Comparing Performance of Rendering Methods for Physically Based Smoke Simulation

Daniel Filby

***Abstract*—This paper analyses the run-time performance difference between two rendering techniques for 3D grid-based smoke simulations, to provide an evaluation on whether their use in video-games is suitable. The two rendering techniques are, a voxel grid approach, and a volume sampling ray tracer. First conducting a comprehensive review on past and current optimisations, for each approach, the research then applies the most suited, to a custom built graphical artefact. An experiment testing each techniques' rendering time at increasing simulation resolutions, is used to determine the fastest renderer and identify any performance patterns. Testing will focus on frame rate, a vital factor for games, especially ones which are pursuing realism and would consider using smoke simulations. The findings prove that ray-casting performs better than voxels, independently and inside game environments, however certain tests revealed voxel rendering's potential.**

I. Introduction

THE standard of visual realism in games is constantly increasing, due to continual developments made by AAA titles. These improvements are led by innovations in the computer graphics field and advancements in personal computer hardware; Both allowing for higher levels of detail in models and effects as well as more complex rendering and lighting techniques. With such advancements, the ability of games to produce smoke that looks and moves realistically, becomes more and more expected by the players. This paper analyses two methods of rendering a visually impressive smoke simulation, with the aim of determining it's viability for game use.

The current preferred method for games to produce visual effects, such as smoke, is by using a simple particle system, which results in an adequate but unrealistic effect. Visuals can be greatly improved by modelling the smoke's movement in line with reality. A physically based simulation achieves this, by tracking the smoke's density and velocity across a discretized grid, using fluid dynamic equations derived from reality. The resulting smoke can be rendered in multiple ways, most commonly with ray-casting which this paper will compare to a voxels based approach. This process can be computationally costly which is why it's important to measure performance, to find the most efficient method for games to render realistic smoke.

The first rendering method covered in this paper is a voxel grid based approach. Since the format of a physically based smoke simulation is a 3D grid of density values, a one-to-one mapping can be applied to a voxel grid, of the same size. With each voxel, modeled as a small cube, visually representing the corresponding density, from the smoke grid, by altering its transparency. The resulting smoke should be rendered with as much accuracy as its simulated. By using the standard rasterization pipeline and employing instancing, this technique should be able to efficiently render the large amount of voxels required. Although this rendering technique seems the most intuitive, there is a lack of research exploring its use, which is why voxels are included in this paper's research, determining its appropriateness for rendering smoke.

The voxel method will be compared to a ray tracing renderer, which for the past 5 years has seen a large increase in popularity for games. This rendering technique produces accurate global lighting from extensively tracking light rays. This paper, however, will analyse a specific ray-casting technique called volume sampling, to render a physically based smoke simulation. Which involves tracking rays through the smoke's grid, to estimate the density along that view angle, which can then be used to visualise the simulation.

Along with the renderers performance comparison, an overall evaluation for physically based smoke's use in video games will be discussed. Aided by results from a specific test emulating a game environment, a determination can be made, of physically based smoke being a suitable improvement over particle systems, for games.

II. Related Work

This section will be split into three parts. The first will discuss how smoke is realistically simulated and will present current developments in the field. The following section will explore recent works related to ray tracing and current optimisation methods. The final section will cover the recent history of voxel grids with the most popular optimisation methods.

A. Smoke Simulation

The process of graphically modelling smoke has evolved over the last four decades, from using a simple particle system to following the laws of fluid motion. The advancements in this field are aimed towards pursuit of a smoke simulation in-distinguishable to one made from nature. Although, since video games are heavily restricted in time per frame, the most common smoke simulations and effects are produced with particle systems. Zhang[1].

1) Particle Systems: Reeves[2] first conceptualised the idea of a particle system, a graphical generation technology that can produce endless effects due to its modular nature. It involves managing a large collection of particles, which are small objects usually spheres or quads facing the camera. Behaviour can be added to these particles to produce the animators intended effect, such as a constant force or collisions.

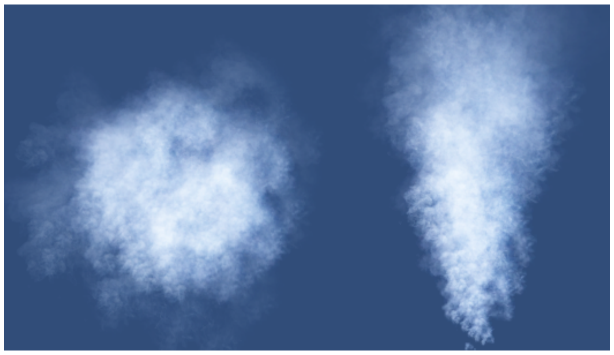


Fig. 1: Smoke effect made with Unity's Particle System, Stejskal[4]

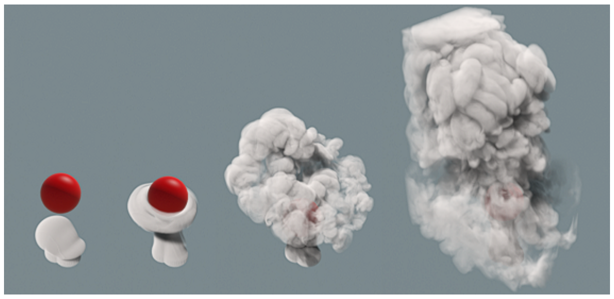


Fig. 2: Smoke flowing past a ball, modelled using BiMocq physically based fluid simulation method, image sourced from [11]

Another important feature of a particle system is the cycle of creation and clearing of particles, animators can define particle lifetimes and particle emission properties for further modification of the result.

Another reason games use particle systems is because of their low performance cost, along with the capability to scale the performance by adjusting the number of particles. This is the reason why most games model smoke effects with a particle system. Dong[3] outlines a smoke effect created using a particle system, by spawning particles with an upward force with an additional wind force, which produces a plume of smoke moving at the set wind direction, with a high frame rate. Figure 1 is another example of a smoke cloud created with a particle system, in the popular game engine Unity. The effect certainly resembles smoke but leaves room for improvement with the unnatural dissipation of density and the unresponsiveness to any advection forces.

As with other methods of simulating smoke, optimisations can be made to particle systems. One of the options available is to offload part of the total computations onto the gpu, as the gpu excels at parallel processing it can compute many particles simultaneously. Xiao [5] shows this by doubling their frame rate for a firework particle system processed on the cpu, by moving the particle calculations to the gpu.

Although particle systems have high frame rates, they don't look natural and can detriment the high realism games are chasing. By using a physically based approach, smoke effects can much closely represent reality.

2) Physically Based Simulation: All modern methods of simulating realistic smoke are based on stable fluids [6], put forward by Stam 1999. His work modelled smoke as an incompressible fluid which would follow the Navier-Stokes equations, to calculate the density and velocity of the smoke within a finite space. The method involved simulating the smoke within a two-dimensional grid of cells, each would store the density of the smoke at that point. A fluid solver would then calculate the diffusion between cells at each time step. This method also included a velocity vector field for forces acting on the smoke at each point where the density is stored. Specifically, using a semi-Lagrangian approach to solve the advection of smoke across the velocity field.

The result was an impressive step towards realistic smoke simulations, producing a more natural flow to the smoke with less visual noise than one made with a particle system. The major improvement from its predecessors was the ability for larger time steps whilst keeping the fluid stable making it more appropriate for animators to use for visual effects. There was a major drawback to this approach however, the numeric dissipation of the semi-Lagrangian solver, where density would dramatically disappear and seem unnatural. This problem would be the focus of many future works.

Fedkiw et al 2001 [7] proposed a solution to solve the dissipation problem, by adding a vorticity confinement step to the solver, to preserve the lost smoke density. The vorticity confinement step calculated the lost density and pushes it back into the simulation. This resulted in the smoke featuring flowing-like formations on its edges, creating a more natural and sustainable result than the previous attempt.

