cube, Unity’s "Mesh" class facilitated the extraction of vertex position data, a prerequisite for initiating the simulation.

Once these vertices were acquired, they were subjected to a negative force in the y-axis to replicate the effects of gravity. This implementation of gravity, though linear and lacking acceleration, was sufficient at this stage of development.
However, challenges arose as the cube failed to maintain its shape and penetrated through surfaces. To rectify this issue, the collision detection stage was incorporated, involving a simple positional assessment of each particle to ensure it remained above the floor boundary.

Additionally, the implementation of particle constraints (Müller et al., 2007), particularly volume and length constraints, became essential for preserving the cube's structural integrity. To address the computational complexity associated with volume calculations, modern XPBD simulations have started to adopt the use of tetrahedralized meshes (Müller, no date). This approach enables the acquisition of crucial data components per tetrahedron, including a particle per vertex, a distance per edge, and a volume per tetrahedron, thereby optimizing computational efficiency and facilitating accurate preservation of mesh geometry.

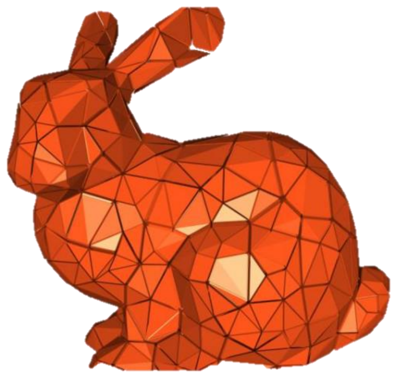


Fig. 3.7. Tetrahedralized Stanford bunny

3.3.2 Tetrahedrons

Tetrahedralized meshes deviate from conventional meshes by incorporating vertices within the mesh to form clusters of tetrahedra. These tetrahedra proved invaluable as they facilitated consistent volume calculations independently of the mesh's external shape, thereby enhancing streamlining and optimization efforts. The conversion process from a standard mesh to a tetrahedralized one was intricate but fortunately only necessitated execution once.

The Incremental Delaunay Method was the chosen approach for generating the tetrahedralized mesh, although it was also applicable for triangulation purposes (Lin and Liu, 2009). Commencing with four temporary points arranged to form a large tetrahedron encapsulating all four points, the method gradually incorporates additional points.

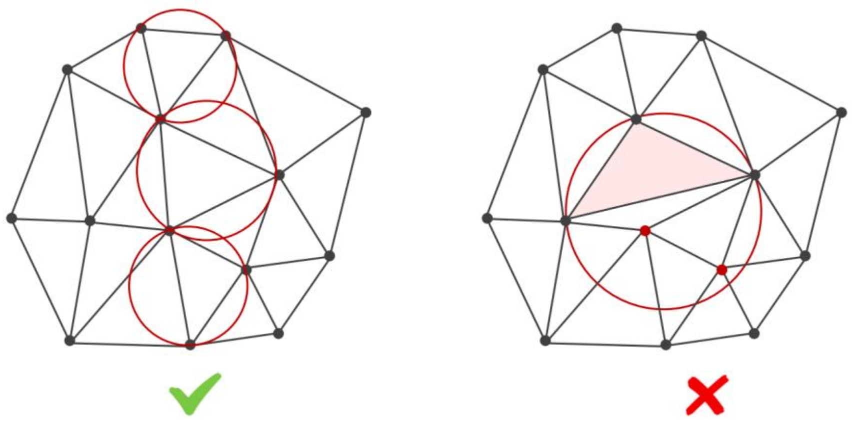


Fig. 3.8. Delaunay compliant and non-compliant mesh

Any tetrahedron contravening the Delaunay condition, stipulating that the circumsphere of any tetrahedron should solely encompass the four adjacent points (Lin and Liu, 2009), is promptly removed. Subsequently, the void created by the deletion is filled using a tetrahedral fan centred at the most recently added point.

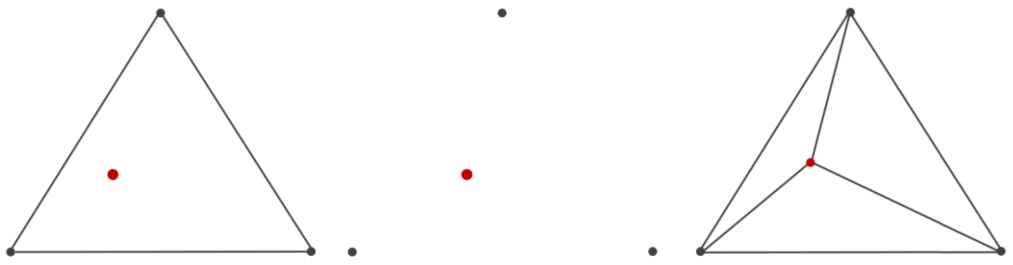


Fig. 3.9. Deletion and creation of tetrahedron

Despite the efficacy of this approach, it may generate an excessive number of tetrahedra, necessitating the removal of any tetrahedron lying outside the original mesh, to eliminate these tetrahedra situated outside the intended mesh, a ray is generated from each point, and its orientation relative to the normal of the closest intersecting face is assessed to determine deletion eligibility.

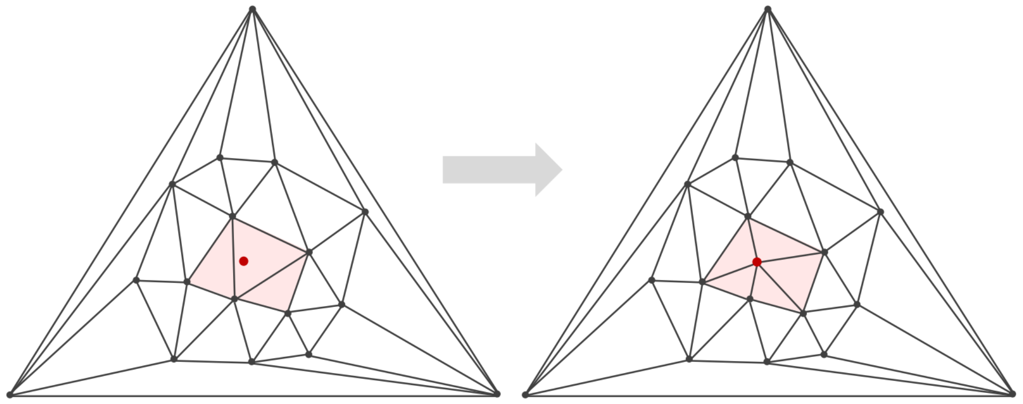


Fig. 3.10. Correction of tetrahedron

Detecting violating tetrahedra during the addition of a new point involves creating a ray from the centre of a randomly selected or last tetrahedron towards the new point. Utilizing this ray, the intersected face is identified, facilitating traversal to the adjacent tetrahedron.

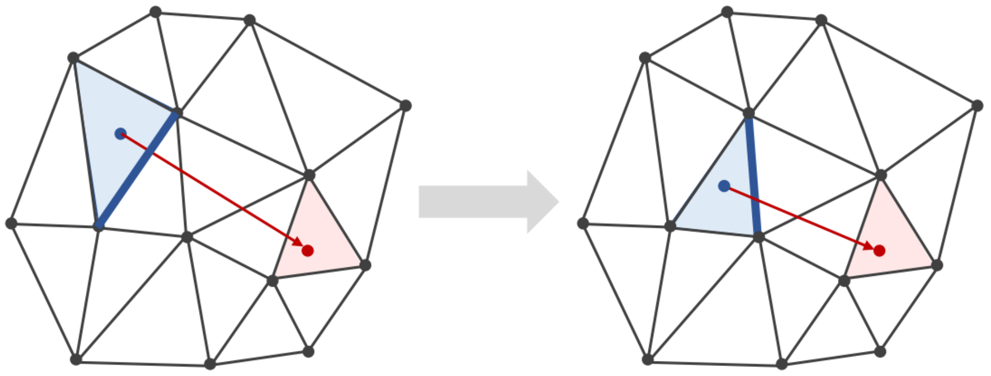


Fig. 3.11. Detection of violating tetrahedron

This iterative process is repeated for each point. Upon completing the tetrahedralization, identifying and removing all violating tetrahedra is imperative, achieved by verifying their compliance with the Delaunay condition.

