Physically Based Smoke Simulation and Behavior for Mixed Reality

Georgijs Sidorovs, B.Sc.

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science (Augmented and Virtual Reality)

Supervisor: Carol O'Sullivan

August 2018

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Georgijs Sidorovs,

August 29, 2018

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Georgijs Sidorovs,

August 29, 2018

Acknowledgments

I would like to thank Carol O'Sullivan for guiding me through the process of my research along with providing me invaluable help and information which helped me achieve my goals.

I would also like to thank my parents for supporting me throughout the year both financially and emotionally relieving stress strengthening my determination to succeed.

Finally I would like to thank all the other lecturers and course colleagues that provided assistance when a barrier was hit in my work along with Gregory Penrose for his help with deployment of the simulation onto the Microsoft HoloLens.

Georgijs Sidorovs,

University of Dublin, Trinity College
August 2018

Physically Based Smoke Simulation and Behavior for Mixed Reality

Georgijs Sidorovs, , Master of Science in Computer Science

University of Dublin, Trinity College, 2018

Supervisor: Carol O'Sullivan

The aim of this dissertation is to simulate physically accurate smoke behavior that can perform on mixed reality headsets like the Microsoft HoloLens. The final goal being a simulation that can interact with the surrounding real life environment along with other virtual objects that can potentially be located within the environment. This consists of multiple areas beginning with a physical simulation using an approximation algorithm combined with forces relevant to smoke fluid simulation. Combining this with the spatial mapping data from the HoloLens as a base environment It is possible to develop interactions between the particles within the simulation with the real world ultimately creating a mixed reality experience. However this isn't simply enough as the simulation must also operate smoothly on a mixed reality headset which takes into account it's restriction resulting in a focus on optimization and alteration to achieve the designated goals.

Video: https://youtu.be/u4X5CyMkxx8

Contents

List of Figures

Chapter 1

Introduction

This chapter will contain some necessary knowledge about the medium that is being used along with a brief reason as to why this research was done and how this dissertation is structured.

1.1 Mixed Reality

To better understand what the aim of this dissertation is, first we need to discuss Mixed Reality (MR) itself along with the Microsoft HoloLens which is the key platform for which this dissertation is intended.
Mixed reality is in a way a compromise between augmented and virtual reality taking aspects from both to create a unique state of limbo where it is both similar and different to both augmented and virtual reality. Firstly it does not rely directly on a full virtual world like virtual environment but rather takes the elements of of the real world and converts them to a virtual environment. Within the HoloLens this is referred to as spatial mapping. A process where the headsets sensors maps the real world to a virtual mesh. The second aspect is that it takes elements from Augmented Reality (AR) to enhance the experience. *Intel* release on the difference is an accurate explanation of augmented reality "Augmented reality keeps the real world central but enhances it with other digital details, layering new strata of perception, and supplementing your reality or environment."[9]. Where in augmented reality it is primary an overlay, in mixed reality to interact with both virtual and real elements in both a virtual and real

way. This can range from interacting with virtual objects using your real body to using real environments affect your virtual elements.

Within the last few years a number of devices have been released within all three realities for different purposes ranging from gaming to industry. The prominent including the Oculus Rift and HTC Vive for gaming along with Daqri for industrial use. Where the Microsoft HoloLens shines is that it's capabilities allow it perform in all spectrum of use. Currently the initial version of the microsoft hololens consists of the following specifications.

- 2GB of Random Access Memory (RAM)
- Custom Microsoft Holographic Processing Unit Hologram Processing Unit (HPU) 1.0
- Intel 32 bit architecture with TPM 2.0 support

As can be seen above the current version of the HoloLens is not particularly powerful with 2GB of RAM being much less than the current flagship mobile phones like the "IPhone X October 2017(3GB)", the "Samsung Galaxy S9+ June 2018(6GB)" and the "OnePlus 6 May 2018(8GB)". This is considerably more than the RAM provided within the HoloLens. Along with this it is difficult to obtain the exact comparison of the HoloLens compared to PC Central Processing Unit (CPU) architecture due to its unique architecture and functionality.

1.2 Motivation and Goals

Our motivation derives from the current state of the art technology that is being developed at a rapid pace. One of these is the aforementioned mixed reality that is becoming more prominent in the world of computer science. With the release of the Microsoft HoloLens an interest has been spiked in the area of mixed reality with Google Trends showing a rapid increase in search starting only a few months after its initial release on March 30th 2016. This means that the current computer science industry is expanding towards the area of AR, VR and mixed reality.Along with this we realized that particles especially ones based on physics simulation is something that should be

attempted and tested within the scope of mixed reality.

Figure 1.1: Google Trends search history for Mixed Reality showing a spike starting in 2017.

Currently the leading state of the art smoke simulation methodologies consist of accurate, complex and demanding implementations that in our opinion might not yet be suitable for such new area like mixed reality where the processing power of devices is still in its early stage. In both the works by Youquan Liu [28] and a more industry based NVIDIA released book GPU Gems 3 [19], the smoke simulation is highly depending on a new state of the art GPU for accuracy and quality as opposed to the efficiency which is needed for it's effectiveness on a mixed reality headset.

With this in mind I set our goals to achieve a balance of both high quality physical simulation of smoke along with a focus on efficiency that is required to perform such a task on the current generation mixed reality headsets. The final goals would be the following.

1. Simulate a physically based smoke particle system.
2. Implement interactivity between the virtual particle system with both the real and the virtual world.
3. Attempt a method for rendering each particle to represent smoke while also allowing for possible low particle counts.
4. Optimize the simulation within both the particle system and the rendering that will result in a smooth performance.

To clarify goal 2 mentions interactivity which should be elaborated on as a form of collision that allows the smoke particles to collide i.e. interact with the world that is mapped by the device. This also includes the ability to collide with not just spatial mapping generated meshes but rather virtual 3D meshes that have been placed inside of the environment. The last aspect mentioned is a smooth performance that is optimally desired as an outcome. The recommended Frames Per Second (FPS) from Microsoft documentation states that any application should meet the standard of 60 FPS [16], however due to the current generation of gaming consoles performing at 30 FPS, we decided that this should be the minimum performance outcome.

1.3 Outline

This dissertation follows the same pattern as the work flow used to develop the implementation. It is split up into 6 chapters with each chapter dealing with either a large area of the project or the results and conclusions of the outcome. Chapters 1 and 2 detail the introduction along with background research that was done both prior and during the implementation of the simulation. Chapter 3 deals with the approximation algorithm used to both calculate particle movement along with the forces necessary to simulate smoke behavior.Chapter 4 contains the interactive part of the simulation which details the implementation of collision along with spatial mapping that is obtained from the HoloLens sensors. Chapter 5 focuses on the rendering aspect, detailing the methodology used to display the particles along with their graphical behavior. Chapter 6 is the final chapter that details the results, conclusions and outcomes of the dissertation, highlighting the performance of the simulation along with providing discussion on whether the goals were reached.

Chapter 2

Background

2.1 Simulation Algorithms

In this section we discuss the prominent algorithms along with their methodologies for fluid simulation. The primary aim is to identify the most suitable algorithm for high performance smoke simulation i.e. even though a simulation could be used for fluid simulation it might not be the optimal algorithm to achieve the set goals for a *smoke* simulation.

2.1.1 Eulerian vs Lagrangian

According to an article by Zhang, Z. and Q. Chen [29] fluid simulation can be classified as either Eulerian or Lagrangian the former named after a Swiss scientist by the name of Leonhard Euler and the latter named after a French scientist named Joseph-Louis Lagrange.
Even though both simulation models are able to predict the movement of particles, they achieve this differently. The Eulerian method revolves around a continuum where the particles are part of a larger function that moves all the particles. On the other hand the Lagrangian method calculates the movement of each particle individually. The key thing to keep in mind is that according to the article, the Eulerian approached performed better on simulations where there was a steady state while the Lagrangian method outperformed the Eulerian in an unsteady state where the forces applied to the particles could potentially change.

2.1.2 Navier-Stokes Equations

The first algorithm we discuss is the *Navier-Stokes Equations*. This simulation model is classified as a Eulerian model. To get a better understanding of what makes this paper more or less suitable for our goals we will focus primarily on Jos Stams paper titled "Real-Time Fluid Dynamics for Games"[23]. Some of the crucial aspects that caught our eye was firstly the fact that this paper focuses primarily on fluid simulation for games rather than theoretical mathematics and the second being its focus on smoke simulation as an example of this. Below you can see the final results of his implementation of the Navier-Stokes Equations.

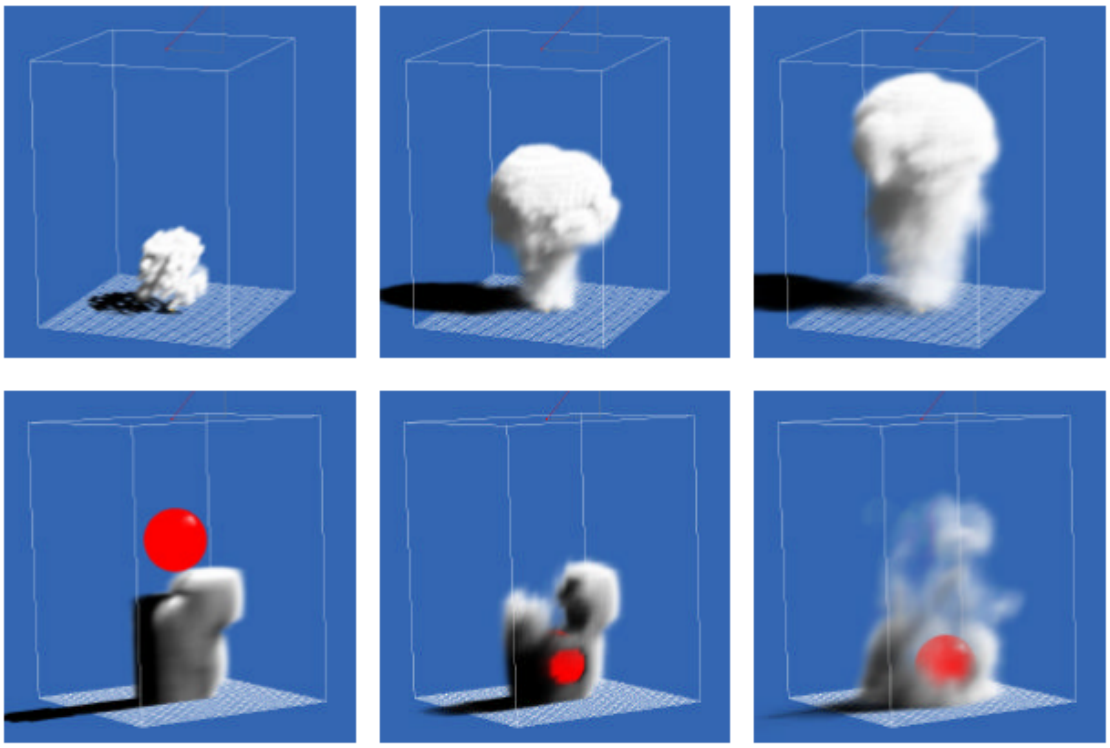

Figure 2.1: This is a collection of screenshots from the results of Jos Stams implementation of smoke simulation through the Navier-Stoke Equations.

The core concept of this algorithm is the idea of velocity vectors that are assigned to each point in space with the Navier-Stokes Equations being a method to calculate the evolution of the velocity vectors over time. On top of this using densities it is

straight forward to implement a shader to render the simulation based on a simple color combined with the assigned density.
The algorithm takes the following steps to perform the simulation.

1. Firstly and initial density is assigned to grid sizes which can either be coded or interactively assigned.

2. Calculate all relevant forces e.g. wind,temperature,pressure.

3. Diffuse the velocity vectors meaning that the density is exchanged between neighboring cells.

4. Finally move the density using advection equation.

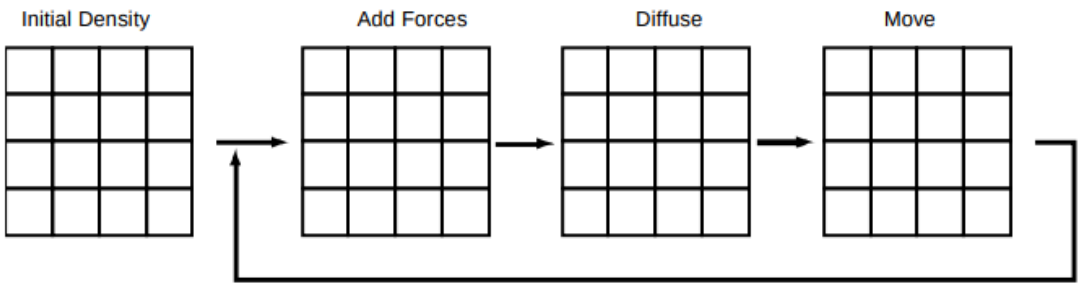

Figure 2.2: Summary of the previous steps used within the solver.