Kim et al 2005 [8] pioneered an alternate solution, by introducing a method to solve the non-dissipation problem, using BFECC (back and forth error compensation and correction). This involved comparing the result from looking forward in time to the result of looking backwards, to estimate the error of dissipation. The error result is then used to adjust the data before advection to provide a more accurate output. Selle et al 2008 [9], proposed an improvement, by using a modified MacCormack method, they were able to improve performance by using the error result to correct the already advected forward data, thus saving an advection call each time step. These methods resulted in realistic turbulent elements appearing in the smoke simulation whilst also keeping the smoke unconditionally stable.

The two methods of vorticity confinement and back and forth error calculation were combined in a paper brought forward by Zhang 2015[10]. Zhang presents a novel approach to solving the dissipation problem by using parts of both previous attempts, he calculates the vorticity before and after advection to get the vorticity correction value. This is used to modify the intermediate velocity of the smoke in each time step, resulting in more accurate tracking of velocity and angular velocity. The visual improvements from this method to previous ones are the overall clarity of the smoke, as well as better turbulent features.

Qu et al 2019 [11], introduces a new approach by using mapping functions to track the smoke density. The algorithm still uses a grid to store the values and the same advection algorithm as other papers, however, they develop a mapping function to track the density across the grid. Similarly, with other approaches, they utilise a back and forth error correction method for accurate mapping. The result, figure 2, is a largely realistic smoke simulation. In comparison to a particle system approach, the smoke looks much denser within the cloud and also with less overall visual noise around the main body. Although much more apparent is the interesting natural turbulent features which occur.

B. Rendering

The latter simulations (2005 onwards) were rendered using sampling ray tracing which has changed from the earlier methods [6],[7] that used voxel grids. The advantage of using a ray tracer is that lighting can be calculated in the same pass as rendering.

1) Ray Tracing: The process of ray tracing has existed for centuries and there are many implementations, Whitted 1980 [12] proposed the first major graphics algorithm utilising ray tracing to illuminate a scene. This improved upon previous lighting works, Phong [13], Blinn [14], by accurately portraying reflections and refractions. Stochastic path tracing, Kajiya 1986 [15], made advancements by accounting for global light scattering which greatly improved environment lighting and became the most realistic renderer. His work utilised ray tracing to solve the rendering equation along light paths, with enough paths calculated it is possible to accurately estimate the scene lighting. The paper employs a Monte Carlo statistical technique to find which paths need to be calculated, considerably reducing the number paths traced. Video games are only just starting to using this technology because of ray-tracing specific hardware and optimisation advancements.

Purcell 2002 [16] utilises an advancement in graphics hardware to compute ray tracing, in parallel, on the gpu, this sees a large increase in performance over cpu driven approaches. Another widely used optimisation is Bounding Volume Hierarchies (BVH), Whitted [17], they greatly reduce the amount of ray tracing intersection checks by enclosing 3d objects in a bounding box and recursively grouping triangles inside, into smaller boxes. 2009 [18] provides an efficient BVH construction algorithm on the gpu which cuts a large amount of computation when run before the rendering pass. This optimisation method easily ties into ray tracing through volume densities (smoke).

Kajiya 1984 [19] introduces a method of graphically portraying objects represented by density grids, their work focuses on clouds but it can be extended to smoke. The method calculates the light scattering through the volume density objects. Perlin [20] implements a brute force approach, using ray marching to step through the density field accumulating the opacity of the ray. This approach is very costly to compute as each step has to recursively ray march to the lights for shading. Distance sampling, Pauly 2000 [21], is a technique that randomly samples light scattering along a ray, improved upon performance but produces visual noise as a trade-off. Kulla 2012 [22] introduces importance sampling, a technique which vastly improves visual quality without increasing performance overhead. The technique efficiently distributes the samples along each ray to areas most affected by the lights. A major optimisation is made by removing the second ray trace when calculating lighting, instead computing the transmittance function for the ray, to access any shadow ray without having to re-trace rays again. This technique produces incredibly realistic results at efficient speeds.

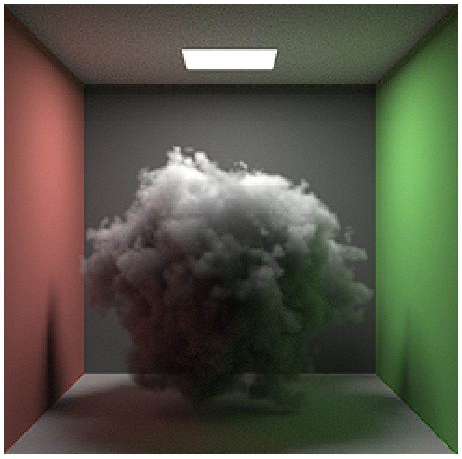


Fig. 3: Smoke rendered using path tracing, image sourced from [22]

2) Voxels: Voxels (volume elements) are used to visually represent 3d objects, the same as pixels represent 2d, by using a three-dimensional grid to store the object's spatial volume. The bulk of research on voxels lies within the medical field, MRI and CT scans use voxel grids to store their data [23]. However, this paper will focus on voxel's use in video games and real-time rendering, as their popularity in these fields has been quickly rising [24].

Texture-based volume rendering, Ikits [25], is a simple approach to visualising a voxel grid. It splits the grid into a series of 2d slices, generating a texture from each. The textures are then applied to a series of planes, perpendicular to the view angle, each representing a cross-section of the voxel grid. They are rendered back to front, using a rasterization pipeline with alpha blending. Lighting, Kniss [26], is computed with a first pass over the grid, calculating how much light reaches each voxel. This approach is faster than comparable ray casting methods as it uses the rasterization pipeline which graphics hardware is majorly optimised for. However, its pitfalls also appear from rasterization, as simply rendering a lot of planes will create a large number of fragments. Li [27] limits this by skipping empty voxels and occluding unseen ones, greatly improving performance.

Although computationally efficient texture-based rendering does not render voxels as cubes, this is achieved in another approach. Crassin [28] introduces a method of efficiently rendering large numbers (several billion) of voxels using an optimised data structure. The technique stores groups of voxels as nodes in an octree structure, with group size usually being 32^3 and referred to as a brick, this allows for approximations to be made of the voxels inside. Improving performance by only rendering the voxels bordering empty space. This paper uses a ray tracing render pipeline, resulting in reasonable performance and impressive visuals. Miller [29] publishes an alternate approach, rendering with rasterization instead, improving on the performance by utilising the rasterization optimised hardware.

The work most closely related to this paper is Zadick's [30] fluid simulation in a voxel game engine. Zadick implements a physically based water simulation into a rasterization voxel engine, with increased performance by pushing computations onto the gpu. They aimed to improve on previous water effects in voxel games, such as Minecraft, but their results don't justify the large performance overhead. The visuals do represent water as a stylised 'blocky' look but couldn't scale up to any realism due to performance restrictions.

III. Methodology

This paper takes a positivism stance on research using deduction to verify all claims made. Claims which are based on results from quantitative experiments ran on the artefact, using frame-time as the main metric. Frame-time is being used in this analysis because it is the key performance indicator in gaming applications, as frame-time shrinks frame-rate increases along with the overall visual experience.

It is important to note that although both rendering methods are rendering the same smoke simulation, minor visual differences will occur. This subjective quality will be ignored because, for games, visual quality will not matter if smoke rendering consumes the majority of frame-time. Thus, performance is the first step towards physically based smoke effects in games.

A. Research question

The research question this paper's addressing is: ***Can voxel rendering, of a smoke simulation, improve run-time performance over ray-casting in a video game environment?*** The hypotheses drawn from this question are shown in table I.

The project's hypotheses are tested by comparing two distinct methods of rendering a pre-computed physically based smoke simulation. The basis of the comparison is the mean average time to render each frame. The resolution of the smoke simulation increases, to identify any trends of resolution affecting performance.

B. Hypotheses

Table I shows the hypotheses tested in this experiment. The two hypotheses aim to compare the performance of both rendering methods, to determine the best renderer, and to measure performance inside a pseudo game environment measuring performance in a real-world scenario.