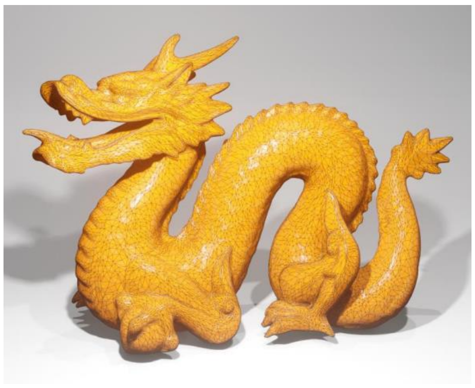


Fig. 3.12. Triangulated dragon mesh

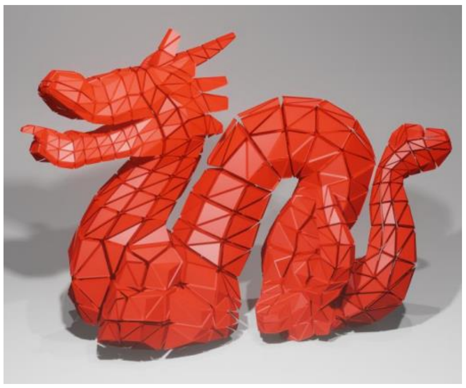


Fig. 3.13. Tetrahedralized dragon mesh

3.3.3 Physical Constraints

Upon finalizing the mesh, completing the primary loop of the simulation assumed paramount importance, with constraint resolution emerging as the central focus at this point. Two constraints were employed to ensure the proper functioning of the simulation: volume and edge constraints (Macklin, Müller and Chentanez, 2016) (Müller et al., 2020). Each constraint incorporated a distinct compliance value, dictating the rigidity or softness of the body by adjusting the stringency of constraint solving. The initial step involved resolving the edge constraints.

$$\lambda = \frac{-C}{w_1|\nabla C_1|^2 + w_2|\nabla C_2|^2 + \cdots + w_n|\nabla C_n|^2 + \frac{\alpha}{\Delta t^2}}$$

Compute correction for point $\mathbf{x}_i$ as: $\Delta \mathbf{x}_i = \lambda w_i \nabla C_i$

Fig. 3.14. General constraint solver formula

Resolving a distance or edge constraint between two particles necessitated several parameters, including the position of both particles, inverse mass, their current and original distance from each other, and the gradient of the constraint function with respect to each particle. The constraint function equates to the original distance minus the current distance, which equals zero when the constraint is satisfied (Müller et al., 2020). With the edge adjustments completed, attention turns to calculating the volume constraint.

$$C = l - l_0$$

$$\nabla C_1 = \frac{\mathbf{x}_2 - \mathbf{x}_1}{|\mathbf{x}_2 - \mathbf{x}_1|}$$

$$\nabla C_2 = -\frac{\mathbf{x}_2 - \mathbf{x}_1}{|\mathbf{x}_2 - \mathbf{x}_1|}$$

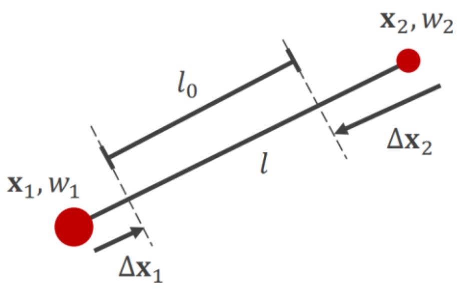


Fig. 3.15. Distance constraint formula

The volume constraint was computed as six times the current volume of the tetrahedron minus the original volume. The formula for tetrahedral volume is straightforward, contributing to the efficiency of this approach. Subsequently, the gradients of the constraint in relation to each particle must be determined (Müller et al., 2020).

$$C = 6(V - V_{\text{rest}})$$

$$V = \frac{1}{6}\left((\mathbf{x}_2 - \mathbf{x}_1) \times (\mathbf{x}_3 - \mathbf{x}_1)\right) \cdot (\mathbf{x}_4 - \mathbf{x}_1)$$

$$\nabla_1 C = (\mathbf{x}_4 - \mathbf{x}_2) \times (\mathbf{x}_3 - \mathbf{x}_2)$$
$$\nabla_2 C = (\mathbf{x}_3 - \mathbf{x}_1) \times (\mathbf{x}_4 - \mathbf{x}_1)$$
$$\nabla_3 C = (\mathbf{x}_4 - \mathbf{x}_1) \times (\mathbf{x}_2 - \mathbf{x}_1)$$
$$\nabla_4 C = (\mathbf{x}_2 - \mathbf{x}_1) \times (\mathbf{x}_3 - \mathbf{x}_1)$$

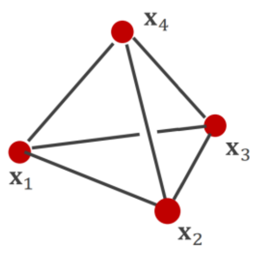


Fig. 3.16. Volume constraint formula

Notably, the original lengths, volumes, and inverse masses were calculated at the beginning of the simulation for both constraints.

3.3.4 Last Step

Concluding the process, the final step involved the post-solve implementation, which served to update each particle’s velocity and to ensure their proper positioning subsequent to the resolution of all constraints. This culminated in the creation of a complete XPBD simulation.

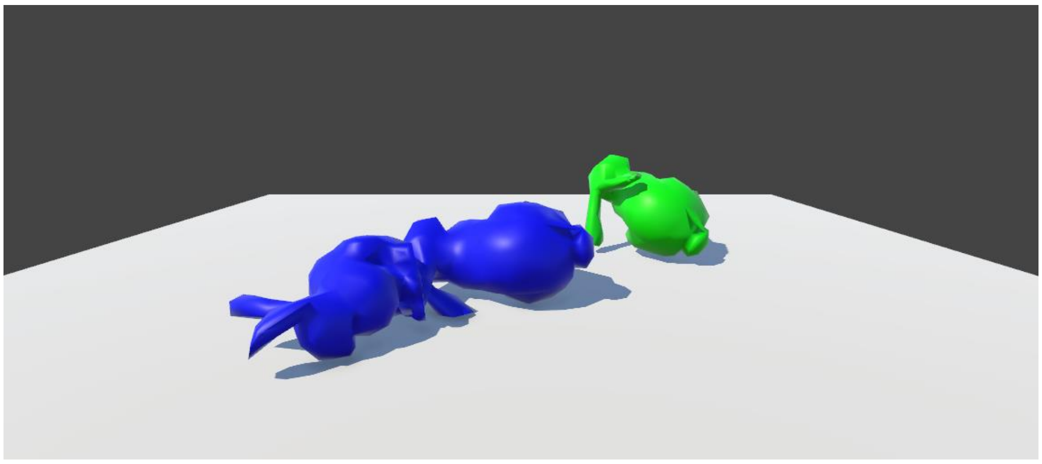


Fig. 3.17. Three Stanford Bunnies being simulated

3.4 Combining SDF and XPBD

Various methodologies exist for amalgamating these two techniques. This segment endeavours to investigate several viable strategies and highlight one that has previously been implemented in industry.

3.4.1 Approaches

SDF did not hinge on XPBD; however, XPBD relied on SDF, thus necessitating a method to adapt XPBD to work with SDF.

As mentioned earlier, XPBD mandates particles, typically represented by the vertices of the 3D mesh. Yet, for SDF, vertex concepts do not exist.

One strategy involved enveloping an SDF within a 3D mesh, wherein each SDF object retains data informing the algorithm of its primitive nature. Employing this method allows XPBD to operate with minimal adjustments, and with the integration of a tetrahedralizer, it can facilitate highly precise simulations. However, this approach somewhat negated the primary purpose of employing SDF for rendering.