The one major flaw within this approach to achieve the goals assigned is the grid sizes that are necessary to achieve implement the solver. This would require to either implement static grid sizes with each grid size representing boundary for a room along with limiting the accurate collision response from the true room mesh. Along with this additional calculations would be needed to locate the real world center of the room due to the fact that the Unity implementation with HoloLens will place objects around you based on the (0,0,0) location of the grid size. One way to resolve this issue would be to access the devices sensors allowing for a dynamically changing grid size leading to recalculation of the velocity vectors each time the HoloLens updates the room data resulting in performance issues that the current version of the device cannot handle.

2.1.3 Smoothed Particle Hydrodynamics

Smoothed Particle Hydrodynamics (SPH) is a model for fluid simulation based on a number of papers by (Lucy, 1977) and (Gingold and Monaghan, 1977) however we will primarily be focusing on a paper presented during the Eurographics 2014 conference titled "SPH Fluids in Computer Graphics" [8] which focuses on the graphical implementation of this model. SPH is a Lagrangian model however it also has many similarities to the Eulerian approach like the use of the Lagrangian version of the Eulerian Navier-Stokes Equations along with also using grid sizes in many implementations.

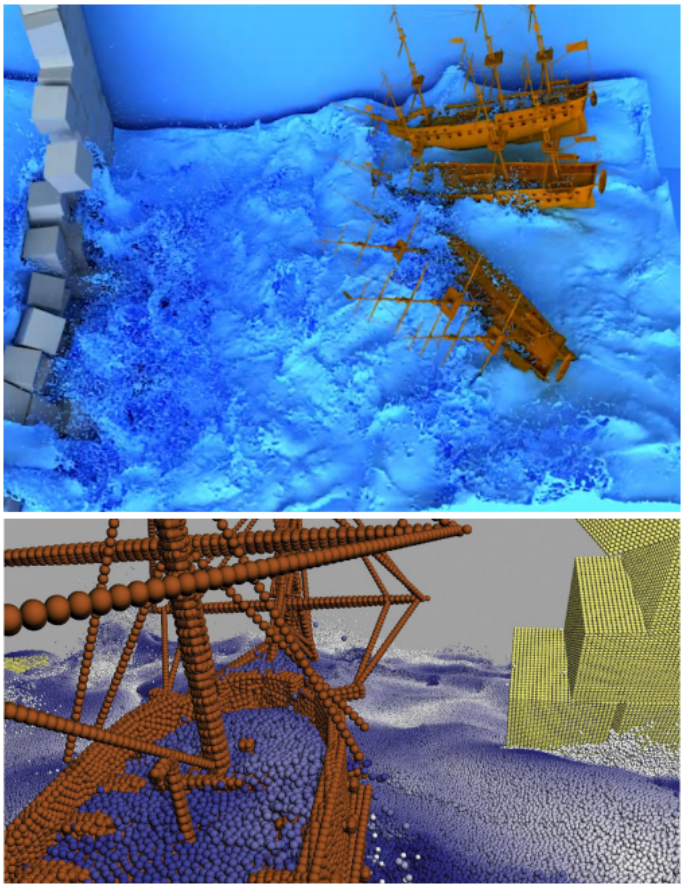

Figure 2.3: Results from the article demonstrating the graphical capabilities of SPH (20 million particles)

As mentioned previously due to the fact that this is a Lagrangian model, it calculates each particle individually while also approximating their positions and quantity

using a smoothing length due to the inability to compress particles. The steps used to calculate this algorithm is as follows for a simple implementation.

1. Find the nearest neighbors for all particles.
2. Compute the pressure for all particles.
3. Calculate forces for each particles e.g. pressure, viscosity.
4. Move the particles.

However this approach also has its flaws which can impact the performance of a HoloLens implementation. Primarily the use of neighbor search for particles adds extra strain on the computational ability of the HoloLens along with extra memory usage due to the implementation of grid sizes. Grid sizes within this model are created using $O(n)$ meaning that this would limit the maximum possible amount of particles that can be spawned if this was implemented compared to if it wasn't.

2.1.4 Lagrangian

The Lagrangian model for fluid simulation is very diverse due to the fact that it is very flexible in it's implementation with a simple version only requiring that the movement of all particles individually. As previously mentioned SPH is a Lagrangian model however on top of just the bare minimum it also contains many additional features that makes it an advanced version of the model. However this section will focus on a more simplistic method of fluid simulation outlined in the paper by Morgan McGuire titled "A Real-Time, Controllable Simulator for Plausible Smoke"[12].
This paper deals with a simplistic method of smoke implementation with a focus on it being plausible rather than visually accurate like the Navier-Stokes and SPH implementation. This allows for some level of accuracy while also providing the ability to customize the rendering aspect of the simulation. One major difference that makes this methodology stand out is that it does not take into account incompressible fluids but rather focuses on those that can be compressed like smoke. Along with this, the paper deals heavily into its interactive implementation focusing on the ability to adjust settings and collision with objects which is one of the key goals of this dissertation.
The simulation detailed is that of a basic movement approximation with forces that

create the effect of smoke. The following steps are the necessary elements required to achieve this.

1. Spawn Particles with attached variables e.g. position, mass, lifetime.
2. Calculate forces that will be applied to the particles.
3. Approximate movement using an approximation algorithm.

However even this methodology has flaws. The primary being that even though this model is very efficient it lacks in accuracy of movement and simulation. At the same time to simulate smoke the forces must be present. On the other side the efficiency of this paper allows the overstraining of the HoloLens by eliminating other elements that the two previous models included. Finally this model is completely independent of the area around it allowing for an easy introduction of this simulation into any environment both within a game and any real world environment in MR.

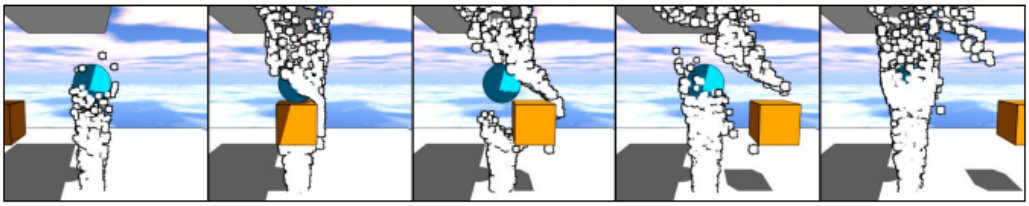

Figure 2.4: Results presented within the paper identifying both aesthetics and collision

2.2 Literature Review

Throughout our research we came across multiple methodologies and ways of simulating smoke these include all the aforementioned algorithms along with their pros and cons. Most of the papers researched focused primarily on accurate simulations based around mathematical equations that will allow for the most graphically and mathematically accurate results (Rørbech, Marinus [22], Zhang, Z and Chen, Q [29], Weaver, T and Xiao, Zhidong [26] ,and Bridson, Robert and Müller-Fischer, Matthias [3]) however due to the fact that these papers revolve around accuracy they were a good insight into fluid simulation, however were lacking the similarities to the goals we had in mind. The information from those was crucial to the understanding but studies like

(Stam, Jos [23],Yongzhe, X and Lee, Kyunjoo and Kim, Eunju and Ki, Jaesug and Lee, Byungsoo [28],Nguyen, Hubert [19],Ihmsen, Markus and Orthmann, Jens and Solenthaler, Barbara and Kolb, Andreas and Teschner, Matthias [8],McGuire, Morgan [12],Ren, Bo and Yan, Xiao and Yang, Tao and Li, Chen-feng and Lin, Ming C and Hu, Shi-min [21] ,and Xu, Yongzhe and Kim, Eunju and Lee, Byungsoo [27]) focused more on either an implementation for games or on improving performance which is a highly sought after characteristic for papers of interest.
The key paper to this dissertation is McGuire, Morgan [12] combining both a physically based fluid simulation while also taking into consideration limitations and focusing on optimization to create a hybrid of both accuracy and performance.
Although the basis of the work is based on that paper, it lacks in terms of graphical representation of particles which is one of the goals for this research. With that in mind a selection of papers were identified to have the necessary understanding of rending of smoke particles; primary Nguyen, Hubert [19] specifically chapter 30 within the book Crane, Keenan and Llamas, Ignacio and Tariq, Sarah [4] along with Larsson, Robert [10] thesis on "Interactive Real-Time Smoke Rendering".

2.3 State of the Art

2.3.1 FDS

Fire Dynamic Simulator (FDS) is a computational fluid dynamics (CFD) model of fire-driven fluid flow used for the sole purpose of modeling fire and smoke behavior within specific situations[20]. This application this model is primarily featured in is called Smokeview (SMV) and is developed and distributed by National Institute of Standards and Technology (NIST). Below is an example of the FDS model from (Yongzhe, X and Lee, Kyunjoo and Kim, Eunju and Ki, Jaesug and Lee, Byungsoo [28]).

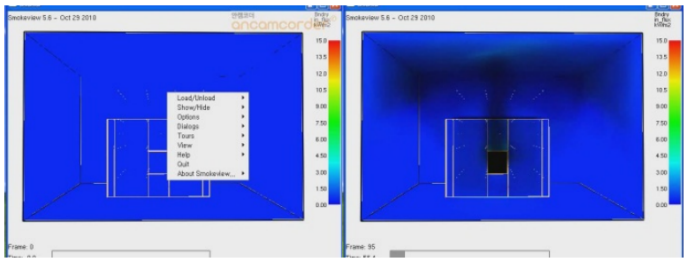

Figure 2.5: FDS model within Smokeview

Outside of just the Smokeview application, the FDS model has also been implemented in other papers[28]. Within this research FDS is the base of smoke simulation within Unity 3D for the use within gaming. The primary objective is the use of outputted excel files containing motion for the use with the Unity particle system. Although this is a very interesting approach to solving the issue of intense computation workload by using recorded situations to ease the strain on the game engine ultimately this would not be effective within the HoloLens due to the number of particles needed to even graphically represent the simulation. However FDS is still a modern smoke simulation model use for both firefighters and fire officers.

2.3.2 Game Industry

One of the key areas of smoke simulation is gaming. Smoke simulation allows for stunning visuals like that in the game *Watch Dogs* by the developer "Ubisoft Montreal" which was released in 2014. The game boasts some of the most advanced effects and graphics even to this day.

Figure 2.6: Smoke Particle System within Watch Dogs

The ability to achieve such visuals demands a powerful GPU and processor to

simply run this game. At the same time smaller projects like *Limbo* by "Playdead" also used smoke and fog to enhance the experience of their game by using them to create and atmosphere for the player to enjoy.

Figure 2.7: Athmospheric Fog and Smoke within Limbo

As can be seen above, the purpose of the effect is to create an atmosphere that adds to the aesthetic feel of the game. Unlike Watch Dogs however limbo is not a demanding game allowing for players using lower end machines to enjoy.
Both games use smoke to their advantage either to create stunning visuals or to create an eerie atmosphere but neither of this could be possible without the intensive work done by NVIDIA especially that Watch Dogs incorporates many of their features within their titles.
NVIDIA corporation is a company that works heavily with graphics and hardware. This comes with no surprise that smoke simulation is also something that is part of their work. Within their book GPU Gems 3[19] is chapter 30 dedicated to the rendering of 3D fluids along with smoke simulation.

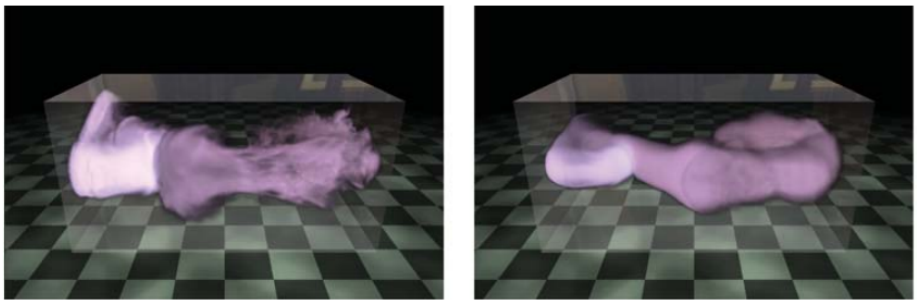

Figure 2.8: Semi Lagrangian Smoke Simulation within GPU Gems 3

These simulations are very detailed and visually intense relying heavily on the GPU to render them. With current generation CPUs and GPUs far outperforming those of previous years it is no wonder that NVIDIA are focusing into graphically demanding effects.

2.3.3 Mobile Games

Within the last few years the mobile gaming market rose due to the same advances in both computing and graphical performance however they are still not able to achieve the same effect that both consoles and computers can. This is quite similar to the HoloLens that currently as mentioned before is slightly behind the capabilities of hand held devices. Most mobile games avoid the use of particle systems and especially physically based simulations due to their limited capabilities. However through research an interesting article was discovered that focuses on those effects for mobile devices; "A Fire and Smoke Simulation for Mobile Games"[24]. This implementation uses Navier-Stokes Equations on a small scale with 2D grid sizes upto 128*128 with even the smallest 32*32 grid size resulting in a 25 FPS. This clearly highlights that even though this article was written in 2008, the goals that I set out would not be achievable due to the fact that a Navier-Stokes implementation for the HoloLens would require a 3D grid size along with a much larger scale.
Currently we were not able to identify any mobile games that have succeeded in an implementation of an adequate physically based fluid simulation with high performance. This led to the idea that even currently with the advances in hardware this is still a difficult to achieve methodology that has been avoided for a more simplistic implementation of animated sprites as an alternative.