As this is a comparison of rendering methods, to accurately test these hypotheses, an environment is created that isolates the rendering step as the main bottleneck. Thus, the smoke itself will be pre-computed and inputted into the artefact on start-up. Another benefit from using this testing strategy is that the simulation resolution becomes experiment parameter and is used to produce a more robust comparison.

C. Experiment Design

As laid out before, the experiment's purpose is to measure the performance difference between two rendering techniques. To get valid data, the experiment gathers the mean frame time of both techniques, at different levels of resolution, and with or without background scenery. Simulation resolution defines the grid size used in the smoke simulation and by increasing it, observations can be made about how well the rendering technique scales. Specifically, the simulation is run at 5 levels of resolution, starting at 16x16x16 and ending at 256x256x256.

The inclusion of the background scenery variable is important, as it further evidences the performance claim, and produces data in a typical game environment. It is achieved by simply adding an extremely dense mesh into the scene, representing the triangle count seen in a typical game frame. From the data gathered, arguments can then be made for physically based smoke's use in the industry.

A single test consists of running the artefact for sixty seconds at the given resolution and scene configuration, with the average frame-time, for rendering smoke, being recorded. The test is repeated five times for each configuration.

The experiments are run on a single computer, with average consumer hardware, Steam hardware survey [31]. Specifically: AMD Ryzen 7 3800x, AMD Radeon rx580, 16GB memory.

D. Data Management

To collect the time taken to render a single frame, markers are added before and after the rendering step, including any calls to the graphics-API which can be affected by the renderers. The markers are used to send the times to a separate data collection class, which calculates the time difference and adds to a cumulative total, as well as increasing the total frames rendered. From this data, the average frame time is calculated, and including other pieces of data collected, each experiment outputs:

- Average time to render a frame
- Smoke simulation size
- Scene configuration
- Date of experiment
- Total running time

Each time the artefact is run, the data is outputted as a single .txt file, which is manually added into an excel spreadsheet stored locally. A back-up of the data is also stored on OneDrive, to ensure its security.

Once enough data was gathered, five sets of results for each simulation configuration, the next step was to use statistical analysis to confirm the outcome of each hypothesis. This is achieved by using R to perform a regression test on the data, for both hypotheses, to determine the relationship of simulation resolution and the rendering method on the frame-time. Scene configuration will be included in the regression test for hypothesis 2. Both the scene configuration, and rendering method, will be encoded into a binary value for the tests, so that it can differentiate between their effects on frame time. Refer to appendix A, for the specific regression model.

TABLE I: Hypothesis

	Hypothesis	Null Hypothesis	Data Source
1	The amount of time to render a frame of a smoke simulation using a voxel grid will, on average, be less than an implementation using ray casting.	The amount of time to render a frame of a smoke simulation using a voxel grid will, on average, be greater than an implementation using ray casting.	Frame-time metrics from the artefact
2	The amount of time to render a frame of a smoke simulation, with background scenery, using a voxel grid will, on average, be less than an implementation using ray casting.	The amount of time to render a frame of a smoke simulation, with background scenery, using a voxel grid will, on average, be greater than an implementation using ray casting	Frame-time metrics from the artefact

Along with how factors affect frame-time, a regression test will also output a p-value for each factor of frame time, which can be used to ensure that its relationship on frame-time is statistically significant and not just random noise. The overall test's p-value, if significant, is used to confirm the hypothesis' result, which in turn will be determined by analysing the outputted regression model.

The outputted model will show the magnitude of each factor's effect on frame-time. Meaning a single value representing the relationship on frame-time for both rendering methods is outlined, which is used to determine if voxel rendering is slower or faster than ray-casting.

E. Ethical Considerations

This paper adheres to the BCS code of conduct [32] and Falmouth University's "Research and Innovation Integrity and Ethics" guidelines [33]. Ethical issues regarding participants, Nuremberg Code [34], are minimal with the research of this paper, as no other people are involved. The process of the experiment only required repeatable testing of the artefact on one machine. This did, however, raise some risk concerns with desk screen over-use, which was minimised by following proper guidance [35].

Furthermore, the research is not unethical in nature, as it just adds to the graphical simulation field. But since smoke can be hazardous, and potentially put lives in danger, it is important for this research and accompanying artefact to be freely available, for industry to, if judged an improvement, implement into their health and safety simulations. This is achieved via an open-source public repository, refer to appendix E.

IV. Artefact Design

Since the experiment is measuring the speed of two different rendering methods, a generic game engine was unfit because of the inability to fully adjust its graphics pipeline. For this reason, the artefact framework was built in a proprietary engine made with C++ and using open-gl for the graphics API. This allowed for low level access to the cpu, gpu, and memory, giving full control over optimisation and performance. The lack of bloat and excess features in a proprietary approach further improved the accuracy and reliability of results. As frame-time is key for the experiment, being able to log performance at any point in the program helped to find any bottlenecks.

The choice of using open-gl as the graphics API over Vulkan or DirectX, was made because of its wider compatibility and superior integration with C++. Additionally, open-gl easily ports to mobile devices through the opengl-ES interface, which is important as it provide performance metrics relevant to the large amount of mobile game developers.

One of the requirements for the artefact, was to switch rendering methods without affecting any other factor of performance. Thus both rendering pipelines were built into the same application, and on startup the program asked for the experiment input parameters:

- Rendering method
- Simulation resolution
- Scene configuration
- Result output location

The inputted settings are loaded into the scene and the experiment starts. During runtime, the user is able to freely move the camera around the scene, but when collecting data this was locked to minimise performance differences.

In terms of application design, the artefact utilises the advantages of an object-orientated approach, using C++ to effectively split the code into maintainable classes. Making sure to follow the SOLID programming principles laid out by Martin [36], outlining effective class design and interactability.

The program's main classes are smoke-simulation, voxels, and ray-casting. By splitting the program's core functionality into separate classes, changes could be made independently, and effective software practices, including agile, can be achieved, Martin [37]. The following sections discusses key artefact components' implementation details.

1) Smoke Simulation: The specification for this class, in context of the experiment, was solely to provide the smoke's data to the renderers, more specifically the simulation's current density grid. To achieve this, the smoke class first had to simulate the smoke, store it to a file, and finally, in the experiment, read the data back into memory frame-by-frame.

For an adequate smoke simulation implementation, the method aimed to follow Stam's stable fluid approach [7], using vorticity confinement for increased natural flows. This technique stores the simulation using multiple flat grids to represent smoke's physical elements in 3D space, including a current and previous grid for density, each direction of velocity, totalling eight grids. The size of each grid is calculated using the inputted simulation resolution, which represents the smoke's width in each dimension, meaning the grids total size is resolution^3. However, the only grid needed for rendering is the smoke's current density, which specifies how much smoke is in each point of space.

To model the smoke's behaviour, the grids are sent through multiple functions every time step. The two central functions are diffusion and advection. Diffusion models how a grid's values spread out over time, achieved by taking a current and previous grid as input, and looping over each value to calculate how much is transferred to and from neighbouring cells. Advection models how a grid's values move according to the current velocities, similarly implemented by looping through the given grid's values and using the current velocities to calculate where the value should move given the time-change. As a measure of reusability, both functions could take any grid as an input, because they are later used with by other grids than just density. These two functions were all that was needed to manipulate density, controlling how the smoke spreads and moves, producing a simple simulation, however, to model the complex naturalistic flow of realistic smoke, manipulation of velocity was required.

The velocity step first followed the same process per frame as density, each directional grid's values spread out using diffusion and then the velocity field moves along itself using advection. This gives the smoke its turbulent behaviour by changing the velocity from being static to flowing like density would. However, since advection is being used by the grids and they are moving themselves according to their own velocity, numerical faults can occur. Thus, each velocity grid is sent through another function named projection, which ensures the total velocity remains constant. Lastly, the velocity step included the vorticity confinement technique, which adds small natural 'curls' in the velocity field, further improving the smoke's realism. This behaviour was implemented by passing the velocity grids to another function, similarly looping through the grids to add vorticity.