An alternative approach entailed treating each pixel of a given SDF shape as a particle for simulation with XPBD. Although seemingly straightforward, this approach raised concerns regarding scalability and performance, indicating that it may be less optimal than the former approach.

3.4.2 Claybook

Claybook, a video game developed by Second Order and released in 2018, exemplifies the fusion of SDF, raymarching, and PBD, mirroring the objectives of this project. Although Claybook employs PBD instead of XPBD due to XPBD not being developed yet, its implementation closely parallels the project's framework. Conversely, its SDF strategy is notably intricate, emphasizing optimization to accommodate console limitations.

Fig. 3.18. Claybook promotional art

To incorporate PBD into SDFs, Claybook first generates the SDF shape, attaining a mathematical portrayal of the object's form, subsequently transformed into a point cloud. Point clouds consist of arrays of 3D points representing the object's surface. These discrete points within the cloud serve as particles for PBD, enabling force application. Given their direct correlation with the SDF, these particles directly induce mesh deformation. While Claybook diverges from this project in certain aspects, its overarching methodology aligns closely with it.

3.5 Testing

The experiments were designed to capture the performance characteristics of SDF, triangulated meshes, and XPBD under different conditions. To achieve this, a controlled experimental setup where the number of objects in the game world is systematically increased was created. Three types of objects were evaluated: SDF objects, triangulated meshes, and XPBD objects.

For SDF and triangulated meshes, two test scenarios were considered: with and without physics simulation, using Unity's built-in physics solution. This allowed for the evaluation of the impact of physics calculations on rendering performance. The experiments were conducted using a range of object counts, starting from a minimal number and consistently increasing to stress-test the rendering and simulation capabilities of each technique.

The following metrics were used to assess the performance of SDF objects, triangulated meshes, and XPBD objects:

Average Frame Per Second (FPS): This metric provided insights into the rendering performance of each technique. A higher FPS indicates smoother and more responsive gameplay experiences.

CPU and GPU Time Spent on a Single Frame: By measuring the time spent by the CPU and GPU to render a single frame, potential bottlenecks and resource utilization patterns could be identified.

Main and Render Thread Activity: Tracking the time spent by the main and render threads on computation versus idling provided insights into the efficiency of multithreaded processing and resource allocation. This metric is not relevant for XPBD; therefore, it was not collected for their experiments.

The experimental procedure involved systematically increasing the number of objects in the game world while recording the aforementioned metrics.

Each test scenario was repeated multiple times to ensure consistency and reliability of results. For XPBD objects, only the average FPS were measured.

Upon completion of the experiments, the collected data was analysed to identify trends, patterns, and correlations between the number of objects and performance metrics for each technique.

Chapter 4 Results

This section will highlight the data acquired from systematically evaluating the techniques implemented in the project within a controlled environment, aiming to mitigate overhead and inconsistencies. Tests were conducted in batches of five, with each batch yielding an average result. The primary variable under scrutiny is the quantity of objects within the scene, intending to assess performance scalability with increasing object counts.

The tests encompassed both SDF objects and conventional triangulated meshes, represented uniformly as spheres for consistency. Evaluation focused on rendering and physics aspects. Regarding rendering, alongside average frames per second, metrics included the duration spent on rendering and idling on both the main and render threads, the collective time taken by the CPU and GPU to generate a frame was captured.

Both SDF and triangulated meshes underwent examination utilizing Unity’s native physics solution, while a smaller quantity of XPBD objects, Stanford bunnies, were introduced for comparative analysis. The metrics recorded for physics evaluation were the average frames per second and the time taken by the CPU and GPU to create a frame. This dataset aimed to display the performance degradation attributable to the increasing object count.

While all tests were executed under uniform conditions, potential inconsistencies may arise due to concurrent background processes on the device used for data collection.

4.1 Thread Performance

In the evaluation of thread performance, the focus was on measuring the activity of the main and render threads during the rendering process. The results revealed significant differences in thread utilization across the three techniques.

For SDF objects, both the main and render threads exhibited increased levels of activity and idling throughout the rendering process, particularly on the Main Thread.

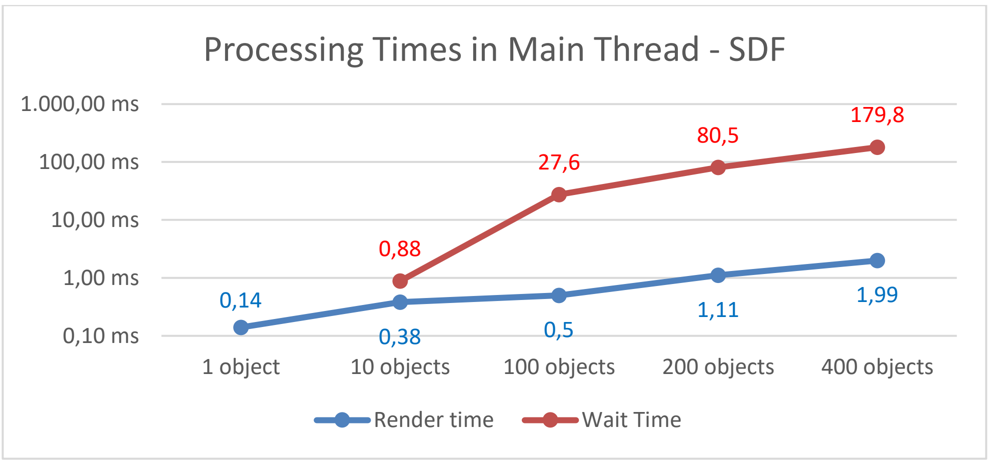


Fig. 4.1. Graph of SDF processing times in Main Thread

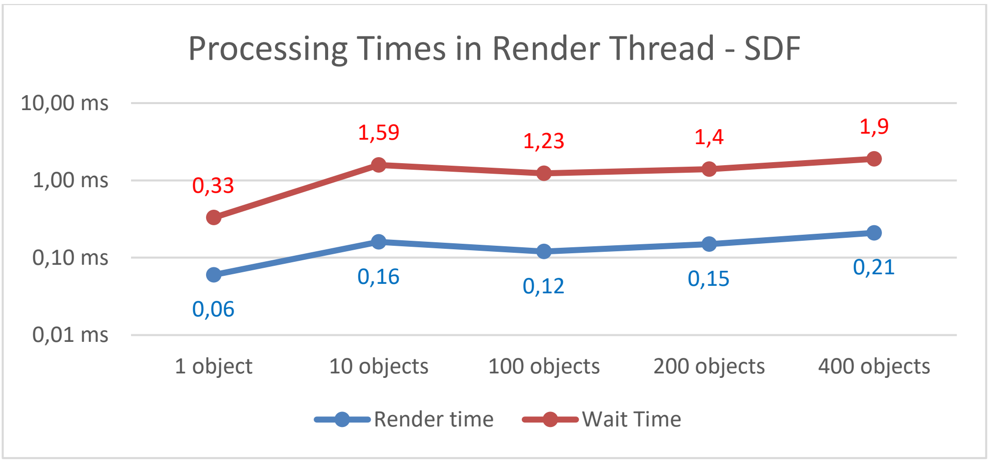


Fig. 4.2. Graph of SDF processing times in Render Thread

In contrast, triangulated meshes showed more balanced thread utilization, with the main and render threads sharing the computational load more evenly and minimising idle times.