Chapter 3

Design

In this chapter we will discuss each of the core chapters associated with the implementation stage of this dissertation. Each section within this chapter will either deal with one of the three key sections within the implementation or discuss the tools used to achieve this.

3.1 Simulation

Throughout the initial research many simulation models spiked our interest however only one approach was chosen for the implementation. This being a pure Lagrangian implementation using forces with an approximation algorithm to predict movement. The approximation algorithm used will be discussed within the chapter along with each of their pros and cons. The selection of the correct algorithm is crucial to the limitations of the HoloLens.
The second aspect of the simulation chapter will deal with the implementation of forces which are necessary to simulate smoke in a way that actually represents the source material. The following forces are outlined and implemented:

- Gravity
- Drag
- Buoyancy
- Vortex

- Wind

This chapter will finally also deal with spawning of particles and the control factors implemented to achieve variable scenarios.

3.2 Interactivity

The second implementation section deals with the interactivity between the simulation and both the real and virtual environments. This was achieved through multiple steps primarily consisting of the sensor data provided by the Microsoft HoloLens as a way of converting the real environment into a virtual one. With this data the chapter will describe how the spatial mapping data was used to achieve collision between the virtual and real along with other virtual objects that could be added into the scene. A detailed explanation will be included consisting of methodologies used for collision behavior like: response, avoidance ,and sliding.

3.3 Rendering

This chapter will deal with the graphical aspect of the simulation including different methodologies of graphically representing the smoke particles along with the strategy used to ultimately achieve the desired effect within this dissertation. The chapter will discuss smoke shaders, texturing, lighting ,and behavior associated with the rendering of a smoke simulation. This will help understand the reasoning behind the approach taken along with the graphical representation versus efficiency issue caused by the lack of powerful hardware. The result of this will determine what necessary steps and sacrifices must be taken to both improve the system while also attempting to maintain a high standard of graphical representation of particles.

3.4 Tools Used

The following tools were used to both implement and analyze this dissertation. Firstly the Microsoft HoloLens was decided to be the MR device to be used due to its capabilities of spatial mapping. The project was coded inside Unity 3D and Visual Studio

2017 using C Sharp as the language. The reason for this being the fact that Unity already contains the development tools needed due to Microsoft stating that the "fastest path to building a mixed reality app is with Unity"[15], along with a large collection of Microsoft written tutorials starting with "MR Basics 100"[17]. Finally the deployment and testing of the implementation was done through the use of Visual Studio for deployment along with both the HoloLens Emulator and HoloLens itself used for testing through their device system performance browsers.

Chapter 4

Simulation

4.1 Numerical Integration

This section will discuss the different algorithms that were discussed for the implementation of the position approximation algorithm. "The behavior of particles can typically be represented by one or more differential equations" as stated by Lyes, Timothy [11].As previously mentioned all of the algorithms are discussed with intent of use within a Lagrangian smoke simulation. They are all a function of time using acceleration and velocity to approximate movement in a three dimensional environment.

4.1.1 Euler's

Euler's Integration is one of the key methodologies for approximation along with being the algorithm that is used for the implementation of the dissertation. This method revolves around the integration of current position to get the future position called *Explicit Euler Integration* (Forward Euler Integration) or the opposite *Implicit Euler Integration* (Backward Euler Integration) which uses the derivatives of the future state to calculate the current state. However within this discussion I will focus on a hybrid Semi-Implicit Euler Integration to discuss it's potential for our implementation due to its more stable and accurate while more efficient implementation that was both discussed in (Dinev, Dimitar and Liu, Tiantian and Kavan, Ladislav [5]) and Lyes, Timothy [11]. The 4 basic variables needed for the implementation of this algorithm is position, velocity, acceleration and time. Initially only velocity and position are

required since time will tend to begin at 0 and acceleration is calculated by combining the forces acting upon the particle.
The base formula for Semi-Implicit Euler's Integration is as follows. This allows for the calculation of the velocity and position but one more aspect is the calculation of acceleration.

$$v_{n+1} = v_n + a_n \Delta t$$
$$p_{n+1} = p_n + v_{n+1} \Delta t$$

The main difference between the basic Explicit Euler Integration is that it utilizes future velocity instead of current velocity and at the same time utilizes current acceleration as to avoid computing future acceleration that would be needed by the pure Implicit Euler Integration.
To calculate acceleration the following elements are required. Firstly a combined vector of forces along with the inverse mass of the object. With all three formulas acquired: velocity, position, acceleration. A shorter version is implemented to combine both velocity and acceleration into one formula.

$$v_{n+1} = v_n + \sum \vec{f} \frac{1}{m} \Delta t$$

With the initial information acquired from the user or from the system the core loop begins for calculating the movement of particles. The following code is the basis for the core loop that calculates the position.

```
Data: Time step
Result: Particle position
for all live particles do
    update lifetime left;
    combine all forces;
    calculate velocity and acceleration;
    calculate position;
    set game object position to calculated position;
    if Sprite then
        adjust opacity;
        adjust scale;
    else
        Do nothing;
    end
end
```

Algorithm 1: Algorithm for Semi-Implicit Euler Integration

The code for this algorithm can be seen in section .1 of the appendix.

Along with the approximation algorithm, the same file stores all other behavior attributed to the particle since any particle behavior, movement, alterations are contained within that specific "Particle Handler". An exception to this is the death and initialization of certain aspects of the particle. The following are the aspects that are tightly connected to the numerical algorithm that is included within the file.

- Lifetime - The life that each particle has left before death. This property adjusts at each update.
- Forces - All forces calculations except for vortex are done within this file.
- Rendering - The graphical representation of the sprites are also adjusted within this file based on life time feature along with other factors mentioned in chapter 6.

4.1.2 Runge Kutta

The following two sub sections deal with two variations of Runge Kutta known as Runge Kutta 2nd Order (RK2) and Runge Kutta 4th Order (RK4). Both of these numerical integration were considered along side the chosen Euler Integration however did not fit the criteria or were simply inferior in achieving the goals set for this dissertation.

RK 4th Order

RK4 is an accurate and demanding alternative to Euler Integration. It produces high accuracy results at the cost of efficiency which is why it was not suitable for this implementation. Even though this is the case, it is still worth mentioning for future research to improve accuracy in further attempts to implement this dissertation.The general version of the algorithm is shown below.

$$
\begin{aligned}
k_1 &= f(y_n, t_n) \\
k_2 &= f(y_n + \frac{h}{2}k_1, t_n + \frac{h}{2}) \\
k_3 &= f(y_n + \frac{h}{2}k_2, t_n + \frac{h}{2}) \\
k_4 &= f(y_n + hk_3, t_n + h)
\end{aligned}
$$

RK4 relies on the use of a start point (k_1), two mid points (k_2,k_3),and an end point k_4 to calculate the final result[11].

$$y_{n+h} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)h$$

As can be seen the algorithm relies more so on the mid point values doubling their effect compared to the start end end k values. With all these extra calculations the possibility of improving efficiency compared to the chosen method is not possible as it requires both more memory to store the results along with more power to calculate them. This is why it is not a suitable algorithm for this implementation.

RK 2nd Order

RK2 is similar to both Euler Integration and RK4, in a way it is an in between to have both accuracy and performance. However even this algorithm was not as efficient as Euler.
The basis of this algorithm was taken from (Badis, Ydri [1]) where only the RK2 methodology from the Runge-Kutta family is discussed. This algorithm just like RK4 relies on k values however it does not include the start and end values but just the midpoint values. This gives us a formula which is a combination of both Euler and RK4.

$$
\begin{aligned}
k_1 &= \Delta x f(x_n, y_n) \\
k_2 &= \Delta x f(x_n + \frac{1}{2}\Delta x, y_n + \frac{1}{2}k_1) \\
y_{n+1} &= y_n + k_2.
\end{aligned}
$$

With this formula the algorithm approximates the mid point position and using a slope calculates the end position.[11] Just like RK4 this is a more accurate approximation algorithm and a variation of it was attempted during the initial phase (Leapfrog Integration), however the implementation caused issued and relied heavily on mass for the calculations that resulted in unnatural behavior due to the small mass of the particles. On top of this the extra calculations would still reduce the efficiency of the algorithm with a decrease in accuracy that would not be seen easily due to the compressible nature of the particles.

4.1.3 Time step

The time step is one of the crucial elements of numerical integration where a large time step would severely decrease accuracy while a low time step would decrease performance. The solution to this problem was to link the time step to the real time FPS statistics. This allowed for a synchronized movement between the amount of frames that were being displayed and the amount of calculations that would be done, allowing for a smooth and constant behavior irrelevant of FPS and performance drops.
With a constant 60 FPS the time step results in being 1/60th of a second which equals to 0.0166. This might not always be possible due to performance issues that cause

a drop in FPS, with a minimum satisfactory performance of 30 FPS the time step is increased to 0.0333. This delta time implementation allows for a link between the FPS shown and the suitable time step for the calculations.

4.2 Forces

Within this section we will explain the chosen forces to be included within the implementation of the simulation although only a base of 5 forces was implemented the system allows for an addition of more forces in the future if required. The forces are based on those within the key paper by McGuire, Morgan [12]. However one of the forces *"Turbulence"* was not implemented due to the fact that the simulation is never intended to reach high velocity where this would be required. The reasoning behind this is the flaw of collision that would begin experiencing issues where particles would seep through the mesh as a higher velocity was applied to the particles.

4.2.1 Gravity

Gravity is one of the key forces that all objects are affected by. Based on Newtons Law of Gravitation, the pull towards the earth is $9.81m/s^2$. This is implemented as a vector pointing in the negative y direction of magnitude 9.81. Resulting in a constant force vector of (0,-9.81,0). Within the implementation this is stored in a Vector3 data structure and never updated.

4.2.2 Buoyancy

Buoyancy is in a way a counteract to gravity that allows the particles to rise. This force assigns a vector of movement that all of the particles must follow. Within my implementation this force is controlled by only the y axis of a vector to adjust how fast or slow the particles rise. For particles to rise instantly a force greater than or equal to gravity must be applied as otherwise they would descend downwards. A recommended power of +(9.81-12.81) is recommended to create a smooth ascend without making it too fast. As mentioned previously, a high or low buoyancy setting would cause the particles to flow outside of the spatial mapped zone and through the virtual objects.

4.2.3 Wind

Wind is a very basic force that was applied to allow for more interactivity between the user and the simulation. A preassigned 3D vector is added to the total force that allows for wind to be added to the scene. The user is able to assign this and affect the strength of the wind as is desired. However a strong force would again cause the particles to phase through objects so caution is advised when adding wind.

4.2.4 Drag

In McGuire, Morgan [12] it is stated that "Drag simulates the presences of an incompressible medium by applying friction to oppose motion". With this in mind a formula was created based on the provided example however it was also adjusted to allow for a varied delta time clamping instead. This resulted in a formula that would multiple delta time instead of dividing which allowed for faster computation time.

$$Drag = m\vec{v}\Delta t;$$

When initially attempting the direct formula from the paper, an issue was detected due to the numerical integration where objects would increase in speed dramatically. Hence the previous made alteration was made to multiple delta time instead of dividing.

4.2.5 Vortex

Vortex is an interesting force that requires a much more elaborate implementation. Just like the normal simulation, particles are created following the same simulation behavior as normal ones, however their interactions differ from smoke particles. These particles are called vortices and upon collision with smoke particles create a vortex effect for normal particles around the vortex. The spawning of vortex particles is controlled within the same code as smoke particles however the particles are not part of the total limit due to the fact that they don not get visually rendered and are instead stored in a separate vector.

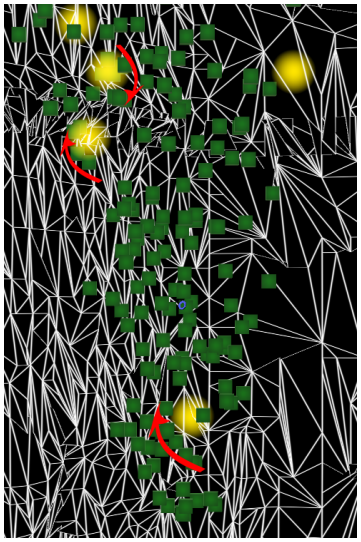

Figure 4.1: Movement of particle upon collision with a vortex

Compared to the formula created within the paper by McGuire, Morgan [12], the amplification of the force is static instead on varying based on distance from center. Our version of the force is described below with P representing particle position and V representing vortex position.

$$d = (P - V) / \|P - V\|$$
$$\vec{f} = (d(VortexRotation) * strength)$$

This formula allows for the same vortex effect upon collision as the original formula however it removes the computational need to calculate a varying strength based on distance and instead uses a static strength variable. The C Sharp code can be seen in Appendix 2.