The density and velocity steps are encapsulated by a single update function, which can be publicly called through the smoke object. The density grid's pointer is also publicly available, thus every class which needs the smoke's density can keep a reference. This design provides minimal class interactions as the only function needed to be called each frame is the smoke's update.

That concludes the smoke's real-time simulation functionality, figure 4 shows the smoke being rendered by ray-casting and using approximating shading method. The approximate shading model, darkens the smoke at areas of high density.

But the experiment requested the smoke to be loaded from a file, to save large amounts of computation per frame. Following the same design principles, updating the smoke each frame, from a file, should be achieved through one public function. To achieve these requirements a new class was created, solely used by smoke, to act as an API for saving and reading smoke simulations to files.

The class has two responsibilities, saving and reading simulations, and uses an initialisation function to start both, either creating a file or opening an existing file. For saving simulations, the class is set-up to take the smoke's density data frame-by-frame and it will consecutively save the data to the file. For reading, once the file is opened, the class is set-up to read-in the smoke's density data frame-by-frame.

The smoke's file format is relatively simple, firstly a 32-byte header is saved to store the simulation information: size, frame-count, and compression settings. Then as smoke's data

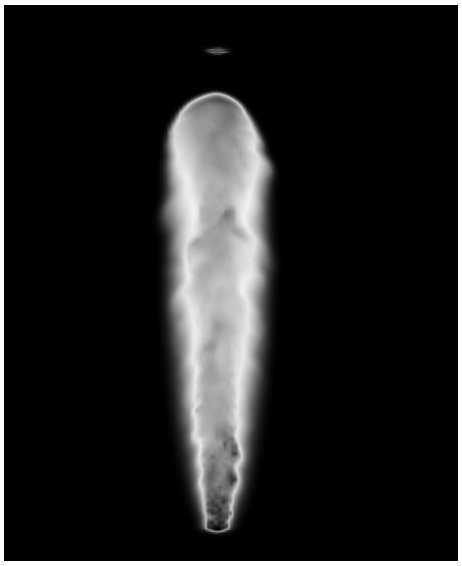


Fig. 4: Smoke, with a resolution of 128, rendered using ray-casting, and approximating shading based on density

is sent in it is compressed, by equally splitting the grid into smaller blocks and only storing a block if its values have changed since the previous frame. Along with the frame's data, a header is also needed to specify which blocks need updating, when reading the file. This cuts a large amount of data from being stored, because density is usually concentrated to a small area in the simulation, leaving the rest of the grid un-changed each frame. An example of the compression efficiency is: a 150 frame, 256 wide, simulation's file totalling $1.5gb$, cutting ≈650% of the estimated $10gb$ file size.

2) Voxel Rendering: The specification for voxel rendering is to take a grid of density values and draw the corresponding smoke to the screen, using 3D voxels. This functionality was built into a single class, which utilises a vertex and fragment shader to communicate with the graphics API, to draw the instanced voxels to the screen.

Instancing is a graphical technique which can efficiently redraw an object to the screen without updating any buffers, meaning one voxel's data can be sent to the GPU's memory and be used to draw all of the million voxels needed. This saves a large amount of frame-time, by cutting the number of data transfers to the gpu from increasing with resolution, to just one per frame.

The voxel rendering class is very simple, on creation it generates a voxel mesh, using a cube's geometry, and creates a gl-object which is used to draw the voxel to the screen. Then the public draw function takes the smoke's density, packages it as a vertex buffer to be sent to the shaders, and then uses an instanced draw call to draw all the voxels to the screen. This means again, just a single interaction point for an object to enact its functionality, per frame.

Lastly, since instancing is used, the shaders complete the bulk of computation by determining where each voxel is drawn and its colour. This is partly achieved in the vertex shader, by using the instance id, a unique index given to each instanced voxel from 0 to N, to determine the voxel's position in the

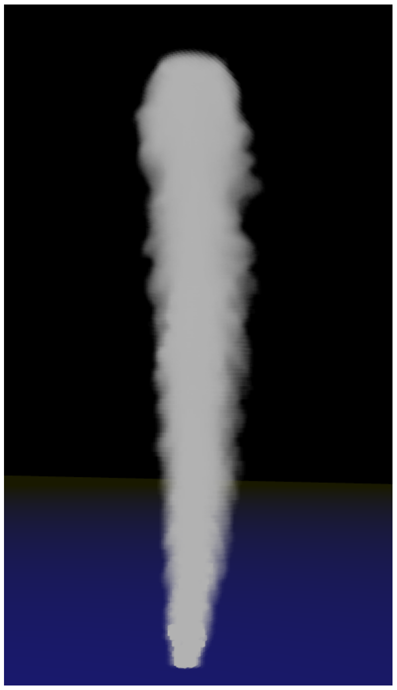


Fig. 5: Un-shaded smoke, with a resolution of 128, rendered using voxels

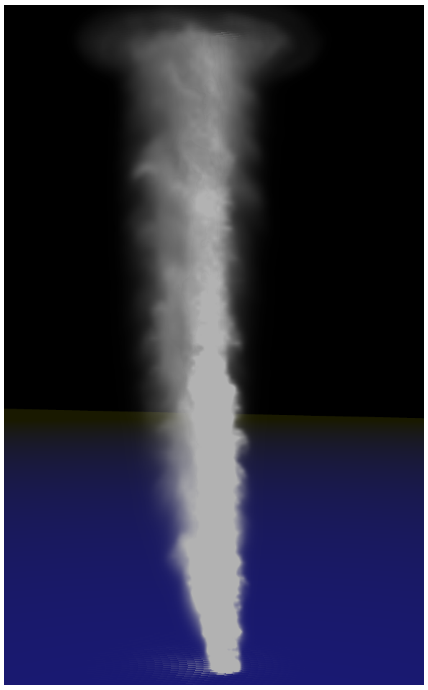


Fig. 6: Un-shaded smoke, with a resolution of 128, rendered using ray-casting

3D grid. Then in the fragment shader, each voxel's colour is calculated using the given density, by altering its opacity. Figure 5 shows the un-shaded result of voxel rendering.

3) Ray-casting Rendering: The specification for ray-casting is mostly the same as voxels, take a density grid as input and output the smoke to the screen, but instead, utilising a volume sampling ray-caster. Similarly to voxel rendering though, only one class and one set of shaders are needed and the class itself follows the same format, only one draw call, with the density given as input, is used to render the smoke.

Where the renderers differ, is how they represent density, ray-casting uses only one cube to represent the bounding simulation space and colours it by sending a 3D texture to the fragment shader. This texture is generated in each frame's draw call, using the density data.

The technique used to colour the cube is called volume sampling, which involves sending rays through each fragment and then estimating the smoke's density along the ray by stepping through the 3D texture. This is implemented by the fragment shader, each fragment generates a ray from the view position into the current fragment on the cube. Then, starting from the fragment's position, the ray steps through the smoke, sampling at each point to accumulate an estimated density along the ray. finally, this estimated density can then be used to colour the fragment on the outside of the cube. Figure 6 shows ray-casting being used to render an un-shaded result of the smoke simulation.

A. Software Development Life Cycle

The software life cycle of the project followed a mixture of two approaches, starting with a waterfall process of incrementally developing base versions of the main components one after another:

1) Open-gl context
2) Voxel grid
3) Ray tracing
4) Smoke simulation
5) Data collection

Once completed, and each feature was working, an iterative approach was applied to gradually optimise each component using the methods researched in the related work section. This benefited development by allowing performance to be measured at each optimisation milestone, giving assurance of improvement, and data for any later performance claims.

These life cycles resulted in the artefact being built on a solid base and then progressively brought up to maximum efficiency.