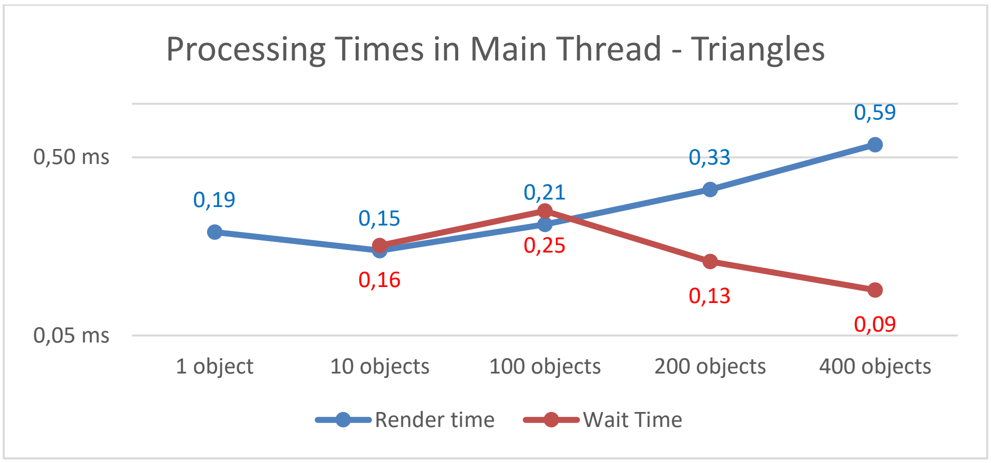


Fig. 4.3. Graph of Triangulated meshes processing times in Main Thread

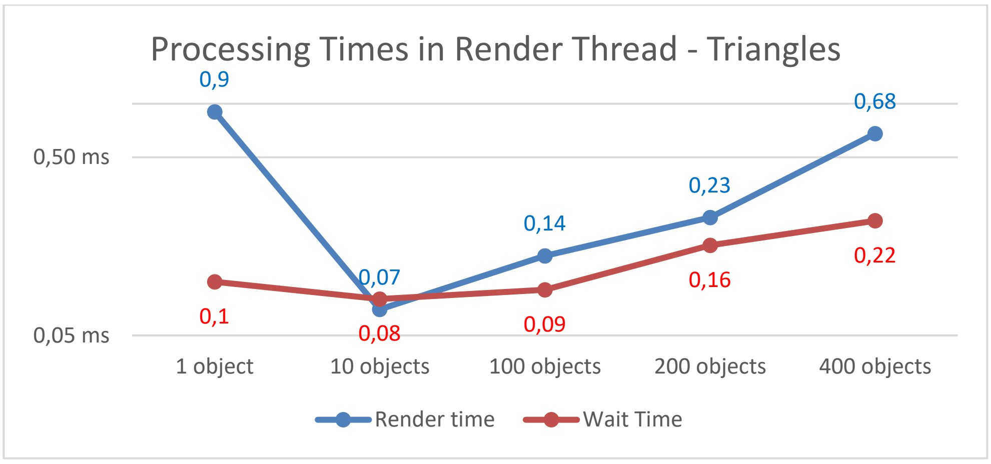


Fig. 4.4. Graph of Triangulated meshes processing times in Render Thread

4.2 CPU and GPU Frame Generation

The analysis of CPU and GPU frame generation times provided insights into the computational overhead associated with rendering each type of object. The results revealed notable differences in the time spent by the CPU and GPU to generate a single frame.

For SDF objects, both the CPU and GPU exhibited prolonged frame generation times, indicating a high computational workload as the number of objects in the scene increased.

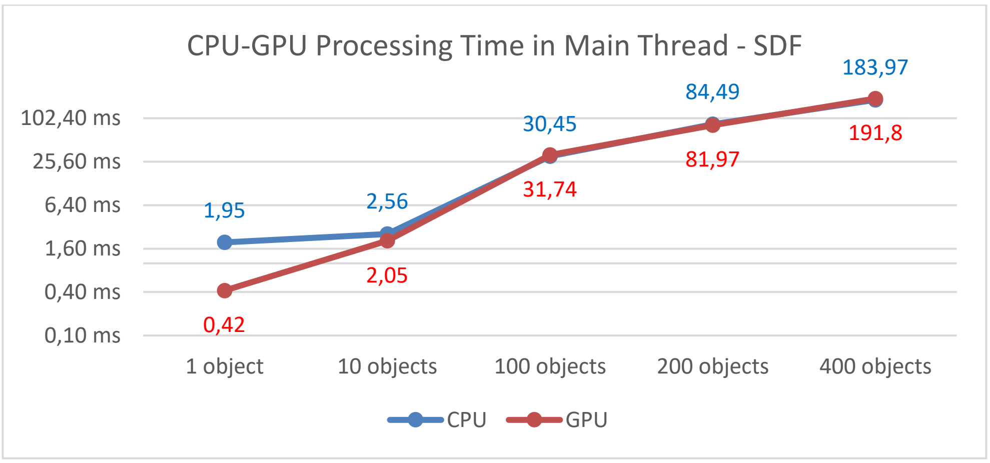


Fig. 4.5. Graph of SDF CPU and GPU processing times in Main Thread

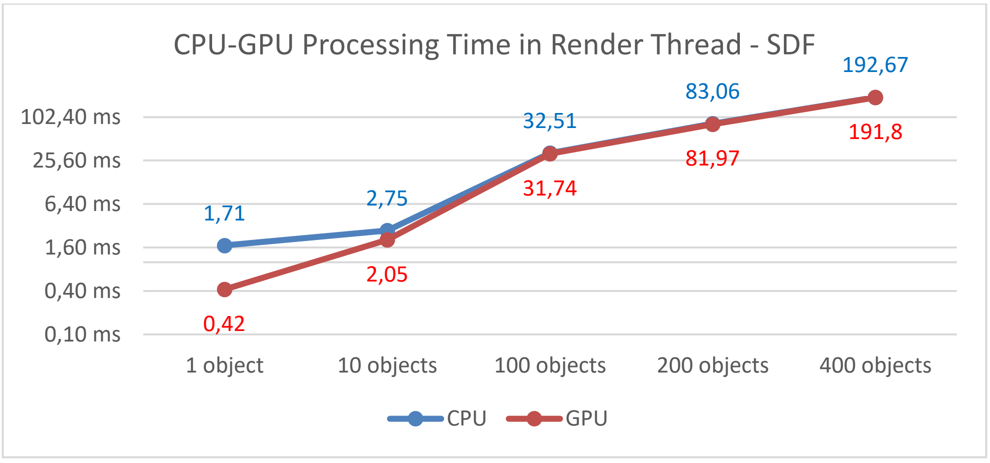


Fig. 4.6. Graph of SDF CPU and GPU processing times in Render Thread

Triangulated meshes showed more efficient frame generation times, with both the CPU and GPU completing rendering tasks more quickly and consistently compared to SDF objects.

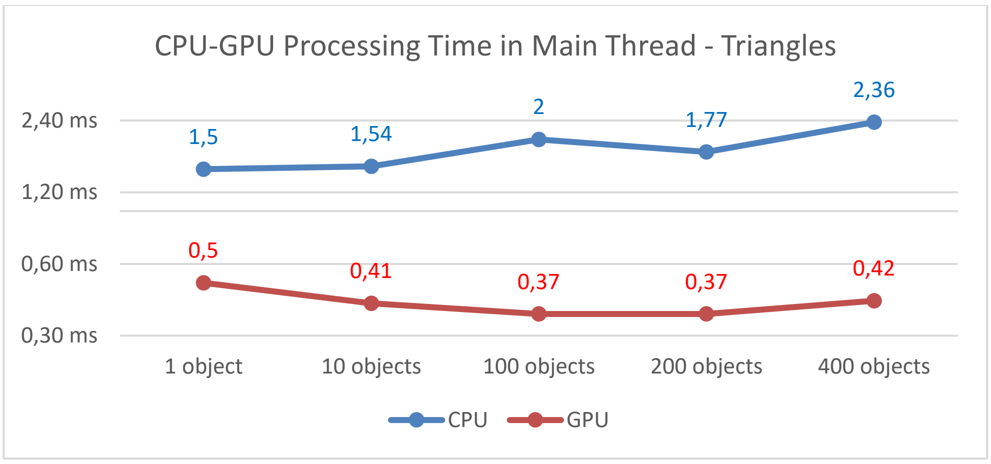


Fig. 4.7. Graph of Triangulated meshes CPU and GPU processing times in Main Thread