4.3 Particle System & Spawning

Unity 3D comes built in with a particle system named *Shuriken* that can be used for the generation and alteration of particles. It supports different features and methods i.e. lifetime, shape, color and so on. Even though the system comes with all these components, accessing some particle data needed to achieve the desired goals is not

possible. With methods and variables locked to the common developer an alternative solution was implemented by coding a homebrew particle system. According to Techopedia the meaning of homebrew is " development of homebrew software is often for the purposes of expanding the function of the restricted hardware device"[25] which is most suitable to the restricted access provided by the Unity engine.
The same particle system is used both for the spawning and death of normal smoke particles along with vortex particles and is done within the same loop. The first section to discuss is spawning of particles. Smoke particles and vortex particles rely on different behavior to spawn new particles. The particles are spawned based on comparison to the maximum allowed particles. i.e. If the current number of particles is less than the maximum amount of particles allowed then each frame will spawn one particle. This particle can either be a smoke or vortex particle depending on the frequency of vortex particles set i.e. if 0.01 is set then only 1 in a hundred particles will be a vortex and the rest will be smoke particles. A second mode is also implemented that allows the initial spawning of a set number of particles that allows for a specific simulation or for testing purposes. This method spawns particles within a basic for loop. The code for this can be seen below:

```
Data: Mode Setting
Result: Particle Generation
for each frame do
    if Real Time then
        if Particle Count is less than maximum then
            randomize Life Time of particle between a min and max time;
            if Random number lower than spawn rate then
                Spawn vortex particle;
                Add particle to vector;
            else
                Spawn smoke particle;
                Add particle to vector;
                Increase particle count;
            end
        else
            Don't Spawn and particles;
            Check if any particles need to be destroyed;
            If smoke particle deleted then reduce particle count;
        end
    else
        Do nothing;
    end
end
```

Algorithm 2: Algorithm for generating smoke and vortex particles. The C Sharp code for this algorithm can also be viewed in Appendix .3 .

As can be seen the smoke particles are the only ones that affect the particle count while the vortex particles do not, however a decision was made that only if smoke particles can be generated should the vortex particles have a chance of spawning. The reasoning behind this is that if the smoke particles are not being spawn then there shouldn't be the spawning of forces that could be applied to those particles while at the same time due to the fact that vortex particles do not get rendered, they should not be part of the particle count since their impact on the performance is insignificant. Upon the initialization of the particles inside of the particle system, the life time, posi-

tion and rotation is passed into the constructor within the *Particle Handler* component. The constructor within the component uses the values from the particle system and adjusts them based on preferences inputed within Unity 3D. The constructor creates the following information about the particles (This is primarily associated with smoke particles rather than vortex particles).

- Spread Size : This controls the initial spread of particles away from the origin of the particle system.
- Mass : This is current set at 0.5 which is the inverse of 2. However this can be adjusted to preference.
- Initial Velocity : Randomized based on functions of spread values.
- Initial Scale : Records initial scale based on sprite.
- Initial Color : Records initial color based on sprite.

with these elements the particles are ready to be simulated. Finally during each iteration is performed to see if any particles current life is less than 0 resulting in their deconstruction and deletion from the vectors.

4.4 Optimization

Optimization is one of the key elements that is included within all the implementation chapters and is featured heavily within the simulation code. Optimization within the simulation primarily focuses on CPU performance and code clarity.
To optimize within this section code was checked to find obsolete and unused code. Along with this the Unity specified methodologies were implemented to make the necessary adjustments. Firstly all the primary loops were surrounded with if clauses to prevent unnecessary loops from being run on each iteration. Secondly any variables would be cached outside of the methods to reduce the memory usage and performance issues associated with declaration and creation of variables. The key places that this was implemented was the Vortex and Particle Handlers along with the Particle System code. Finally the last optimization technique implemented is to avoid certain operations that are more demanding than others. One of these is avoiding divide operators

and instead keeping a already inversed variable for future use e.g. inverse of mass. The second place where this was implemented was the distance formula within the vortex code. This distance code was used to calculate the intensity of the force based on distance between a vertex and smoke particle. This was eliminated as this formula would be calculated each time a vortex particle collides with a smoke particle, resulting in numerous calculations done during each loop.

Chapter 5

Interactivity

5.1 Spatial Mapping

Spatial mapping is one of the key elements that makes the Microsoft HoloLens the selected platform for this dissertation. Spatial mapping provides a virtual scan of a real world environment which developers can use to create applications[18]. The primary objects generated from the sensors is the *Spatial Surface Observer* and the *Spatial Surface*. Using both of these interactions can be implemented into the environment along with the implementation of caching.

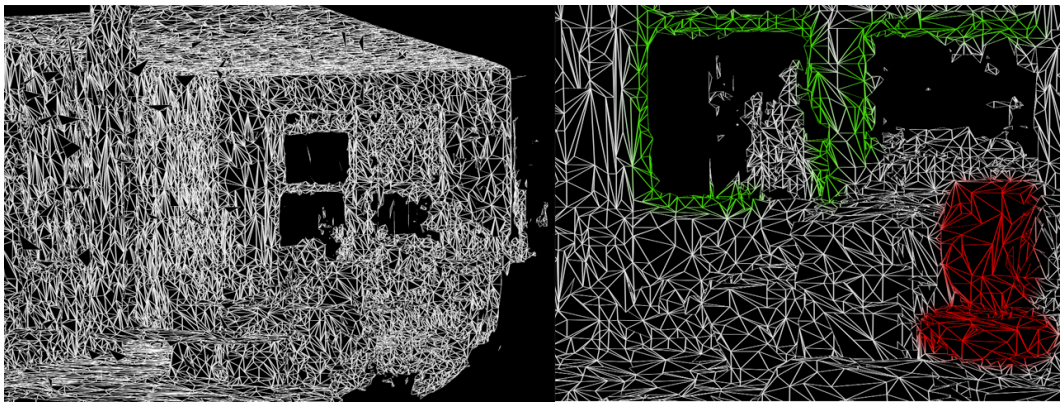

Figure 5.1: **Left:** An overview of a HoloLens generated mesh representing a real world room. **Right:** An enlarged part of the room demonstrating windows highlighted green and a chair facing towards the user highlighted red.

As can be seen in the image above the spatial mapping can be used to map both

the room and the single object contained within the room. Any virtual objects placed within the room will be remembered and upon restart would still be displayed within the HoloLens. This allows for easy access to previous states along with reducing performance usage due to caching.
Spatial mapping is the key element to implement Interactivity between the virtual and the real. With the data obtained from the generated mesh, objects are able to collide with the real world along with the possibility of implementing physics into holograms and applications.
Initially the spatial mapped mesh is invisible which allows for decrease in GPU and screen clutter, however to properly test, record and inspect the behavior between the smoke simulation and the spatial mapped environment a visual mesh was applied to the scene that allowed for graphical confirmation of correct movement. To achieve this a number of Microsoft Courses were followed including Course 101[13] and Course 230[14] which dealt with both deployment of an application and spatial mapping.

5.2 Collision

Collision is one of the mandatory elements to achieve the desired goals. Using collision the smoke simulation will be able to interact with the spatial mapped environment along with virtual objects that are added to the scene. This section will discuss the three methods that were implemented during the course of the implementation. Ultimately only sliding was kept as the most adequate methodology for collision. The subsections are written in the order that they were attempted with the first two containing the decision making on why they were not suitable for this implementation.
Unity allows for custom collision however their in built physics are already optimized and easy to attach to a simulation without much extra work to implement our own collision detection. The key elements needed for objects to collide within Unity are colliders and rigid bodies. A sphere collider was used for both smoke and vortex particles. A necessary large size was used to avoid particles from phasing through either badly mapped or not yet mapped areas within the HoloLens. A radius of 0.15 was used to suit both the particles in their initial state along with particles that would increase in scale over lifetime without the need to increase the size of the sphere collider with them. The second element that was crucial for the collision to work was a rigid body.

This allowed for the simulation of behavior upon collision. Rigid bodies within Unity use their own physical properties like gravity, mass ,and drag. This is however completely negated since their collision is overwritten within the files hence those elements were either set as default or turned off where possible. Without rigid bodies attached to the particles the collision response would not be recorded with only having a sphere collider attached.

5.2.1 Collision Response

The first methodology used is called collision response. This methodology detects collision and acts upon impact. It can be summarized in three steps:

1. Detect Collision: Done by Unity within our implementation.
2. Apply Changes to Movement.
3. Move object away from other collider.

Collision response methodologies can be summed up into either allowed penetration or not. The following two collision response methodologies were presented by Michael Manzke while the presentation itself was written by John Dingliana[6]. Spring based response is a good example of collision response that allows penetration. The forces applied to the objects are proportional to the penetration. However this is a costly procedure as smaller time steps are required to calculate the response from the springs the more greater the penetration. This method was instantly non plausible due to two reasons. You will see how these two reasons apply to the other collision response methodology. Firstly our implementations is time to delta time of each update meaning that adjusting the time-step needed to calculate the response is not plausible. Secondly a bouncing effect is not suitable for smoke simulation as smoke particles do not bounce of the surface.

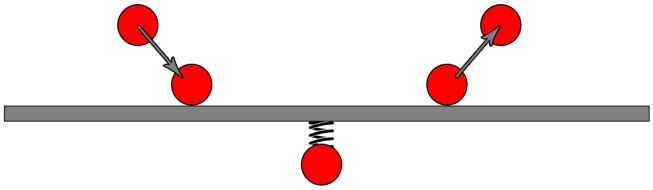

Figure 5.2: As the sphere is moving towards the object on the left side no collision is applied but once the sphere penetrates the wall it is pushed out using a spring effect on the right side.

The second method is rigid body collision which also incorporates impulse based collision response. This methodology revolves around movement adjustments during the moment of impact. This can either be using a calculated impulse that's applied by force or adjusting the velocity or position of the object. This was the methodology that was applied for our implementation which was later switched to collision avoidance. This was done by implementing calculating the reflection vector upon collision of the initial velocity vector and then setting the velocity vector to the reflection vector.

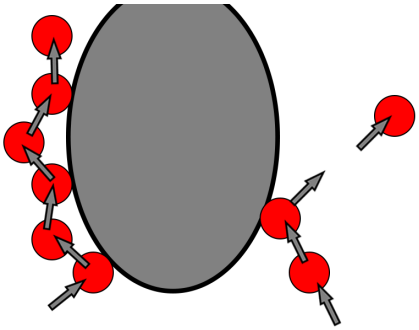

Figure 5.3: **Left:** Collision response where the object would bounce away from the collider and then back towards it repeatedly. **Right:** A collision response where the object would completely bounce away from the collider

As can be seen in the above image two version of this was unsatisfactory. The first

version would completely bounce of the object which is not natural (pictured right) and the second version (pictured left) would revert to the original velocity after a few frames resulting in a constant bouncing effect which is also unnatural to a smoke simulation. Hence this methodology was scrapped to implement collision avoidance.

5.2.2 Collision Avoidance

This implementation was based on a tutorial from Fernando Bevilacqua[2]. Collision avoidance is based around the task of avoiding objects in advance of collision. This is primarily done by predicting the future position either using ray casts or progressing movement along the current velocity. This is primarily done using the following formula.

$$FuturePosition = Position + normalize(Velocity) * Distance$$

This allows to predict the future position and check if it will collide with an object. Once this is done, the first object to be collided with will be avoided. This step was done by calculating the avoidance force. Avoidance force is calculated based on the centers of objects to create a force that can be applied to a velocity which will ultimately cause it to adjust it course away from the future collision.
While implementing this we came across a few problems. The first problem being that we were relying on the Unity sphere colliders to calculate collision detection rather than our own algorithm performing this task. This led firstly to the problem of not being able to use future positioning to calculate position and secondly not being able to collide with the spatial mapped mesh due to the fact that we will be colliding with it from the inside out rather than the other way. The second problem is associated with the spatial mapping again where we never aim to fully avoid the mesh as this will not be possible.
To combat these problems the future position code was eliminated leaving only the code for avoidance force calculation. Now this code would run upon actual collision as opposed to future collision. This led to unsatisfactory results where the particles would clump up and upon a large build up push through the colliders like a large build up of water bursting through a dam. However even if collision avoidance did perform

as expected graphical representation would be inaccurate where particles would avoid the object a lot in advance.

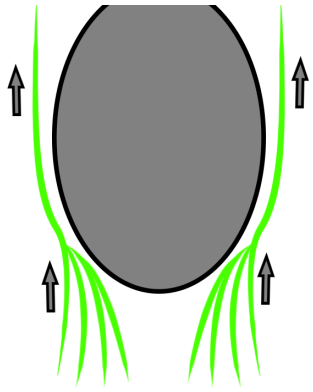

Figure 5.4: Collision avoidance as moving around the object before actually colliding with it.