In aid of the development process, version control was used, which predominantly backs-up the artefact online, providing a fail safe for any errors or file corruption. Additionally, version control also tracks any modifications made to the project over its lifetime, which can be used to highlight refactors for future maintainability. Refer to Appendix E for a link to the artefact's repository.

B. Validation and Verification

To ensure this paper's research validity, it followed the eight-point software quality model outlined by ISO/IEC 25010:2011 [38]. While some factors were disregarded because of the nature of the artefact, reliability was the main concern and is checked by verifying several aspects of the application. Most importantly making sure the smoke simulation is accurate, also testing the graphical interface, and finally verifying the data collection.

It's important for the smoke simulation to be accurate and reliable because the whole experiment relies on its data. Therefore, its functionality was heavily tested with unit tests, these were implemented using visual studio's unit testing framework. As the simulation was split into several functions acting on a single array of data, unit tests are perfect for verifying that each step was working correctly. A generic unit-test structure includes, stating input density, applying the specific smoke function, for example advection, then checking the updated smoke density against an expected result. The test fails if any differences are detected between the expected and actual density grids. Along with the simulation validation, writing and reading smoke to files was also unit tested, ensuring the simulation is not modified during these processes. Examples of all these unit tests and their results can be found in appendix B.

For graphical reliability, integration tests [39] were implemented. Specifically incremental testing, which involved testing integration with the graphics API (open-gl) each time a new feature was added. This was deployed to verify the reliability between the graphical output and the application, by checking that the graphical window is displaying exactly what the program specified. These tests were conducted by running the artefact through a set of specified outcomes, by a controlling class, each test is visually checked, manually, and marked passed or failed in an accompanying document. Refer to Appendix C for an example integration test.

Other artefact components which are unsuitable for unit and integration tests, were validated by static-code analysis and manual code inspections. For example, data collection, another vital artefact component, was thoroughly checked by code walkthroughs verifying its markers were placed correctly, to ensure it was precisely isolating the renderers performance.

V. Results

The data gathered from the experiments was collected into an excel spreadsheet, stored on the cloud, and later imported into R-Studio. R was then used to visualise the data and analyse any statistical significance for each hypothesis. Refer to appendix A for r code.

A. *Hypothesis 1: The amount of time to render a frame of a smoke simulation, using a voxel grid will, on average, be less than an implementation using ray-casting*

This hypothesis focuses solely on comparing the runtime performance of voxel rendering compared to ray-casting for smoke simulations. Figure 7 visually represents the frame time difference between both renderers at increasing simulation resolutions, specifically the under-performance of voxel based rendering. At around a resolution of 75, voxel rendering diverges from ray-casting, increasing in time taken to render each frame. This deficit only expands as the resolution get larger, which strongly indicates that voxel rendering is slower than ray-casting. A statistical analysis was performed to confirm this analysis.

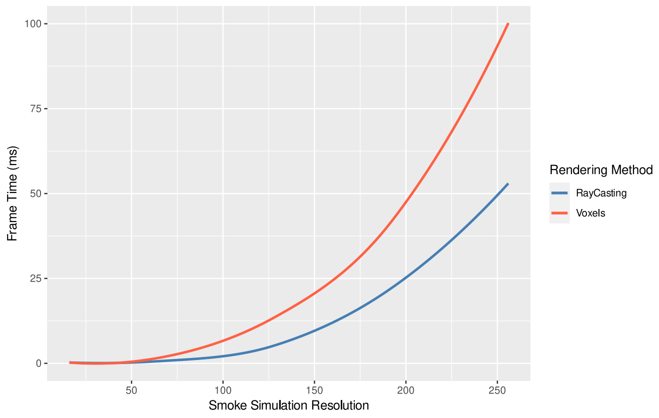
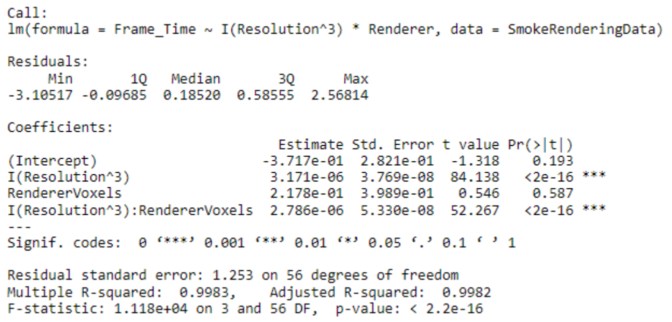


Fig. 7: Line chart comparing frame time for both rendering methods



```
Call:
lm(formula = Frame_Time ~ I(Resolution^3) * Renderer, data = SmokeRenderingData)

Residuals:
     Min       1Q   Median       3Q      Max 
-3.10517 -0.09685  0.18520  0.58555  2.56814 

Coefficients:
                                 Estimate Std. Error t value Pr(>|t|)    
(Intercept)                    -3.717e-01  2.821e-01  -1.318    0.193    
I(Resolution^3)                 3.171e-06  3.769e-08  84.138   <2e-16 ***
RendererVoxels                  2.178e-01  3.989e-01   0.546    0.587    
I(Resolution^3):RendererVoxels  2.786e-06  5.330e-08  52.267   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 1.253 on 56 degrees of freedom
Multiple R-squared:  0.9983,	Adjusted R-squared:  0.9982 
F-statistic: 1.118e+04 on 3 and 56 DF,  p-value: < 2.2e-16
```

Fig. 8: Regression analysis on smoke rendering data

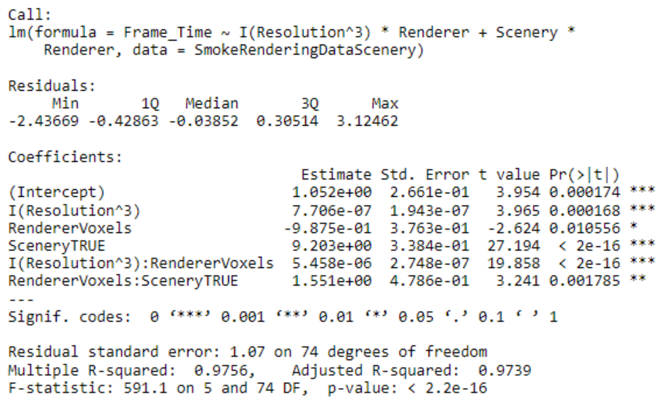
Figure 8, shows the results of a linear regression analysis performed on the data. By looking at the coefficients, a term relating voxel-rendering to resolution has a near zero p-value, which highly indicates additional frame time is added when using voxel rendering. Combined with an overall p-value substantially below 0.05, the model can be used to definitively satisfy the null hypothesis, voxel rendering takes longer to render a frame than ray casting. Given the high r-squared value, showing the model accounts for 99% of variance, an equation can be formed to accurately estimate frame-time:

$$ft = -0.37 + (3.2 \times 10^-6)R^3 + (2.9 \times 10^-6)R^3V \quad (1)$$

With ft: frame time, R: resolution, and V as a binary encoded value representing voxel rendering with 1 and a 0 for ray casting. The equation gives a baseline for frame time using resolution, and additionally adding costs when using voxel rendering. The lone voxel rendering constant, from the regression model, is ignored due to the high p-value, meaning the model found no significance on frame-time from including a constant increase from voxel rendering.