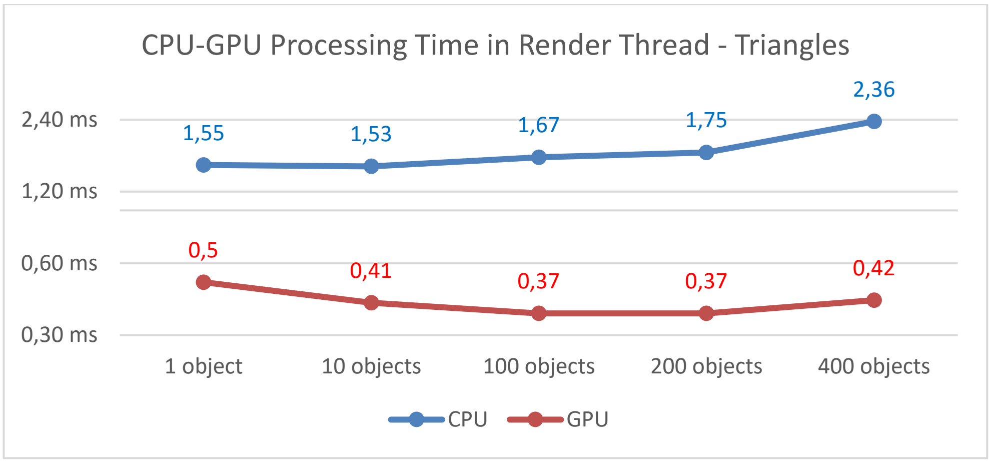


Fig. 4.8. Graph of Triangulated meshes CPU and GPU processing times in Render Thread

XPBD objects, being primarily focused on physics simulation, showed extremely high CPU frame generation times due to the computational complexity of physics calculations. However, GPU frame generation times were minimal.

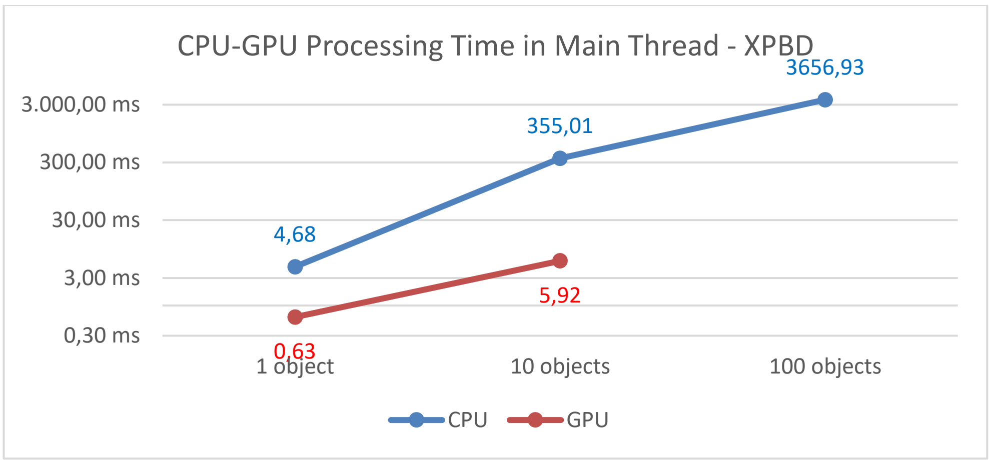


Fig. 4.9. Graph of XPBD objects CPU and GPU processing times in Main Thread

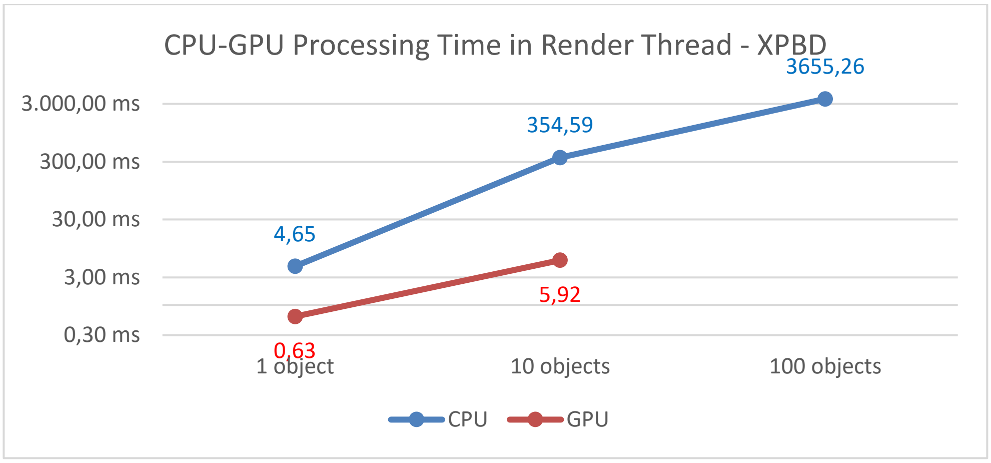


Fig. 4.10. Graph of XPBD objects CPU and GPU processing times in Render Thread

4.3 Frames per Second

The evaluation of FPS provided insights into the real-time rendering performance of each technique. The results revealed varying levels of rendering performance across different scenarios.

For SDF objects, the FPS progressively got lower, as the number of objects in the scene increased.

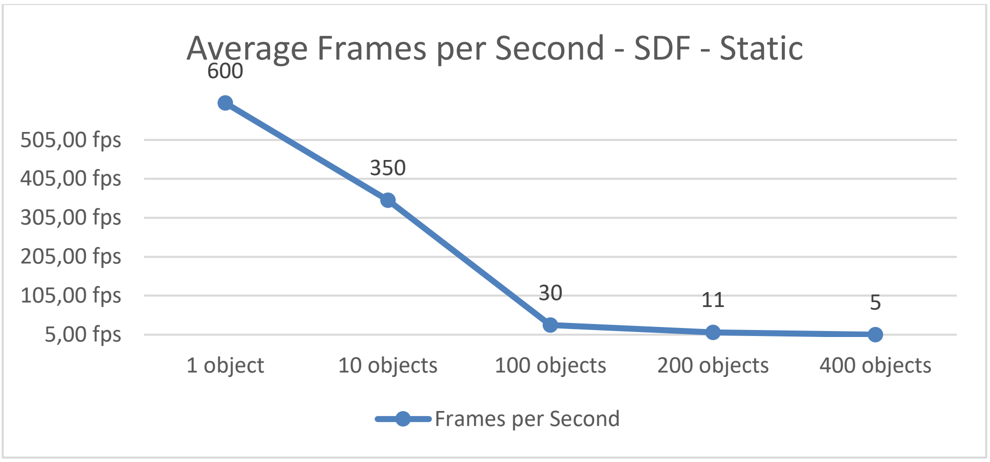


Fig. 4.11. Graph of SDF average frames per second without physics

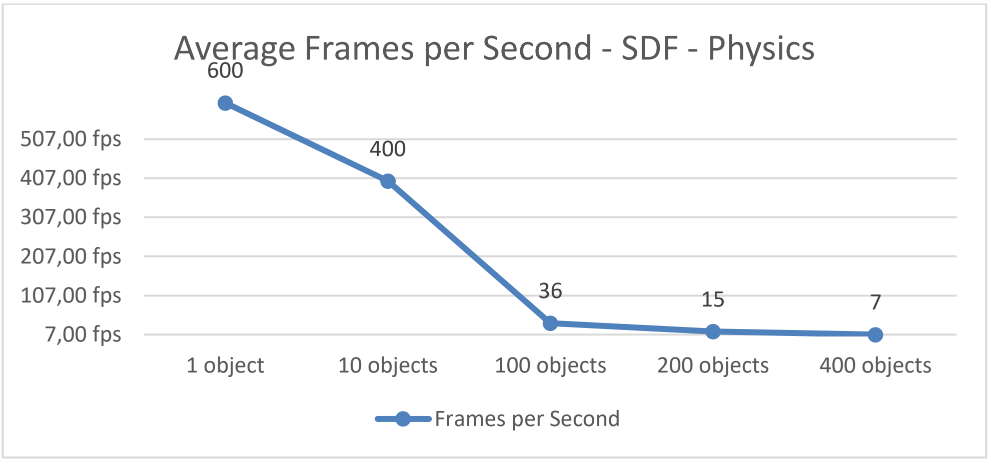


Fig. 4.12. Graph of SDF average frames per second with physics

Triangulated meshes exhibited vastly higher FPS compared to SDF objects, indicating better real-time rendering performance. Nevertheless, FPS decreased slightly as the number of objects in the scene increased.