As can be seen above due to the fact that the particles would avoid the object, the visual effect of particles engulfing the object would not be possible.

5.2.3 Sliding

The final methodology considered for this implementation was sliding. This was ultimately the methodology that would stay as the final collision behavior that the smoke simulation would follow. Sliding is a very interesting methodology that allows for an interesting effect, however it also has some minor flaws that will be discussed.
To implement sliding we needed to access the normal vectors of the surfaces that the particles would be colliding with. Luckily for us, spatial mapping allows for the access of normal vectors of each triangle that represents a surface making this implementation suitable for our simulation. The key knowledge for this implementation was obtained from a paper by Fauerby, Kasper [7] titled "Improved Collision detection and Response" which deals in detail with the logic and necessary knowledge to implement sliding between two objects. Our implementation of this articles resulted in the following formula.

$$\vec{v} -= \hat{N} \times \vec{v}.\hat{N}$$

Where $\hat{N}$ is equals to the normal vector and $\vec{v}$ is equals to the current movement vector. By simply subtracting the movement into the collider from the current movement vector the particle is able to avoid collision and slide along the plane.

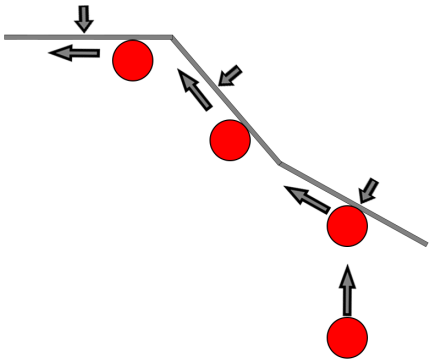

Figure 5.5: Movement of a particle along the normal planes of a collider object.

This methodology is very effective and much more plausible than the other two because firstly it conserves energy since there is never any stop in motion, allowing for a fluid and steady movement along any surface. Secondly sliding is capable on moving along any object that has a normal plane which in Unity is built into any object. Lastly this algorithm allows for very aesthetically pleasing movement which is simulates smoke like behavior of simply sliding along objects rather than bouncing of them.

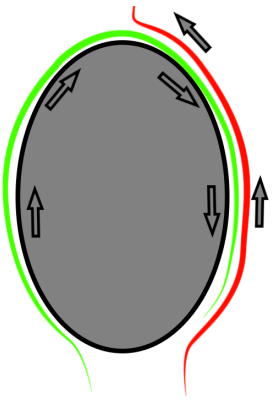

Figure 5.6: Movement of particles around a spherical object creating the surrounding effect.

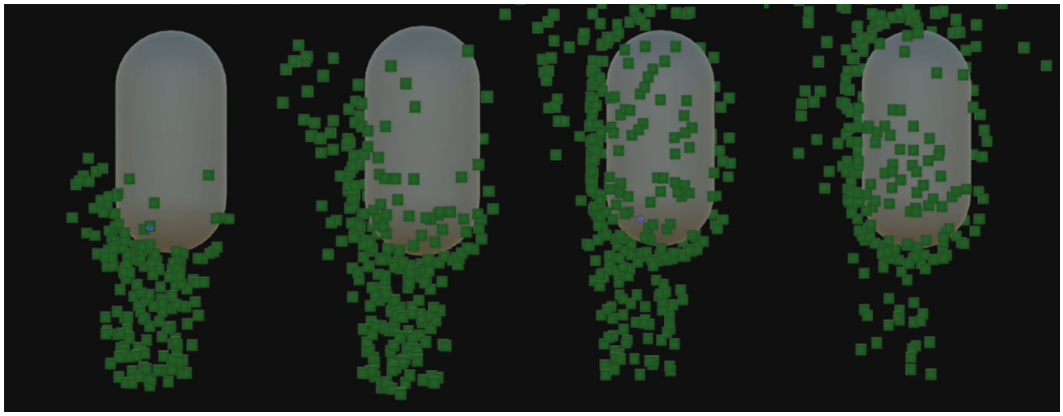

Figure 5.7: Unity render of particles surrounding and object creating the engulfing effect over time.

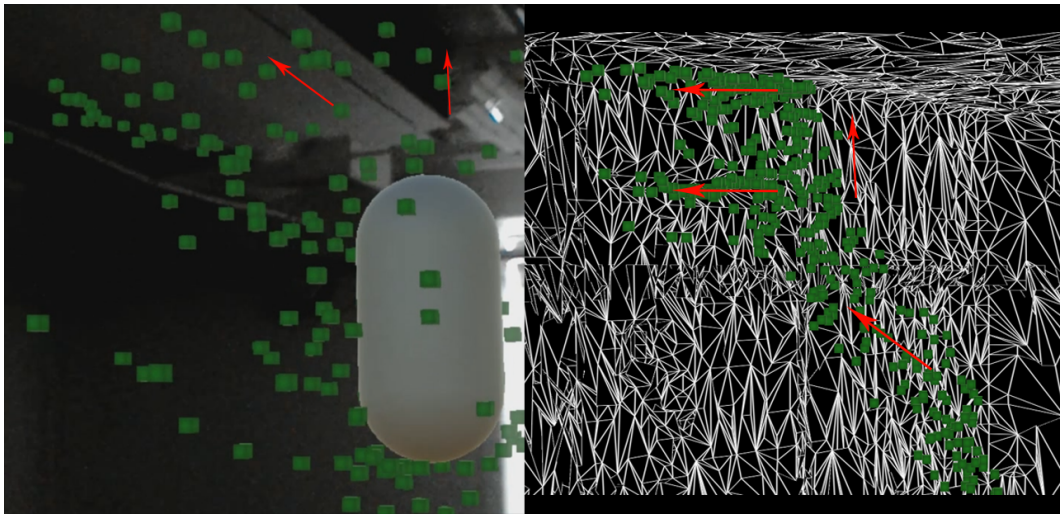

Figure 5.8: **Left:** HoloLens recording of particles moving along spatial mapped objects. **Right:** Unity screen shot of particles sliding up a wall and building up on top.

The algorithm also has a flaw where sometimes the particles would get stuck in a loop of simply circling objects if they are fully spherical. This flaw does not apply to other object shapes. The code for sliding can be viewed in section .4 of the Appendix.

5.3 Optimization

Optimization of the collision code was done both during the implementation and by the Microsoft HoloLens iteself. Firstly we will discuss mesh caching and how this helped with the performance of collision. Mesh caching is something that the HoloLens does automatically meaning that it will remember the meshes already in the environment as they will have a tag assigned to each surface[18] which both helps the HoloLens remember the objects but also realize where it itself is located. This allows for easier access of the normal vectors to calculate the needed velocity adjustments ultimately improving the performance.
The second aspect we discuss is the code adjustments that were attempted to increase performance within the collision code. Firstly the vortex particles do not have any collision with objects but only smoke particles and rely primarily on the simulation to move them which means that they will go through objects and out of the scene. This choice was taken since their effect is not as prominent as collision while also reducing the workload that will be needed if they were to collide with objects. Secondly the removal of certain elements that are more intensive to calculate was implemented especially again within the vortex where the distance between it and the smoke particles was removed to a static intensity multiplier allowing for quicker computation time and allowing to focus more performance into larger particle counts.

Chapter 6

Rendering

6.1 Materials & Shaders

The first area of graphical representation of smoke particles for this dissertation was the research into materials and shaders. However an issue was reached due to the limitation discovered within Unity. Although you are able to write custom shaders you are not able to do so without applying a material onto the objects. This meant that it was not possible to write a complete custom shader that would use the smoke simulation algorithms particle positions. An alternative was needed to represent the particles. Since it was mandatory to have a visual object attached to each particle position, it narrowed down the search to two solutions.

- 3D Object e.g. sphere
- 2D Texture/Sprite representation

The first option proved to be inefficient as even within Unity rather than the HoloLens emulator we were experiencing frame drops and stuttering. Initially a sphere was implemented however the number of polygons required to imitate smoothness was astounding, a unit cube was used instead which was a large improvement however even with this it was not graphically pleasing. Although the idea of the cubes was scrapped they were still used for testing the algorithm as they were easier to follow independently rather than the chosen solution.

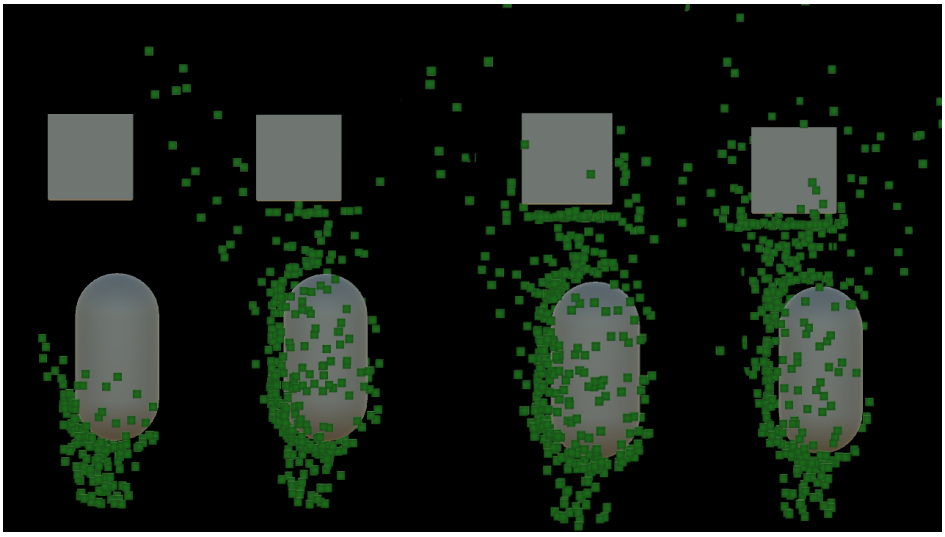

Figure 6.1: Movement of cube particles around multiple objects during testing.

The chosen solution ended up being a 2D Sprite which was a simple image of a sphere that allowed for improved performance while also able to imitate alpha blended smoke particles. Using the sprite approach we eliminated large amounts of polygons and the need to write a smoke shader since ultimately all that will need to be done is rendering billboard textures.

6.2 Texturing

This section deals with the actual sprite itself that was used along with it's introduction into the Unity engine. Firstly the texture itself was created within Adobe Photoshop CS6 and went through 3 separate instances where different textures were created using different brushes and effects.

Figure 6.2: Three sprites that were implemented during development. **Right:** Final sprite used for end simulation.

Initially a fully solid white circle was used to allow for adjusting scaling and opacity settings later it was changed to the middle texture with a bright center and dull outer ring. However when many particles were near each other it was clear where one particle ended and another began since the inner and outer rings had a high difference in opacity. Finally the final texture was incorporated demonstrating a linear fall off from the inner to the outer areas of the texture causing a linear blend between each sprite. With this final linear opacity fall off we were able to create a soft particle that is outlined within Larsson, Robert [10]. This allows for seamless blending between particles to reduce artifacts and anti aliasing. The use of a greyscale texture also allowed us to adjust the coloring of smoke particles very easily simply by adjusting the settings within Unity.

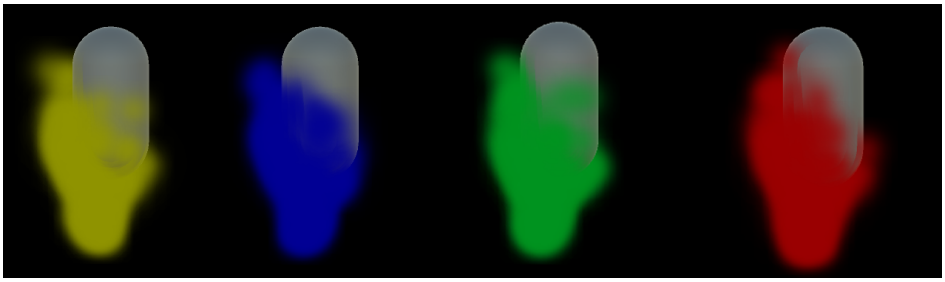

Figure 6.3: Different colors used for smoke particles within Unity.

Figure 6.4: Smoke Particles captured within the Microsoft HoloLens moving around real life objects.

6.3 Lighting

One element of rendering that was greatly considered was lighting and is it plausible to implement for this simulation. Initially the answer was yes as Unity is already able to implement lit particles. However through more thinking some problems with this idea came to light.

Firstly the particles themselves will already be semi lit. Since the particles use opacity, within the HoloLens light is able to travel through the smoke particles making it harder to see them the brighter the room is. This means that in a way they were already lit based on real world lighting.

Secondly making the particles lit would be and issue due to the fact that either all virtual objects within an environment or none of them should be lit. If some objects are lit while others aren't and issue will arise with the immersion experience of the simulation.