B. *Hypothesis 2: The amount of time to render a frame of a smoke simulation with background scenery, using a voxel grid will, on average, be less than an implementation using ray-casting*

This hypothesis seeks to test the performance of both rendering methods in a pseudo video game environment, by using background scenery totaling of around 50 million triangles.



```
Call:
lm(formula = Frame_Time ~ I(Resolution^3) * Renderer + Scenery *
    Renderer, data = SmokeRenderingDataScenery)

Residuals:
     Min       1Q   Median       3Q      Max
-2.43669 -0.42863 -0.03852  0.30514  3.12462

Coefficients:
                              Estimate Std. Error t value Pr(>|t|)
(Intercept)                  1.052e+00  2.661e-01   3.954 0.000174 ***
I(Resolution^3)              7.706e-07  1.943e-07   3.965 0.000168 ***
RendererVoxels              -9.875e-01  3.763e-01  -2.624 0.010556 *
SceneryTRUE                  9.203e+00  3.384e-01  27.194  < 2e-16 ***
I(Resolution^3):RendererVoxels  5.458e-06  2.748e-07  19.858  < 2e-16 ***
RendererVoxels:SceneryTRUE   1.551e+00  4.786e-01   3.241 0.001785 **
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 1.07 on 74 degrees of freedom
Multiple R-squared:  0.9756,    Adjusted R-squared:  0.9739
F-statistic: 591.1 on 5 and 74 DF,  p-value: < 2.2e-16
```

Fig. 9: Regression analysis on smoke rendering data with background scenery

Figure 9 shows the resulting linear regression model, which uses both the regular data and rendering with background scenery data. This model differs from the previous by adding an additional term measuring the relationship between both renderers and rendering background scenery.

Similarly, this model has an incredibly low p-value and high r-squared value, which means the model can be used to accurately predict frame time. Examining the coefficients, each is statistically significant, including, the term relating rendering method and background scenery. This term shows that voxel rendering has additional costs to frame-time, when there is background scenery, compared to ray-casting. Therefore, the null hypothesis can be accepted, voxel rendering, on average, takes longer to render a frame with background scenery, than ray-casting.

The model presents the effect of rendering with background scenery as a flat $\approx$10ms increase on frame time, with an additional 1.5ms if rendering with voxels. Interestingly though, this model also shows a decrease in frame-time consumption from the resolution term, irrespective of scenery, than the hypothesis 1 test. Another difference from the previous model, is the voxel rendering term being significant and actually decreasing frame time by $\approx$1ms, although this gets cancelled out when rendering background scenery which adds $\approx$1.5ms. However this does suggest at low resolutions, when rendering without scenery, voxel rendering would perform better than ray-casting.

VI. Discussion

This experiment aimed to test the viability of using physically based smoke simulation in video games through measuring the performance of two distinct rendering methods. The results conclusively show ray-casting as the better performing renderer, by frame-time metrics, from both null hypotheses being accepted. It's harder to explicitly determine viability for video games, because of the vast differences between each game, however this experiment resulted in incredibly useful and usable information to help developers choose whether using a physically based simulation is viable for their own game.

The first relevant finding from the results is the graph generated. Figure 7 visually portrays the exponential relationship of smoke simulation resolution to frame-time. This relationship was expected for voxel rendering, due to individually rendering each cell of smoke, and which the total amount of cells increases exponentially with resolution. However, ray-casting only renders one cube, colours it inside the fragment shader, and still has an exponential correlation to frame time. So despite both renderers performing widely different calculations each frame, they both follow similar frame time curves. This suggests that the frame time is being bottle-necked by an underlying computation shared between the two, this could be passing the large amount of simulation data to the shaders each frame. For example, a resolution of 128 would require a grid of 2 million floats to be sent each frame. This would be a key area for future improvement.

The statistical analysis of hypothesis-1 confirmed the graph's indication of voxel rendering increasing frame-time over ray-casting. The model measured the relationship of resolution-cubed and the rendering method on frame-time. Verifying, with high accuracy, that resolution does inherently affect both rendering methods' frame time at an exponential rate. The coefficient for resolutions effect on frame time is low, but since resolution is cubed the effect on performance soon becomes drastic, this was expected as it's known from big-O notation that introducing exponential rates into algorithms heavily affect performance.

Another key take-away from the regression model is the exact effect of using voxel rendering, on frame-time. Interestingly the model doesn't find any significance of a constant difference between the two renderers, but instead finds high correlation when combined with the resolution. Essentially meaning, when using the voxel renderer the frame-time difference, from ray-casting, will be multiplied by the current resolution-cubed, which is far worse than a flat change because of the aforementioned exponential issues. This drawback seems to be inevitable when rendering with voxels as that is the nature of the renderer, drawing the smoke grid in a one-to-one mapping. Future work could look at targeted voxel drawing, only rendering the needed voxels and disregarding the rest, this has been done in similar work using an octree data structure.

The most useful finding from hypothesis 1's results, was equation 1. Because of the regression model's high accuracy, accounting for 99% of variance, an equation could be formed to accurately predict frame time, from resolution and the renderer type. The equation doesn't reveal anything new from the results, but formats them in a readable fashion. Which is incredibly useful for developers who need to calculate whether using a physically based smoke simulation fits their frame-time budget.

The purpose of hypothesis 2 was to determine the effect of rendering the simulation in a game environment, by adding 50 million triangles to the scene. The results concluded that voxel rendering, again, performed worse than ray-casting, by adding an additional frame-time when including background scenery. This was determined by observing the coefficient relating both, voxel rendering and scenery, being positive, meaning that additional frame-time is added when both are active.

The regression model did, however, show a slight flaw in the data collection. The model found that when rendering scenery, a flat 9ms is added to frame time. This would not be significant in the model if data collection truly isolated the smoke rendering, and just measured the effect scenery had on the smoke renderer not overall frame-time. Examining data collection, the flaw comes from including the call `glSwapBuffers` inside the timer. This is needed because it affects frame-time and can differ between both renderers, but also gets affected by other objects in the scene which is where the problem arises, distinguishing between differences on the renderers because of scenery or just the scenery. However, in this case, because of the previous hypothesis' test revealing a direct correlation between resolution and frame time, and the scenery being a flat increase regardless of resolution, and the model is highly accurate, it can be confirmed as independent and having no effect on smoke rendering. Though this issue should be addressed in further research.

Interestingly, this model differs slightly from the previous, by finding significance in a flat decrease of frame-time when rendering with voxels, but, on the other hand, doubling the resolution$\times$voxels coefficient. This would suggest at low resolutions, without scenery, voxel rendering performs better than ray-casting, until resolution reaches around 60 then voxel rendering will start to negatively diverge from ray-casting. The result probably occurred due to the smaller data-set, only using data up-to a resolution of 128, which would not yet encounter the large frame-time differences occurring, between the renderers, at higher resolutions.

The most important finding from hypothesis 2 is that rendering a physically based smoke simulation in a video game environment doesn't negatively affect its frame-time or the scene's, unless using voxel rendering which gives a slight frame-time increase. This means that a developer does not have to consider the smoke affecting the scene's existing performance, only the performance of rendering the smoke itself. Being performance independent is vital for physically based smoke simulations to be considered viable for games.

VII. Limitations

Due to time constraints, a suitable smoke shading algorithm was not implemented for either rendering method. This limits any claims for viability in video games, since shading is an essential component for any real-world applications, video games included. However, the shading algorithm would not heavily impact performance and would share many similarities between the two renderers, thus any claims about their comparative performance would remain valid.

Another shortcoming due to time constraints was the renderers optimisations. Although the renderers are optimised commendably, they do not encompass the most recent advances made in the field, which were researched in the related work section. At this time, both renderers are at their most basic, functionally sound but perhaps over-working, which does mean that one is not getting optimised over the other giving some credibility to the results. But for an accurate determination of the best rendering method, both need to be optimised fully from further development.

Lastly, the performance testing of the smoke's implementation within a game environment could have been improved by using a industry-standard game engine. The only engine which fits this purpose, whilst being realism orientated and utilising C++, is Unreal Engine. This option was unavailable due to lack of experience.

VIII. Further Research

Further developments in this area of research should first focus on addressing this paper's limitations, the most important being shading. For voxels a simple approach would be to follow the two-pass algorithm laid out in Stam [7], and simple shading for ray-casting would be to follow Pauly's [21] method, stepping through the smoke to track the ray's opacity. But for cutting-edge shading, research should aim to replicate the path-tracing and scattering found in Kulla [22].

Another future development would be to improve upon this paper's voxel rendering implementation. One approach would be to combine this paper's method with the research done by Crassin [28] on giga-voxels. Utilising this paper's research on smoke rendering with the larger voxel rendering capability of giga-voxels, would achieve drastic performance gains as well as portraying an accurate smoke simulation.