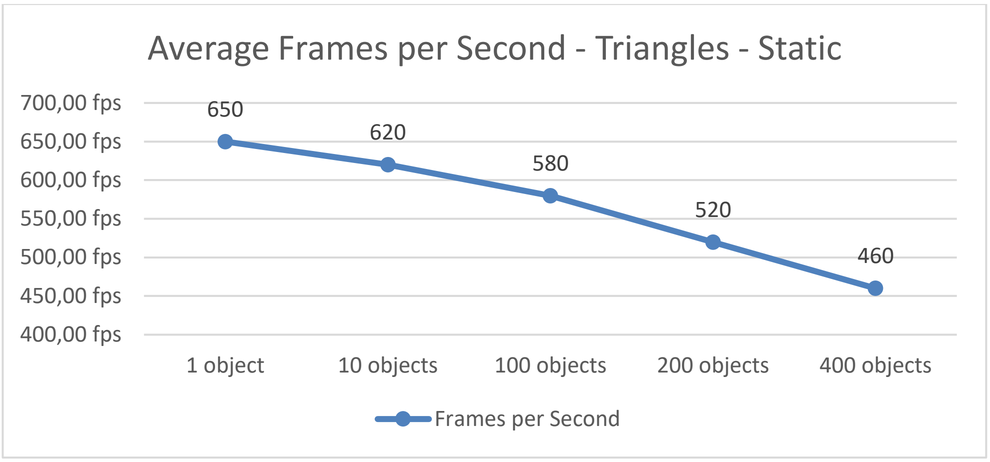


Fig. 4.13. Graph of Triangulated meshes average frames per second without physics

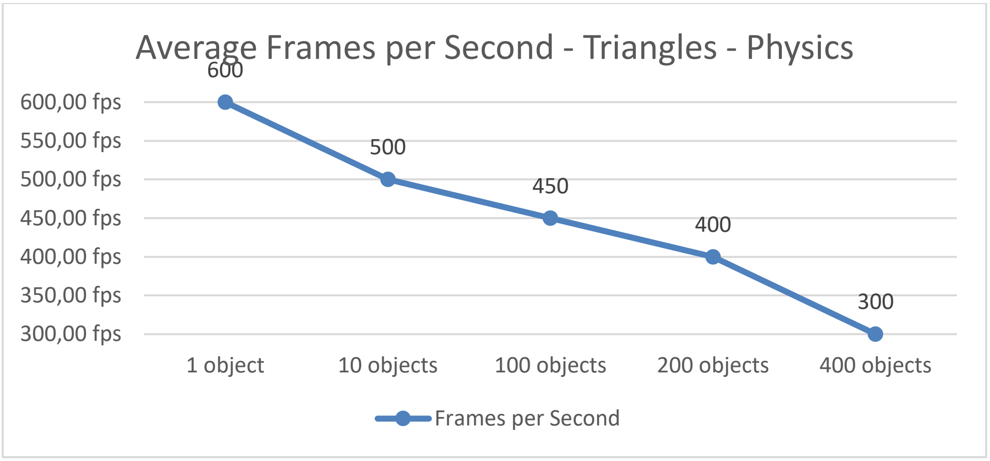


Fig. 4.14. Graph of Triangulated meshes average frames per second with physics

XPBD objects showed incredibly poor FPS, with performance largely dependent on the complexity of physics simulations. As the number of simulated objects increased, FPS decreased drastically.

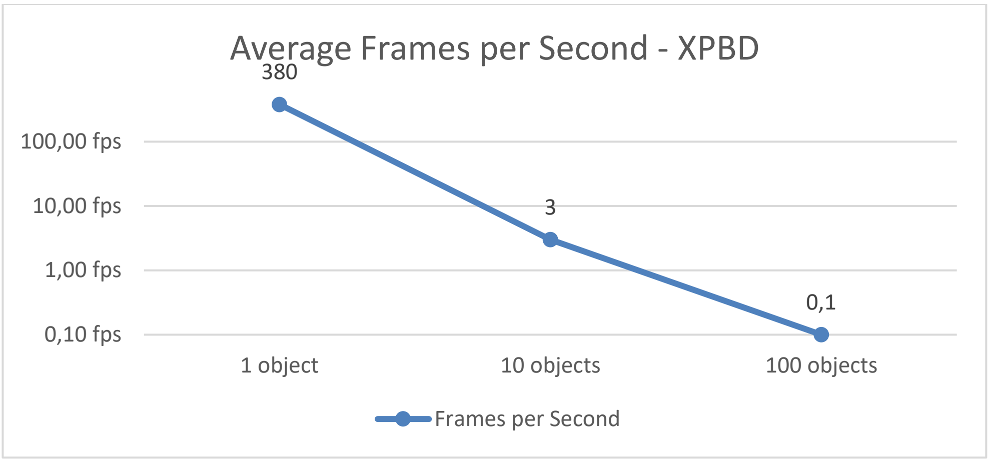


Fig. 4.15. Graph of XPBD objects average frames per second

Chapter 5 Discussion

The Discussion chapter serves as a platform to delve into the implications, applications, and insights garnered from the implementation of SDF and XPBD in video game environments. By examining the performance, limitations, and potential enhancements of these techniques, valuable insights into their practical applications and future developments can be gained.

In this chapter, the practical implications of SDF and XPBD beyond the realm of video games, considering their potential in diverse fields such as computer graphics, engineering, and medicine are explored. By analysing the strengths and weaknesses of each technique, opportunities for further research and development, as well as potential challenges that may arise in their implementation can be identified.

5.1 Performance Analysis

In light of the outcomes derived from testing triangulated meshes as the baseline, both SDF and XPBD techniques fell short of achieving comparable performance levels.

The implementation of SDF, albeit employing optimizations such as compute shaders to harness the GPU's computational prowess, demonstrated subpar performance. Notably, a discernible decline in average frames per second occurred when rendering just over a hundred objects, a modest count by modern gaming standards. More revealing are the metrics gauging CPU and GPU rendering times, both of which exhibited considerable durations. The CPU was heavily taxed with buffer preparation for each SDF object, while the GPU grappled with recalculating pixel values multiple times due to object coordinate overlaps. Such inefficiency placed this SDF implementation below the threshold of viability for practical game development applications.

Similarly, XPBD exhibited concerning performance metrics, notably imposing a significant burden on the CPU, as evidenced by the meagre three FPS achieved when simulating a mere ten objects. Although the XPBD simulation functioned as intended, its performance failed to meet the desired standards.

In stark contrast, triangulated meshes showcased robust performance, maintaining efficient frame generation times and consistently high frame rates.

5.2 Realism and Visual Quality

In terms of their overarching objective to enhance both visual quality and realism in games, both SDF and XPBD excelled at it.

SDF offered precise control over rendered output, facilitating the creation of visually striking effects with unique characteristics. By incorporating standard lighting and shadow calculations common in rasterization rendering pipelines, SDF achieved comparable visual fidelity, while also mitigating one of rasterization's primary weaknesses: aliasing. While SDF is typically employed to craft novel environments and effects, it possesses the capacity to attain a significant degree of realism.