Finally lit particles would require the knowledge of light sources including their locations, directions and intensity to accurately simulate lit particles, however this is not yet possible within the HoloLens due to this limited capabilities. At the same time if this were possible, it would need to be applied to all virtual objects otherwise the

second reason becomes relevant again. On the other hand implementing a virtual light source would cause the opposite where the objects and particles would be lit but this would not affect the real world rendering with the scene not being affected.

6.4 Behaviour

One of the key behaviors implemented for the rendering of smoke particles is called billboarding. Similar to that which was mentioned in Larsson, Robert [10]. This method is designed to give the feel of a 3D volume while only using 2D sprites. This is done by drawing sprites facing towards the camera while at the same time letting them overlap.The article discussed two methods of achieving this. The first being adjustments to the view matrix while the other being done within the vertex shader to rotate the points so that they are aligned with the view plane. Our solution was neither. Our implementation used the look-at method to adjust the particle to always face towards the current camera. Without this being done movement around the real room would lead to particles being shown as flat.

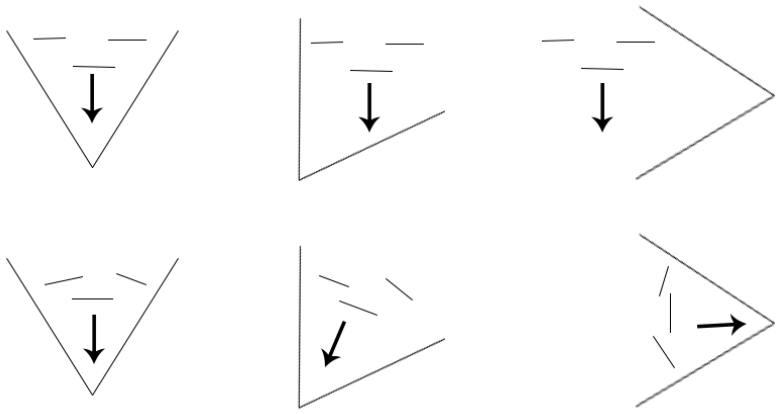

Figure 6.5: **Top row:** Billboarding rotation not applied. Particles constantly aiming towards the same point. **Bottom row:** Billboarding rotation applied, sprites always aiming towards the camera.

In the next chapter we will discuss the performance of the simulation however even at this stage it was clear that the simulation would not be able to handle particles of large quantity meaning that if the particle count would be low, a patchy smoke

simulation would happen where only small amount of particles was used. To combat this a dispersion algorithm was implemented that would scale and adjust the opacity of particles the longer they are alive. Based on two variables called *Dispersion Factor* and *Scale Factor* it was possible to fill in blanks where simulations used low particle counts. The results can be seen below.

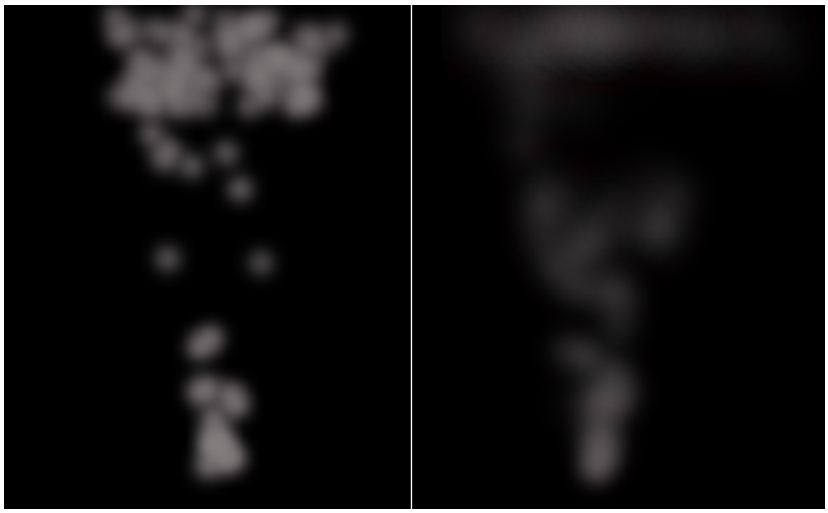

Figure 6.6: 100 particle count simulation. **Left:** Dispersion formula not applied, easy to see individual particles. **Right:** Formula applied, blending improved aesthetics however problem not fully solved.

The formula to achieve the opacity is as follows where o is initial opacity, a is current opacity, tt is total lifetime and tl is time left till death.

$$a = ((tl/DispertionFactor)/tt) \times o$$

The second formula deals with scale dispersion and follows the same symbols as the previous formula with the addition of s representing current scale and is representing the initial scale.

$$s = is + (is \times ScaleFactor)(1 - (tl/tt))$$

With both of these combined we were able to create the effect pictured above. The code for this can be seen in Appendix .1.

6.5 Optimization

Optimization within this part of the implementation dealt heavily with the texture itself along with features within Unity. Initially the sprite size was 512x512 creating a very detailed smoke particle however using this size the simulation would lag causing the texture to be reduced to 16x16. This was also a problem as due to the scale behavior mentioned above the particles would reduce in quality greatly as the scale would increase. This resulted in trial and error attempting all power of two sizes between 16x16 to 512x512 with 64x64 being the final choice. The selected file choice being compressed PNG. The texture was also saved in greyscale allowing for reduction in size.
Along with this an unlit mobile shader was used within Unity to represent the texture. This choice was taken due to Unity already made optimization within that shader to run on mobile devices. sprite size,unlit material, greyscale texture.

6.5.1 Editor GUI

One of the last part of the implementation was the introduction of Graphical User Interface (GUI) editor that would allow to change settings for simulations within Unity rather than the code itself. This greatly benefits future projects where quick changes can be made. The GUI implements many adjustments with the key ones listed below.

- Particle Count: Sets the maximum amount of particles that can be alive.
- Min and Max Life: These two settings adjust the life span of particles.
- Vortex Change: Sets the frequency for how often vortex particles will be spawned.
- Real Time: This boolean controls if the simulation is real time with particles living and dying or a set amount of particles is created.
- Wind Force: This allows to add wind force in any direction to the particles.
- Rising Power: This adjusts the speed at which particles rise based on the Buoyancy force.
- Dispersion Factor: Sets how quickly the particles lose opacity over time.

- Scale Factor: Sets how quickly the particles become larger over time.

Chapter 7

Results and Discussion

7.1 Results

This section deals with the gathered results after the implementation stage was completed. The testing was primarily performed using the HoloLens emulator to record the primary result data. However a second test was recorded on the physical HoloLens with only a few test cases being run to record the difference in results. Since the HoloLens emulator is a virtual machine it is near identical to the performance of the physical unit. Hence a very similar result was recorded when the test cases were performed on the device itself.

7.1.1 Performance

Performance was one of the key goals that were mandatory for a successful result based on the original goals of the dissertation. One of the issues that was noticed during testing is the limited capabilities of feedback from the simulation. With only a web page to record performance and statistics available to record results, some of the desired performance tests that were planned were not viable to be recorded. This is due to the fact that only a fully deployed application in release mode was able to be deployed onto the device and emulator. This led to only the HoloLens provided statistics to be accessible. The tests that were not able to be analyzed included, single particle CPU and GPU render time along with batch render times used to create a set number of particles into the scene. The available data is as follows: *CPU Usage*, *GPU Usage*,

FPS and *Memory Used*.
One assumption had to be made at this point being that the computer the emulator was run on was more powerful than the power needed to accurately run the HoloLens emulator. Due to the fact that the exact performance statistics of the CPU and GPU of the HoloLens are unavailable to be compared their computer alternatives, the assumption was made on the fact that the simulation run within Unity was always performing significantly better at higher particle counts without any decrease in performance.
All of the following sections were tested within four different situations.

- Algorithm only: Here the simulation would run with only the algorithm running without any graphical representation on a set number of particles.
- Rendering only: Here the simulation would run only rendering a set number of particles moving from side to side without the inclusion of the algorithm.
- Algorithm & Rendering: Here a set number of particles was spawned instantly that would both be simulated and rendered however they would not include the spawning and death of particles.
- Real Time: This is the final test case where the particles would spawn over time to reach a maximum count while also dying upon lifetime hitting 0. This is end product as the simulation is meant to be used. Within this simulation a minimum and maximum life of 0 and 30 seconds was set respectively.

Frames Per Second

FPS is the key element for testing the simulation. As previously stated according to Microsoft the HoloLens should perform at a recommended constant 60 FPS [16]. However even from the earlier stages of the implementation it was noticed that this may not be plausible, hence a compromise was made that was mentioned previously to reduce this to 30 FPS for our implementation since this still coincides with the current mobile and gaming console performance.
Based on this only test cases where results over 30 FPS are recorded as successful.

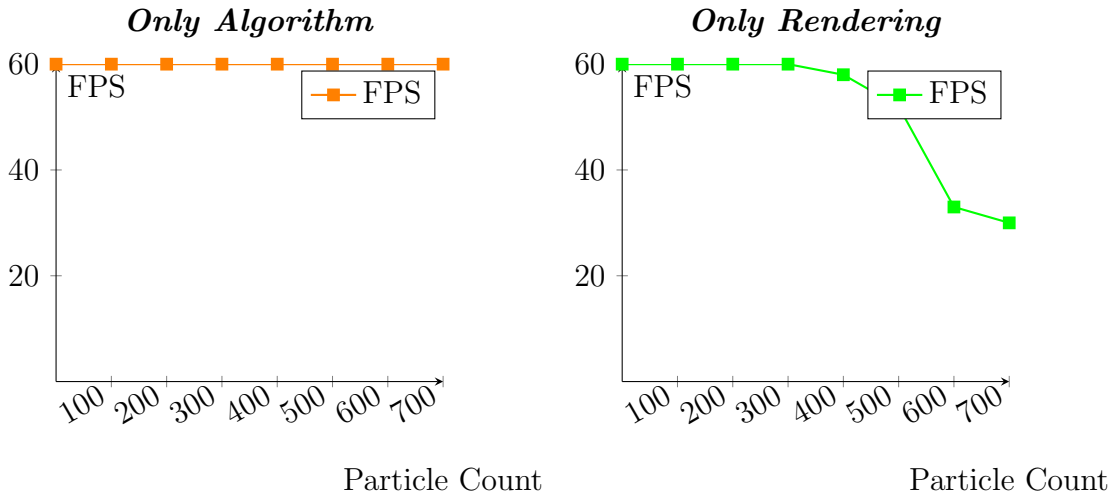

Figure 7.1: FPS performance for only algorithm and only rendering test cases.

Within the tables above it is clear that the big drawback of performance is rendering with the algorithm able to perform at maximum capped FPS without any drops. On the other hand when only rendering the particles somewhere between 300 and 400 particles the FPS starts to drop with 700 particles reaching 30 FPS.
The testing was done only up to 700 particles because after that the initial stress of creating that many particles caused a strain on the emulator which would cause it to freeze and become unresponsive.

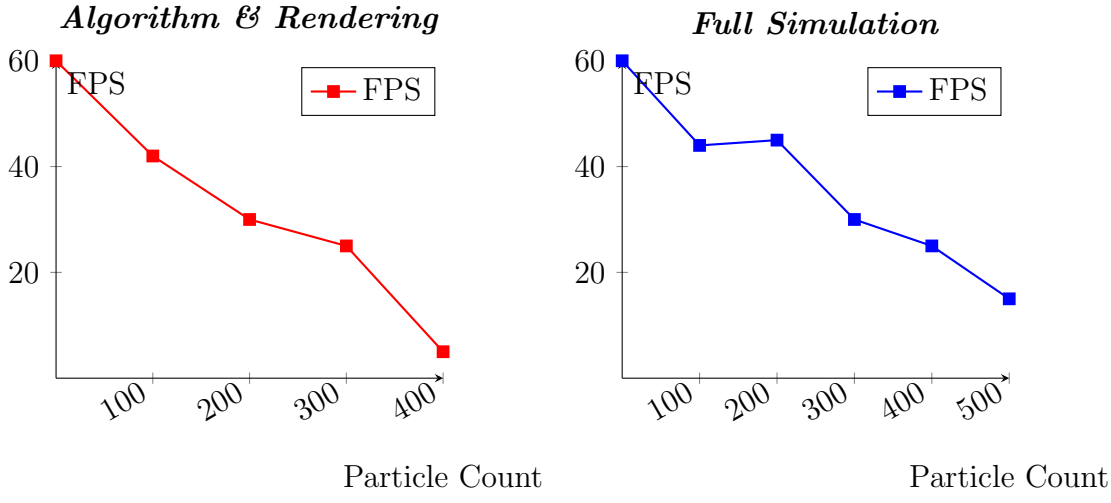

Figure 7.2: FPS performance where **Left:** both the algorithm and rendering is used and **Right:** full simulation.

Here a more drastic drop in FPS is seen with the algorithm and rendering table showing a drastic drop after 300 particles from 25 to 5 FPS. However the full simulation performed better with a more linear drop with 30 particles reaching the maximum viable FPS of 30. The reasoning for this in our opinion is that the initial stress from creating a large amount of particles at start with simulation pictured on the left caused stress on the machine continuing during the simulation as well.