A specific area of improvement for both rendering methods, which was mentioned earlier in the discussion section, is the performance bottle-neck of sending the smoke data to the shaders each frame. There is a certain lack of research in this area, as previous work on smoke does not focus on real-time rendering. Solving this problem would be certain to improve performance across the board. A solution could involve a similar approach to the compression algorithm for saving smoke to files, this is because most of the smoke's grid does not change between frames only certain sections need to be updated. Thus there's an opportunity to cut the amount of data transferred to the shaders, done efficiently would largely improve performance.

A different direction of future research could be to comprehensively measure and analyse the effect on performance from adding smoke into multiple existing video games. Specifically with the intention of identifying any and all weaknesses of rendering smoke in real-world scenarios. Further research could then be conducted to address these weaknesses, thus improving the smoke's performance in a game environment. Repeating this approach is certain to lead smoke rendering to the point of video game viability.

IX. Conclusion

To conclude, this paper compared the performance of different rendering methods for physically based smoke simulations, with the intention of providing an alternate visual improvement to the popular particle-system based smoke effects currently used in games.

Current smoke simulation techniques and rendering methods were reviewed, and a standard smoke implementation was developed in a graphical artefact made within a proprietary

C++ engine. Two rendering methods were tested, ray-casting using a volume sampler and an instanced voxel renderer. Both renderer's performance was measured at varying smoke simulation configurations, including a pseudo video game environment, using a metric of the average time to render each frame.

The experiments confidently concluded that ray-casting performed better than voxels across the board. Although showing potential at low simulation sizes, voxel rendering suffers almost twice the performance cost from increasing the resolution, which scales exponentially and soon becomes a large performance burden. Other tests proved the smoke's compatibility inside game environments with their performance being independent to the game's existing scene. Overall the research shows promise for physically based smoke, rendered using ray-casting, to be adopted by games pushing the limits of visual realism.

References

[1] B. Zhang and W. Hu, "Game special effect simulation based on particle system of unity3d," in *2017 IEEE/ACIS 16th International Conference on Computer and Information Science (ICIS)*, pp. 595–598, 2017.

[2] W. T. Reeves, "Particle systems—a technique for modeling a class of fuzzy objects," *SIGGRAPH Comput. Graph.*, vol. 17, p. 359–375, jul 1983.

[3] W. Dong, X. Zhang, and C. Zhang, "Smoke simulation based on particle system in virtual environments," in *2010 International Conference on Multimedia Communications*, pp. 42–44, 2010.

[4] J. Stejskal, "How to create realistic smoke in unity," 2015. Available at: http://johnstejskal.com/wp/how-to-create-realistic-smoke-in-unity/.

[5] H. Xiao and C. He, "Real-time simulation of fireworks based on gpu and particle system," in *2009 First International Workshop on Education Technology and Computer Science*, vol. 1, pp. 14–17, 2009.

[6] J. Stam, "Stable fluids," in *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pp. 121–128, 1999.

[7] R. Fedkiw, J. Stam, and H. W. Jensen, "Visual simulation of smoke," in *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, p. 15–22, Association for Computing Machinery, 2001.

[8] B. Kim, Y. Liu, I. Llamas, and J. Rossignac, "Flowfixer: using bfecc for fluid simulation," in *Proceedings of the Eurographics Workshop on Natural Phenomena, NPH 2005, Dublin, Ireland, 2005.*, pp. 51–56, 01 2005.

[9] A. Selle, R. Fedkiw, B. Kim, Y. Liu, and J. Rossignac, "An unconditionally stable maccormack method," *J. Sci. Comput.*, vol. 35, pp. 350–371, 06 2008.

[10] X. Zhang, R. Bridson, and C. Greif, "Restoring the missing vorticity in advection-projection fluid solvers," *ACM Trans. Graph.*, vol. 34, jul 2015.

[11] Z. Qu, X. Zhang, M. Gao, C. Jiang, and B. Chen, "Efficient and conservative fluids using bidirectional mapping," *ACM Trans. Graph.*, vol. 38, jul 2019.

[12] T. Whitted, "An improved illumination model for shaded display," *Commun. ACM*, vol. 23, p. 343–349, jun 1980.

[13] B. T. Phong, "Illumination for computer generated pictures," *Commun. ACM*, vol. 18, p. 311–317, jun 1975.

[14] J. F. Blinn, "Models of light reflection for computer synthesized pictures," *SIGGRAPH Comput. Graph.*, vol. 11, p. 192–198, jul 1977.

[15] J. T. Kajiya, "The rendering equation," *SIGGRAPH Comput. Graph.*, vol. 20, p. 143–150, aug 1986.

[16] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan, "Ray tracing on programmable graphics hardware," *ACM Trans. Graph.*, vol. 21, p. 703–712, jul 2002.

[17] S. M. Rubin and T. Whitted, "A 3-dimensional representation for fast rendering of complex scenes," *SIGGRAPH Comput. Graph.*, vol. 14, p. 110–116, jul 1980.

[18] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast BVH Construction on GPUs," *Computer Graphics Forum*, 2009.

[19] J. T. Kajiya and B. P. Von Herzen, "Ray tracing volume densities," *SIGGRAPH Comput. Graph.*, vol. 18, p. 165–174, jan 1984.

[20] K. Perlin and E. M. Hoffert, "Hypertexture," in *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '89, (New York, NY, USA), p. 253–262, Association for Computing Machinery, 1989.

[21] M. Pauly, T. Kollig, and A. Keller, "Metropolis light transport for participating media," *Rendering Techniques*, vol. 2000, 11 2000.

[22] C. Kulla and M. Fajardo, "Importance sampling techniques for path tracing in participating media," *Comput. Graph. Forum*, vol. 31, p. 1519–1528, jun 2012.

[23] R. Spin-Neto, E. Gotfredsen, and A. Wenzel, "Impact of voxel size variation on cbct-based diagnostic outcome in dentistry: a systematic review," *Journal of digital imaging : the official journal of the Society for Computer Applications in Radiology*, vol. 26, 12 2012.

[24] K. Gao, J. He, and Y. Qi, "A relevant research on the establishment of a voxel gaming world," in *2018 IEEE International Conference on Consumer Electronics-Taiwan (ICCE-TW)*, pp. 1–2, 2018.

[25] M. Ikits, J. Kniss, A. Lefohn, and C. Hansen, "Volume rendering techniques," in *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics* (R. Fernando, ed.), Pearson Higher Education, 2004.

[26] J. Kniss, S. Premoze, C. Hansen, P. Shirley, and A. McPherson, "A model for volume lighting and modeling," *IEEE Transactions on Visualization and Computer Graphics*, vol. 9, no. 2, pp. 150–162, 2003.

[27] W. Li, K. Mueller, and A. Kaufman, "Empty space skipping and occlusion clipping for texture-based volume rendering," in *IEEE Visualization, 2003. VIS 2003.*, pp. 317–324, 2003.

[28] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann, "Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering," in *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, I3D '09, (New York, NY, USA), p. 15–22, Association for Computing Machinery, 2009.

[29] M. Miller, A. Cumming, K. Chalmers, B. Kenwright, and K. Mitchell, "Poxels: Polygonal voxel environment rendering," in *Proceedings of the 20th ACM Symposium on Virtual Reality Software and Technology*, VRST '14, (New York, NY, USA), p. 235–236, Association for Computing Machinery, 2014.

[30] J. Zadick, B. Kenwright, and K. Mitchell, "Integrating real-time fluid simulation with a voxel engine:," *The Computer Games Journal*, vol. 5, 09 2016.

[31] Valve, "Steam hardware and software survey: November 2022," 2022. Available at: https://store.steampowered.com/hwsurvey/Steam-Hardware-Software-Survey-Welcome-to-Steam.