XPBD demonstrated remarkable accuracy in simulating soft bodies and boasts extensive customization capabilities, enabling the realistic simulation of objects with diverse physical attributes. However, the technique's realism may have exceeded the requirements of typical games, as real-time realism of this calibre could often be simulated through animations. This suggested that while XPBD offers unparalleled realism, it may not always align with the practical needs of game development, where realism is often achieved through alternative means such as handcrafted animations or pre-simulated behaviours.

5.3 Implementation Challenges and Solutions

The use of SDF drew upon a rich history of development spanning decades, offering a wealth of literature and prior works for reference. This extensive body of knowledge significantly eased the implementation process, with numerous online resources available to address usual challenges. However, the endeavour faced complexities when attempting to innovate beyond established practices, particularly in integrating advanced SDF properties such as rotation, symmetry, and displacement. While these properties offer expanded creative possibilities, they were deemed non-essential for evaluating SDF’s viability in game rendering, though they undoubtedly enhance the potential for creating intricate objects and environments.

In contrast, XPBD represented a relatively newer technique, resulting in more limited available resources despite the dedicated community support and instructional materials provided by original authors like Matthias Müller.

Implementing XPBD proved to be a formidable task, given the method’s intricate nature and multifaceted applications. While the basic PBD algorithm and collision calculations were relatively straightforward to implement, complexities emerged when navigating the various implementation paths. Opting for tetrahedralized meshes offered a more advanced approach yielding realistic results but presented unforeseen challenges in implementation.

Developing XPBD to function with tetrahedralized meshes proved to be demanding yet feasible. However, difficulties arose in the conversion of triangulated meshes into tetrahedralized ones, requiring sophisticated mathematical solutions, for this reason, development focused once more on conventional meshes.

Ultimately, constraints resolution, a critical component of the PBD algorithm, required significant adaptation to accommodate conventional meshes. Attempts to apply constraints designed for tetrahedrons, notably the volume constraint, severely impacted simulation performance. With limited development time and persistent performance issues, the implementation of XPBD was restricted to simulating tetrahedralized meshes exclusively, necessitating complex data structures for input into the simulation.

5.4 Scalability and Optimisation

It was evident that neither technique is scalable in their current state, necessitating optimizations before widespread implementation could occur.

Conventional rasterization techniques benefit from various optimization strategies, with culling being among the most commonly employed methods. Implementing different culling techniques for SDF could significantly enhance GPU performance. However, the primary bottleneck on the CPU side appeared to stem from the continuous creation of buffers for each individual object. Notably, many of the values transmitted via these buffers, such as light position and colour, remain unchanged. Streamlining the packaging of these values into buffers sent to the GPU would alleviate some of the CPU's and GPU's processing burden.

5.5 Practical Applications

Considering the practical applications of SDFs and XPBD outside of video games, both techniques offer unique advantages and can be utilized in various fields beyond the realm of entertainment.

SDFs have a wide range of practical applications in fields such as computer-aided design (CAD), medical imaging, robotics, and augmented reality (AR).

In CAD, SDFs can be used to represent complex geometries with high precision, allowing designers to create detailed models of mechanical parts, architectural structures, and industrial equipment. The ability to manipulate SDFs in real-time enables interactive design processes and facilitates rapid prototyping.

In medical imaging, SDFs play a crucial role in image segmentation, registration, and reconstruction tasks. By representing anatomical structures as SDFs, medical professionals can analyse and visualize patient data more effectively, leading to improved diagnostic accuracy and treatment planning.

In robotics, SDFs can be used for path planning, obstacle avoidance, and collision detection in autonomous systems. By encoding the spatial information of the environment into SDFs, robots can navigate complex environments safely and efficiently, making them suitable for applications such as warehouse automation, surgical robotics, and unmanned aerial vehicles (UAVs).

XPBD also has diverse applications in various fields, including engineering, animation, and simulation.

In engineering, XPBD can be used for simulating the behaviour of deformable structures such as textiles, soft tissues, and elastomers. By accurately modelling the physical properties of materials using XPBD, engineers can simulate complex mechanical systems and predict their response to external forces, leading to more robust designs and optimized performance.

In animation, XPBD enables realistic character animation and physics-based simulation of cloth, hair, and fluids. By simulating the dynamics of soft bodies using XPBD, animators can create realistic animations with natural-looking movement and interactions, enhancing the visual quality and realism of animated films, video games, and virtual simulations.

In simulation, XPBD can be applied to simulate a wide range of phenomena, including fluid dynamics, granular materials, and elastic objects. By leveraging the computational efficiency and stability of XPBD, scientists and researchers can model complex systems and analyse their behaviour in real-time, enabling advancements in areas such as weather forecasting, material science, and biomechanics.

Overall, the practical applications of SDF and XPBD extend far beyond video games, offering valuable tools and techniques for a wide range of industries and disciplines. As research and development in these areas continue to advance, we can expect to see even more innovative applications and solutions emerge, further enhancing our ability to model, simulate, and interact with the world around us.

5.6 Industry Standards

When comparing the outcomes derived from SDF and XPBD against those of rasterization and force and impulse-based physics, the preference for the latter became evident. Despite rasterization nearing its limitations, it occupies a stable position, and altering the state of virtual environment rendering presents formidable challenges. GPUs are designed to support rasterization, advocating for the continued exploration of this technique’s capabilities. Rather than seeking a replacement for rasterization, enhancing its existing framework may offer a more pragmatic approach, given its solidified role in graphics processing.

Physics implementation remains a challenging yet gratifying aspect of video game development, where novel methodologies promising improved performance and accuracy garner attention. However, for new methods to gain traction, they must align closely, or at least minimally deviate, from established norms and workflows. Introducing solutions that diverge significantly from these conventions may inadvertently introduce complexities, dissuading developers from their adoption. The enduring appeal of established rendering and physics standards in video games stems from their maturity and extensive research, providing developers with reliable frameworks that, while not flawless, allow for iterative improvements to meet project objectives.

Chapter 6 Conclusion and Future Work

This dissertation has delved into the intricacies of raymarching, SDF, and XPBD, exploring their potential applications in video game development. While these techniques offer exciting possibilities for enhancing visual fidelity, realism, and interactivity, their current state presents significant challenges and limitations that must be addressed before widespread adoption in games can occur.

Integrating raymarching, SDF, and XPBD into existing game engines and frameworks presents unique challenges that require careful consideration. Future investigations could explore the best practices for integration, including techniques for efficiently transferring data between CPU and GPU, optimizing rendering and physics pipelines for real-time performance, and ensuring compatibility with other game systems and features.

The project has shed light on the potential of raymarching and SDF in creating visually stunning and artistically expressive environments. However, it has also revealed the computational inefficiencies and technical complexities inherent in these techniques. As such, the immediate implications suggest that while raymarching and SDF hold promise for future game development, they require further refinement and tool creation to be viable for real-time applications.

On the other hand, XPBD, while effective in physics simulations outside of games, faces significant hurdles when integrated into interactive environments. Its high computational demands and lack of scalability make it impractical for medium to large real-time simulations in its current state. Thus, it is suggested that XPBD should remain primarily in the domain of offline physics simulations until advancements are made to improve its performance and usability in interactive contexts.

Throughout development, several limitations and challenges have been identified. For SDF, the main obstacle lies in its computational overhead and lack of robust tools for creating and manipulating SDF objects. While raymarching allows for intricate visual effects, the process of generating and optimizing SDF data remains cumbersome and time-consuming. Similarly, for XPBD, the absence of a functional tetrahedralizer and universal constraint solver impedes its integration into real-time game environments.

To address these challenges and unlock the full potential of raymarching, SDF, and XPBD in game development, multiple enhancements and further development avenues can be pursued. For SDF, the addition of user-friendly tools for creating, editing, and saving SDF objects would streamline the workflow for artists and developers, making it more accessible for game development.

Meanwhile, XPBD can benefit from the development of a performant tetrahedralizer capable of generating tetrahedral meshes efficiently during runtime. Additionally, the implementation of a universal constraint solver tailored for XPBD would significantly improve its usability and flexibility in interactive environments, enabling more complex and realistic simulations.