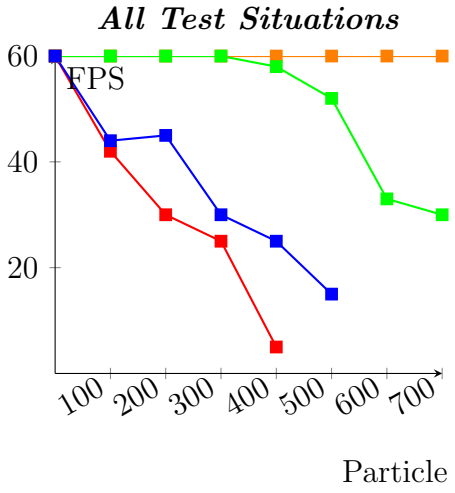

Figure 7.3: Combined graph of all 4 test situations.

Figure 7.3 displays all four situations into one table where we can easily see how the drop in performance for full simulations is very similar in that it starts at 300 particles while the rendering also starts to drop between 300 and 400 particles. Giving us a max particle count for viable simulation being 300.

CPU Usage

The second performance statistic we were able to access was CPU usage which represents the computational usage of the processor during the simulation. Just like the FPS performance analysis a similar trend started showing up. The first figure shown below displays the CPU usage for the algorithm and rendering simulations separately.

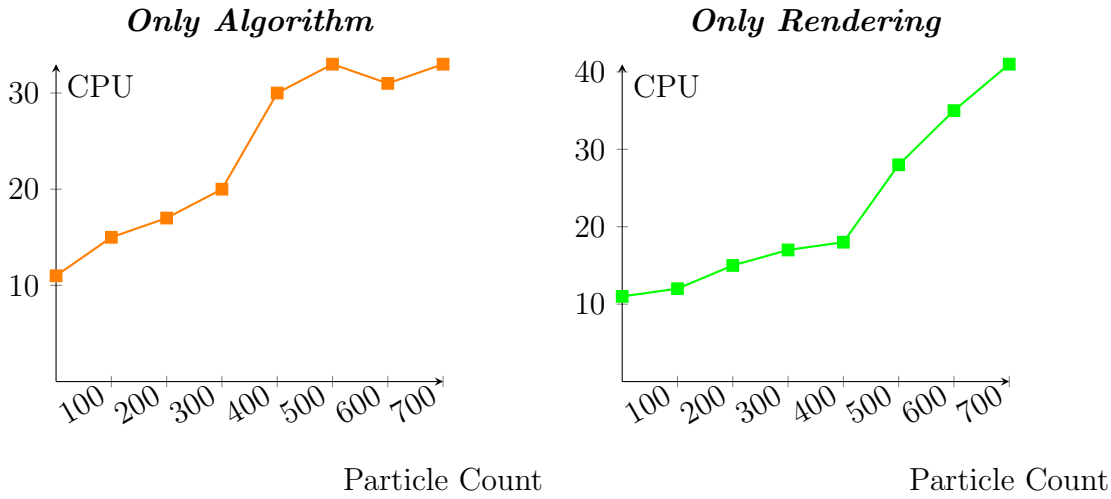

Figure 7.4: CPU performance for only algorithm and only rendering test cases.

The base value of CPU usage at 0 particles was recorded at 11% with a maximum reached usage going over 30% for the algorithm and over 40% for the rendering tests. Similar to that of FPS initially the usage would increase linearly until hitting 300 particles for the algorithm and 400 particles for the rendering at which point a more rapid increase in usage is seen.

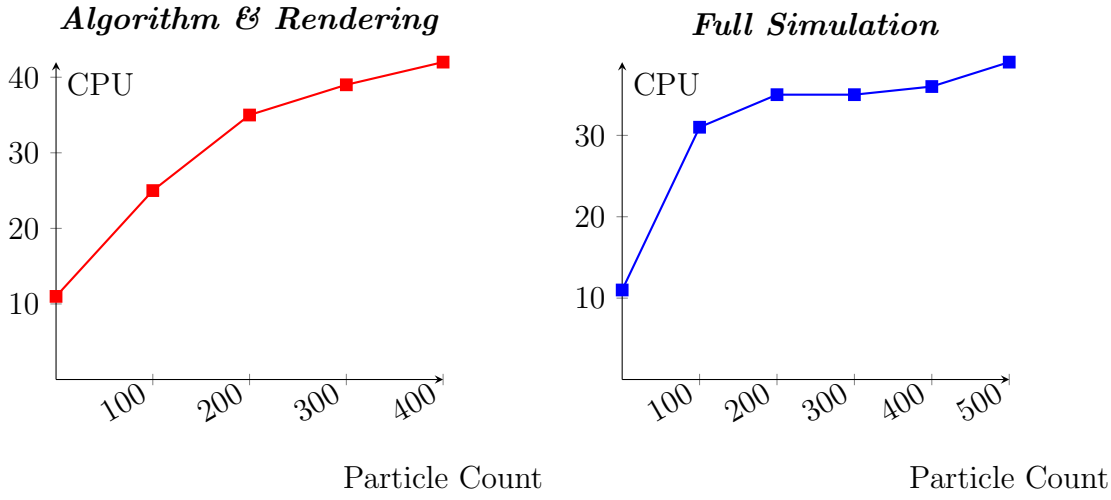

Figure 7.5: CPU usage where **Left:** both the algorithm and rendering is used and **Right:** full simulation.

Here we see a different situation to the previous figure where a rapid increase began

at a specific number of particles. Within these test cases a more linear increase is shown with only the full simulation consisting of an initial high spike in CPU usage. This could be due to the inclusion of the spawning and death methods along with introduction of lifetimes compared to them being excluded in the left table. The major difference between the previous two graphs and these is that to reach 40%+ usage a lower amount of particles was needed.
As previously mentioned we were not able to test over 400 particles for the algorithm and rendering test case due to initial strain on the machine hence we are not sure how the graph would progress after the 400 particles that so far were identified as one of the key moments where large changes occur.

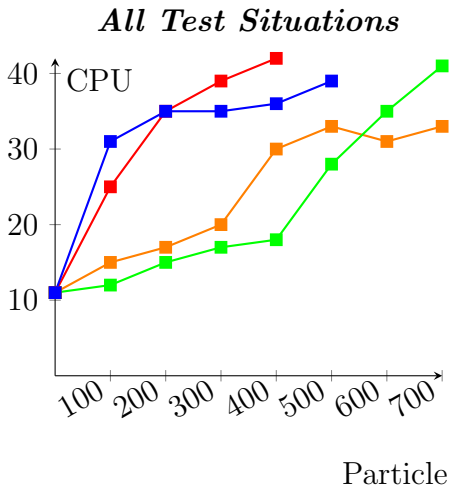

Figure 7.6: Combined graph of all 4 test situations focusing on CPU usage.

Above we can see that although all 4 test situations follow their own path with different initial usage increase and different progression. They all seem to be starting to converge between 400 and 500 particles with a hypothesis that they will all converge after 700 particles.

GPU Usage

The third performance statistic analyzed is GPU usage which is the performance that it takes the GPU to render the smoke simulation. The base value for this is 31% which

according to Microsoft is primarily used by the scanners, spatial mapping, menus ,and other functionality that comes within the HoloLens.
This is somewhat a problems since near a third of the power available is instantly used up before the simulation is even started. For the graphs an adjustment was made to set the minimum Y axis value to 30 since it was very difficult to visualize changes between different particle counts during the simulation.

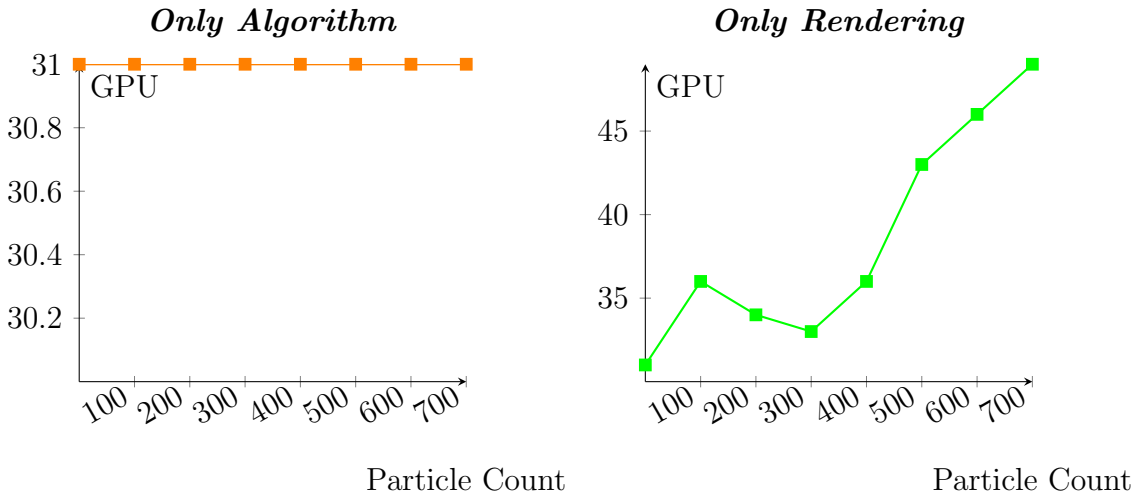

Figure 7.7: GPU performance for only algorithm and only rendering test cases.

The first two tables like previously display the algorithm and rendering test cases individually. The algorithm graph never changed for each particle instance primarily due to the fact that when only the algorithm is simulating no graphical rendering is being performed meaning that we can be sure that our algorithm is solely CPU based. The rendering statistics however were initially fluctuating and only began to rapidly increase after 300 to 400 particles.

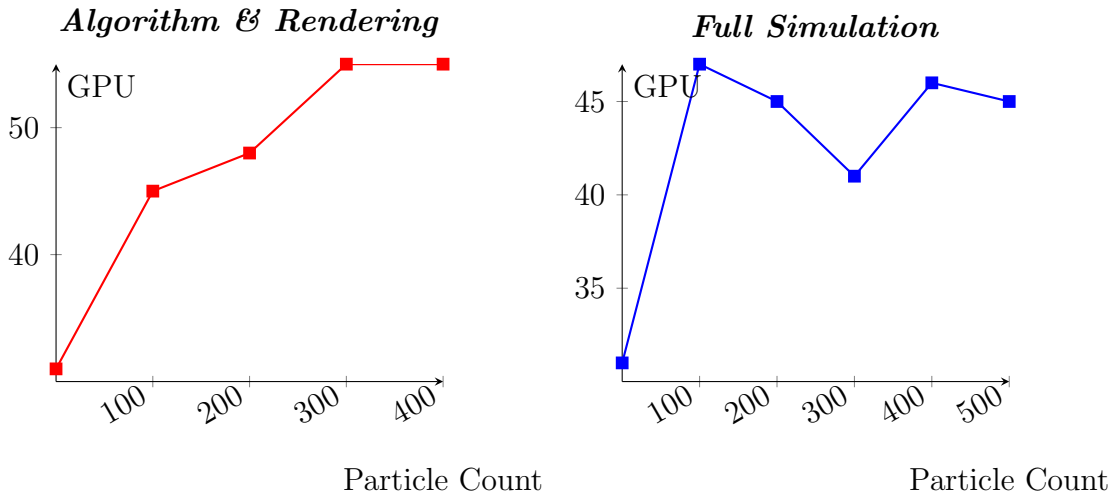

Figure 7.8: GPU usage where **Left:** both the algorithm and rendering is used and **Right:** full simulation.

In Figure 7.8 we experienced something that was initially difficult to explain where the usage would either fluctuate or stop increasing. This was very interesting since it was more unpredictable as to it's behavior. The reason for this ended up being object occlusion that is within the HoloLens which would increase and decrease GPU usage if particle came in or out of the screen. Along with this upon a large particle count death a drop in usage was also recorded since after the scaling of particles over life in the full simulation longer GPU render time would be needed, hence upon death a smaller particle would replace the large one requiring less render time.

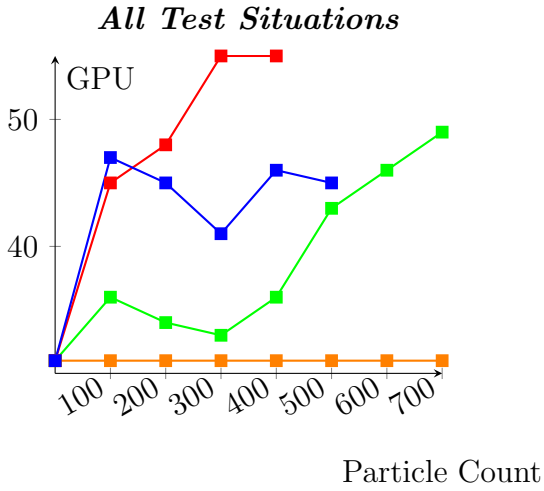

Figure 7.9: Combined graph of all 4 test situations focusing on GPU usage.

Once again within the figure above it is very visible how the GPU usage starts to increase after 200-400 particles with very high usage percentages recorded with particles counts over 300.

Memory Usage

The final performance statistic that we were able to analyze is the memory usage. The initial available RAM within the HoloLens is 2 gigabytes. Microsoft Recommends a limit on the usage over 1 gigabyte (1024 megabytes)[16]. Without any of the test cases running the starting memory used is 740-745 megabytes. Throughout all 4 test cases only the algorithm itself managed to reach that limit with 700 particles hitting 1 gigabyte exactly. On the other hand a rendering only simulation reached 762 megabytes used since it was only the same sprite being re-drawn without the need for any extra storage.
In between 0 particles and maximum particles in each test case the memory usage would always fluctuate while increasing at the same time. The full simulation peaked with 789 megabytes used.

7.2 Discussion

This is the final section of the dissertation which wraps up all of the chapters to a final discussion based on the implementation and results.

7.2.1 Limitations

Throughout both the implementation and analyses stage of the work many limitations were noticed that either hindered the ability to perform a certain task or to adjust the implementation.

Microsoft HoloLens

Due to the limited power of the HoloLens previously mentioned it was very difficult to create the high accuracy physical simulation that was initially intended. A balance between accuracy and performance was mandatory for a successful simulation to be developed. Along with this a limit on the graphical capabilities of the HoloLens was also evident resulting in a minimal graphical representation of a smoke particle to be achievable.

Unity Limitations

Certain areas of Unity were locked to us from the beginning. The ability to access function within their particle system component meant that the algorithm needed to be implemented into our own particle system as opposed to changing the behavior of the in built one. Along with this a limitation within the shader implementation mentioned previously caused issues since it was impossible to create a shader without an object to represent and a material applied to the object.

HoloLens Statistics

Another previously mentioned limitation is the limited performance statistics capabilities provided with the HoloLens. Unlike computer debugging you are unable to access any debugging features once you deploy the application. The only way to test it's performance is either through the use of a Head-up Display (HUD) which will also

drain performance or using the device localhost website. This website only provides a limited amount of statistics which are capable of only reporting the bare minimum needed statistics for a thorough testing.

7.2.2 Conclusions

We were able to achieve a physically based smoke simulation that is capable to be simulated in mixed reality with the use of the Microsoft HoloLens. Although this simulation is not as accurate as some of the other methodologies presented here it is able to combine both the physically simulated fluid motion along with a graphical representation of it. This was achieved using a Lagrangian fluid simulation methodology with Unity based rendering. The resulting implementation proved to be very versatile with the ability to alter settings for custom simulations of smoke while also able to perform over the set goal of 30FPS as long as the particle count does not exceed the recommended particle count of 300 for the HoloLens and the emulator. Along with these benefits the resulting simulation was able to traverse an real world environment as a virtual object colliding with both the real and virtual world leading to all of the initial goals that were set being achieved.

The main contributions that were achieved within this work is the simulation that is capable of performing power draining mathematical calculations while also rendering particles on a low performing device. This simulation also means that the implementation managed to do a complete cycle starting with an algorithm based on real world physics which was implemented into a virtual environment and later displayed and interacted within the real world where it originated.

However this wasn't the only thing that stood out within the implementation. To combat limited performance capabilities, sacrifices were made and adjustments had to be performed to allow the simulation to even be successful within the HoloLens. The prime example of this is the dispersion algorithms that were developed from scratch to combat the visual representation of low particle counts. This allowed to mask the limitations of the HoloLens by allowing a more dynamic smoke simulation that would fill up a room. The final results can be seen below which were recorded using Cortana within the HoloLens. The recording due to smoke opacity is not represented as vibrantly as through the HoloLens screen.

Figure 7.10: Smoke simulation recorded within HoloLens interacting and moving around a real environment. **Top row:** bright lighting. **Other rows:** dark lighting.

7.2.3 Future Work

If this work to be continued further our recommendations would be to firstly attempt the implementation on if possible a newer iteration of the HoloLens to see the improvement. Otherwise an implementation within a different engine supported by the HoloLens could be attempted to solve the issue of rendering only using a shader which could potentially improve the performance further.
Some other recommendations would be to attempt another algorithm. Perhaps Stam, Jos [23] Navier-Stokes equations to see the difference in performance along with tackling the issue of grid sizes needed for the simulation which this paper was not able to solve. Lastly the improvement of already existing forces or the implementation of new forces could also be a possibility for future work and research to further enhance the quality of the simulation by adding more real world physics.

Bibliography

[1] Badis, Ydri. *Computational physics: an introduction to Monte Carlo simulations of matrix field theory*. World Scientific, 2017.

[2] Fernando Bevilacqua. Understanding steering behaviors: Collision avoidance. URL `https://gamedevelopment.tutsplus.com/tutorials/understanding-steering-behaviors-collision-avoidance--gamedev-7777`.

[3] Bridson, Robert and Müller-Fischer, Matthias. Fluid simulation: Siggraph 2007 course notes video files associated with this course are available from the citation page. In *ACM SIGGRAPH 2007 courses*, pages 1–81. ACM, 2007.

[4] Crane, Keenan and Llamas, Ignacio and Tariq, Sarah. Real-time simulation and rendering of 3d fluids. *GPU gems*, 3(1), 2007.

[5] Dinev, Dimitar and Liu, Tiantian and Kavan, Ladislav. Stabilizing integrators for real-time physics. *ACM Transactions on Graphics (TOG)*, 37(1):9, 2018.

[6] John Dingliana. Rigid body collision response. URL `https://www.scss.tcd.ie/~manzkem/CS7057/cs7057-1516-09-CollisionResponse-mm.pdf`.

[7] Fauerby, Kasper. Improved collision detection and response. *DK, private communication*, 2003.

[8] Ihmsen, Markus and Orthmann, Jens and Solenthaler, Barbara and Kolb, Andreas and Teschner, Matthias. Sph fluids in computer graphics. 2014.

[9] Intel. Demystifying the virtual reality landscape. URL `https://www.intel.com/content/www/us/en/tech-tips-and-tricks/virtual-reality-vs-augmented-reality.html`.

[10] Larsson, Robert. Interactive real-time smoke rendering. 2010.

[11] Lyes, Timothy. *GPU accelerated particle methods for simulating and rendering fire and water effects*. PhD thesis, Massey University, 2015.

[12] McGuire, Morgan. A real-time, controllable simulator for plausible smoke. Technical report, Citeseer, 2006.

[13] Microsoft. Mr basics 101: Complete project with device, . URL `https://docs.microsoft.com/en-us/windows/mixed-reality/holograms-101`.

[14] Microsoft. Course 230: Spatial mapping, . URL `https://docs.microsoft.com/en-us/windows/mixed-reality/holograms-230`.

[15] Microsoft. Unity development overview, . URL `https://docs.microsoft.com/en-us/windows/mixed-reality/unity-development-overview`.

[16] Microsoft. Performance recommendations for hololens apps, . URL `https://docs.microsoft.com/en-us/windows/mixed-reality/performance-recommendations-for-hololens-apps`.

[17] Microsoft. Mr 100 tutorial, . URL `https://docs.microsoft.com/en-us/windows/mixed-reality/holograms-100`.

[18] Microsoft. Spatial mapping, . URL `https://docs.microsoft.com/en-us/windows/mixed-reality/spatial-mapping`.

[19] Nguyen, Hubert. *Gpu gems 3*. Addison-Wesley Professional, 2007.

[20] NIST. Fds and smokeview. URL `https://www.nist.gov/services-resources/software/fds-and-smokeview`.

[21] Ren, Bo and Yan, Xiao and Yang, Tao and Li, Chen-feng and Lin, Ming C and Hu, Shi-min. Fast sph simulation for gaseous fluids. *The Visual Computer*, 32(4): 523–534, 2016.

[22] Rørbech, Marinus. Real-time simulation of smoke using graphics hardware. *SIMS 2004*, page 331, 2004.

[23] Stam, Jos. Real-time fluid dynamics for games. In *Proceedings of the game developer conference*, volume 18, page 25, 2003.

[24] Studio, KOG. A fire and smoke simulation for mobile game. *development*, 2(4), 2008.

[25] Techopedia. Homebrew. URL `https://www.techopedia.com/definition/10649/homebrew`.

[26] Weaver, T and Xiao, Zhidong. Fluid simulation by the smoothed particle hydrodynamics method: A survey. VISIGRAPP 2016-Proceedings of the 11th Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications 2016, 2016.

[27] Xu, Yongzhe and Kim, Eunju and Lee, Byungsoo. Simulation of smoke improve computing coordinate performance. *International Journal of Multimedia and Ubiquitous Engineering*, 8(4):363–376, 2013.

[28] Yongzhe, X and Lee, Kyunjoo and Kim, Eunju and Ki, Jaesug and Lee, Byungsoo. Simulation of smoke to improve unity 3d game engine particle system based on fds. *International Journal of Smart Home*, 7(5):263–274, 2013.

[29] Zhang, Z and Chen, Q. Comparison of the eulerian and lagrangian methods for predicting particle transport in enclosed spaces. *Atmospheric environment*, 41(25): 5236–5248, 2007.

Appendix (Code)

.1 Euler Integration

```
void Update () {
    m_time = Time.deltaTime;
    m_timeLeft -= m_time;

    //calculate forces
    Vector3 totalForce = Vector3.zero;
    totalForce += m_gravity;

    //Drag
    m_drag = (m_mass * m_velocity) * m_time;
    totalForce += m_drag;

    //Vortex
    totalForce += m_vortex;
    m_vortex = Vector3.zero;

    //Wind
    totalForce += m_wind;

    //Approximation
    m_velocity = m_time * (m_mass * totalForce) + m_velocity;
    m_position = m_velocity * m_time + m_position;

    this.transform.position = m_position;

    //Sprite Adjustments
    if (m_sprite)
    {
        m_density = ((m_timeLeft / m_dispersionFactor) / m_lifeTime) * m_initialOpacity;
        this.transform.localScale = m_initialScale + (m_initialScale*scaleFactor) * (1 - (m_timeLeft/m_lifeTime));
        m_color.a = m_density;
        this.GetComponent<SpriteRenderer>().color = m_color;
        this.GetComponent<SpriteRenderer>().transform.LookAt(Camera.main.transform.position, -Vector3.up);
    }
}
```

Figure 11: C Sharp code detailing the Euler Integration algorithm.

.2 Vortex Code

```
Vector3 otherPosition = col.gameObject.GetComponent<ParticleHandler>().GetPosition();
Vector3 d = (otherPosition - this.m_position)/Vector3.Distance(otherPosition,this.m_position);
col.gameObject.GetComponent<ParticleHandler>().SetVortex(Vector3.Cross(d,transform.rotation.eulerAngles)*0.1f);
```

Figure 12: C Sharp code containing the vortex force that is applied onto a smoke particle.

.3 Particle System

```
private void Start()
{
    if (!realtime)
    {
        for (int i = 0; i < maxParticles; i++)
        {
            time = Random.Range(minLife, maxLife);
            new Particle(transform.position, transform.rotation, time, objName);
        }
    }
}

void Update()
{
    if (realtime)
    {
        if (particleCount < maxParticles)
        {
            time = Random.Range(minLife, maxLife);
            if (Random.Range(0.0f, 1.0f) <= vortexChance)
            {
                Vortex v = new Vortex(transform.position, transform.rotation, time);
                vortices.Add(v);
            }
            else
            {
                Particle p = new Particle(transform.position, transform.rotation, time, objName);
                particles.Add(p);
                particleCount++;
            }
        }

        //Kill Particles
        for (int i = 0; i < particleCount; i++)
        {
            particles[i].m_life -= Time.deltaTime;
            if (particles[i].m_life < 0)
            {
                particles[i].Kill();
                particles.RemoveAt(i);
                particleCount--;
            }
        }

        for (int i = 0; i < vortices.Count; i++)
        {
            vortices[i].m_life -= Time.deltaTime;
            if (vortices[i].m_life < 0)
            {
                vortices[i].Kill();
                vortices.RemoveAt(i);
            }
        }
    }
}
```

Figure 13: C Sharp code detailing particle system spawn and death logic.

.4 Sliding

```
ContactPoint collisionPoint = col.contacts[0];
Vector3 normal = collisionPoint.normal;

m_velocity -= collisionPoint.normal * (Vector3.Dot(m_velocity, collisionPoint.normal));
```

Figure 14: C Sharp code for particles sliding upon collision across the collider.

Acronyms

AR	Augmented Reality.
CPU	Central Processing Unit.
FDS	Fire Dynamic Simulator.
FPS	Frames Per Second.
GUI	Graphical User Interface.
HPU	Hologram Processing Unit.
HUD	Head-up Display.
MR	Mixed Reality.
RAM	Random Access Memory.
RK2	Runge Kutta 2nd Order.
RK4	Runge Kutta 4th Order.
SPH	Smoothed Particle Hydrodynamics.