While this project has primarily focused on the technical aspects of raymarching, SDF, and XPBD, further research could explore their potential for artistic expression and creative storytelling in games. Investigating how these techniques can be used to evoke emotion, convey narrative themes, and engage players on a deeper level could open up new avenues for innovation and experimentation in game design.

Notably, beyond video games, raymarching, SDF, and XPBD have potential applications in other interactive media. Future research could explore how these techniques can be adapted and extended to create immersive and interactive experiences across a range of platforms and contexts.

Moving forward, further investigations are warranted to address the limitations and challenges identified. Research efforts should focus on optimizing SDF for real-time applications, developing more efficient algorithms for tetrahedralization, and refining constraint solvers for XPBD.

While raymarching, SDF, and XPBD hold immense promise for revolutionizing game development, their full potential can only be realized through continued research, innovation, and collaboration. By addressing the challenges and limitations outlined, and by embracing new opportunities for enhancement and refinement, the future of game development holds boundless potential for immersive, engaging, and unforgettable experiences for players around the world.

List of References

4A Games (2019) *Metro Exodus* [Video game]. Deep Silver.
(4A Games, 2019)

Aaltonen, S. (2018) *‘GPU-based clay simulation and ray tracing tech in Claybook’*, [online] Available at: https://ubm twvideo01.s3.amazonaws.com/o1/vault/gdc2018/presentations/Aaltonen_ Sebastian_GPU_Based_Clay.pdf [Accessed 15 Oct. 2023].
(Aaltonen, 2018)

Atari (1972) *Pong* [Video game]. Atari.
(Atari, 1972)

Beaudry, P., Doms, M. and Lewis, E. (2010). ‘Should the Personal Computer Be Considered a Technological Revolution? Evidence from U.S. Metropolitan Areas’. *Journal of Political Economy*, 118(5), pp.988–1036. doi: 10.1086/658371.
(Beaudry, Doms and Lewis, 2010)

Bourg, D.M. (2004). ‘How physics is used in video games’. *Physics Education*, 39(5), pp.401–406. doi: 10.1088/0031-9120/39/5/002.
(Bourg, 2004)

CD Projekt Red (2020) *Cyberpunk 2077* [Video game]. CD Projekt.
(CD Projekt Red, 2020)

Core Design (1996) *Tomb Raider* [Video game]. Eidos Interactive.
(Core Design, 1993)

Crytek (2007) *Crysis* [Video game]. Electronic Arts.
(Crytek, 2007)

Dally, W.J., Keckler, S.W. and Kirk, D.B. (2021). ‘Evolution of the Graphics Processing Unit (GPU)’. *IEEE Micro*, [online] 41(6), pp.42–51. doi: 10.1109/MM.2021.3113475.
(Dally, Keckler and Kirk, 2021)

Lin, Y. and Liu, D. (2009). ‘Incremental delaunay algorithm based on signed volume’. *Journal of Computer Applications*, 29(2), pp.459–461. doi: 10.3724/sp.j.1087.2009.00459.
(Lin and Liu, 2009)

Fischer, T., Axel Böttcher, Coday, A. and Liebelt, H. (2010). ‘Defining and Measuring Performance Characteristics of Current Video Games’. *Lecture Notes in Computer Science*, pp.120–135. doi: 10.1007/978-3-642-12104-3_11.
(Fischer et al., 2010)

Geer, D. (2006) *‘Vendors Upgrade Their Physics Processing to Improve Gaming’.* doi: 10.1109/MC.2006.284.
(Geer, 2006)

Hart, J.C. (1996) ‘Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces’, *The Visual Computer*, 12(10), pp. 527–545. doi:
10.1007/s003710050084.
(Hart, 1996)

id Software (1992) *Wolfenstein 3D* [Video game]. Apogee Software, FormGen.
(id Software, 1992)

id Software (1993) *Doom* [Video game]. id Software.
(id Software, 1993)

Kojima Productions (2019) *Death Stranding* [Video game]. Sony Interactive Entertainment, 505 Games.
(Kojima Productions, 2019)

Lai, Y.-K. and Chung, Y.-C. (2015). *'An efficient and high quality rasterization algorithm and architecture in 3D graphics systems'*. doi: 10.1109/iscas.2015.7169202.
(Lai and Chung, 2015)

Lowood, H. (2009). 'Videogames in Computer Space: The Complex History of Pong'. *IEEE Annals of the History of Computing*, 31(3), pp.5–19. doi: 10.1109/mahc.2009.53.
(Lowood, 2009)

Macklin, M., Müller, M. and Chentanez, N. (2016) 'XPBD: Position-Based Simulation of Compliant Constrained Dynamics', *Proceedings of the 9th International Conference on Motion in Games*. doi: 10.1145/2994258.2994272.
(Macklin, Müller and Chentanez, 2016)

Microsoft (1995) *DirectX* [Computer program]. Available at: https://support.microsoft.com/en-au/topic/how-to-install-the-latest-version-of-directx-d1f5ffa5-dae2-246c-91b1-ee1e973ed8c2
(Microsoft, 1995)

Müller, M., Heidelberger, B., Hennix, M. and Ratcliff, J. (2007). 'Position based dynamics'. *Journal of Visual Communication and Image Representation*, 18(2), pp.109–118. doi: 10.1016/j.jvcir.2007.01.005.
(Müller et al., 2007)

Müller, M., Macklin, M., Chentanez, N., Jeschke, S. and Kim, T. (2020) 'Detailed Rigid Body Simulation with Extended Position Based Dynamics', *Computer Graphics Forum*, 39(8), pp. 101–112. doi: 10.1111/cgf.14105.
(Müller et al., 2020)

Nintendo EAD (1996) *Super Mario 64* [Video game]. Nintendo.
(Nintendo EAD, 1996)

Nintendo R&D1, Ikegami Tsushinki (1981) *Donkey Kong* [Video game]. Nintendo.
(Nintendo R&D1, 1981)

Nintendo R&D4(1981) *Super Mario Bros.* [Video game]. Nintendo.
(Nintendo R&D4, 1985)

Nvidia (2004) *PhysX* [Computer program]. Available at: https://developer.nvidia.com/physx-sdk
(Nvidia, 2004)

Osher, S. and Fedkiw, R. (2003). 'Signed Distance Functions'. *Applied mathematical sciences*, pp.17–22. doi: 10.1007/0-387-22746-6_2.
(Osher and Fedkiw, 2003)

Quilez, I. (no date). *Distance Functions*. [online]
iquilezles.org. Available at:
https://iquilezles.org/articles/distfunctions/
(Quilez, no date)

Müller, M. (no date). *Ten Minute Physics*. [online]
matthias-research.github.io/pages/. Available at:
https://matthias-research.github.io/pages/tenMinutePhysics/
(Müller, no date)

Rockstar North (2008) *Grand Theft Auto IV* [Video game]. Rockstar Games.
(Rockstar North, 2008)

Rockstar San Diego (2010) *Red Dead Redemption* [Video game]. Rockstar Games.
(Rockstar San Diego, 2010)

Silicon Graphics and Khronos Group (1992) *OpenGL* [Computer program]. Available at: https://registry.khronos.org/OpenGL-Refpages/
(Silicon Graphics and Khronos Group, 1992)

Taito (1978) *Space Invaders* [Video game]. Taito.
(Taito, 1978)

Telekinesys Research Limited (2000) *Havok* [Computer Program] Available at: https://www.havok.com/havok-physics/
(Telekinesys Research Limited, 2000)

Valve (2004) *Half-Life 2* [Video game]. Valve.
(Valve, 2004)

Valve (2020) *Half-Life: Alyx* [Video game]. Valve.
(Valve, 2020)

Watt, A. H. and Fabio Policarpo. (2000) *‘3D Games: Real-Time Rendering and Software Technology’*. api.semanticscholar.org/CorpusID:60868838
(Watt and Policarpo, 2000)