[32] British Computer Society, "Bcs code of conduct," 2022. Available at: https://www.bcs.org/membership-and-registrations/become-a-member/bcs-code-of-conduct/.

[33] Falmouth University, "Research & innovation integrity & ethics," 2022. Available at: https://www.falmouth.ac.uk/research/research-ethics-integrity.

[34] A. Mitscherlich and F. Mielke, "The nuremberg code (1947)," *BMJ*, vol. 313, no. 7070, p. 1448, 1996.

[35] Health & Safty executive GOV, "Working safely with display screen equipment," 2023. Available at: https://www.hse.gov.uk/msd/dse/.

[36] R. C. Martin, "Design principles anddesign patterns," 2000.

[37] R. C. Martin, *Agile Software Development*. Pearson Education, 2003.

[38] I. 25010:2011, "System and software quality models," standard, International Organization for Standardization/International Electrotechnical Commission, 2011.

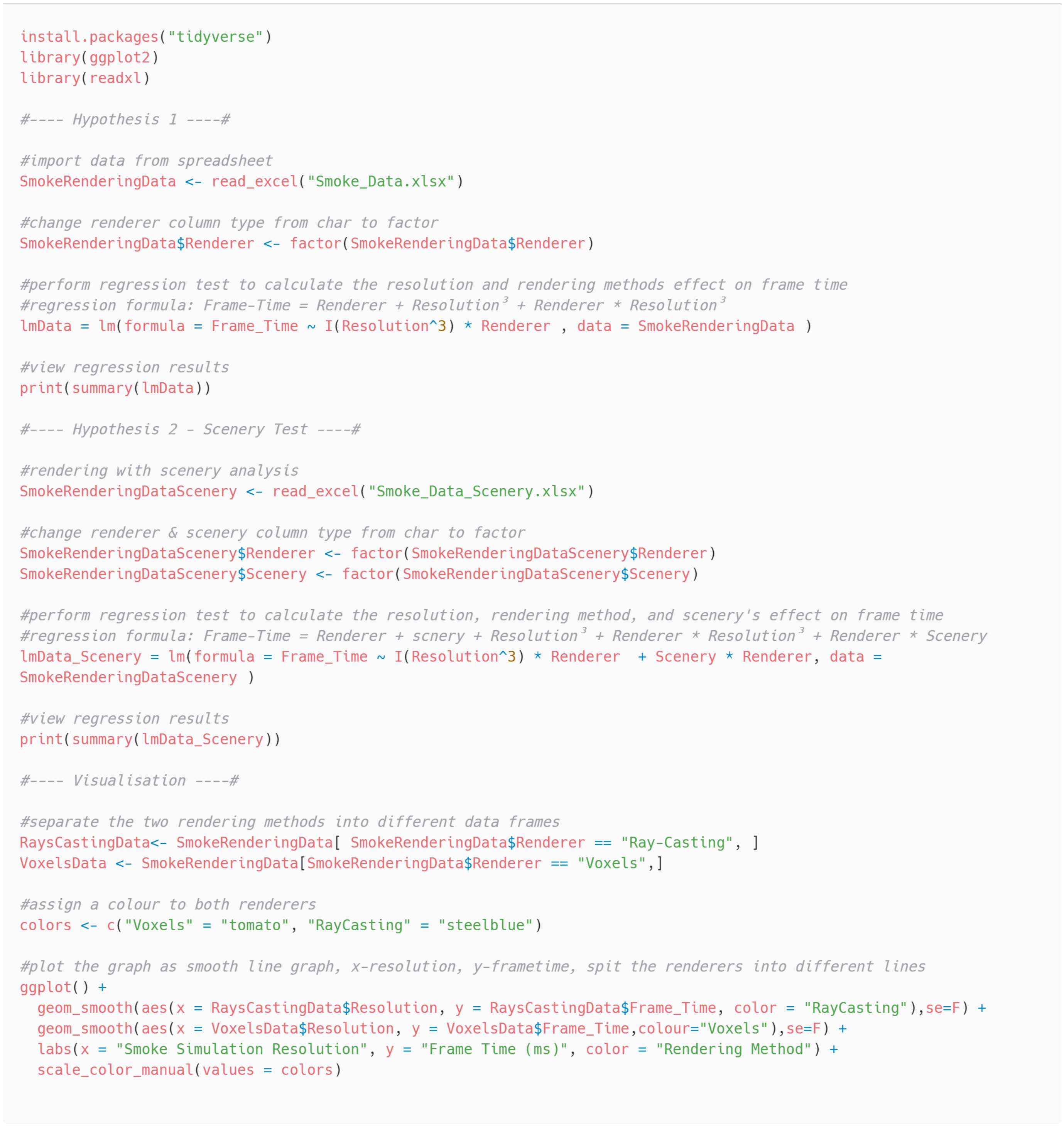
[39] H. Leung and L. White, "A study of integration testing and software regression at the integration level," in *Proceedings. Conference on Software Maintenance 1990*, pp. 290–301, 1990.

[40] M. S. Nabizadeh, S. Wang, R. Ramamoorthi, and A. Chern, "Covector fluids," *ACM Trans. Graph.*, vol. 41, jul 2022.

[41] Y. Deng, Y. Ni, Z. Li, S. Mu, and W. Zhang, "Toward real-time ray tracing: A survey on hardware acceleration and microarchitecture techniques," *ACM Comput. Surv.*, vol. 50, aug 2017.

[42] P. Clarberg, S. Kallweit, C. Kolb, P. Kozlowski, Y. He, L. Wu, E. Liu, B. Bitterli, and M. Pharr, "Real-Time Path Tracing and Beyond." HPG 2022 Keynote, July 2022.

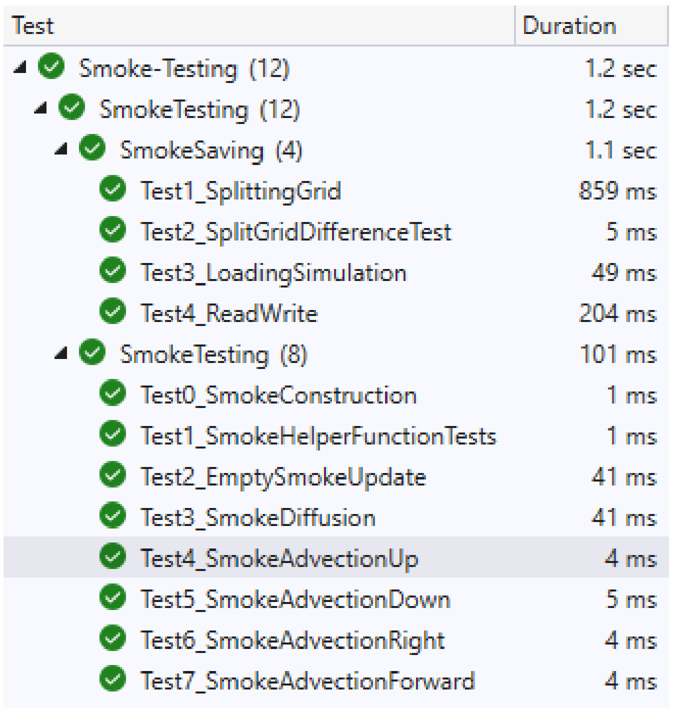
Appendix A
Data-Analysis using R



```
install.packages("tidyverse")
library(ggplot2)
library(readxl)

#---- Hypothesis 1 ----#

#import data from spreadsheet
SmokeRenderingData <- read_excel("Smoke_Data.xlsx")

#change renderer column type from char to factor
SmokeRenderingData$Renderer <- factor(SmokeRenderingData$Renderer)

#perform regression test to calculate the resolution and rendering methods effect on frame time
#regression formula: Frame-Time = Renderer + Resolution$^3$ + Renderer * Resolution$^3$
lmData = lm(formula = Frame_Time ~ I(Resolution^3) * Renderer , data = SmokeRenderingData )

#view regression results
print(summary(lmData))

#---- Hypothesis 2 - Scenery Test ----#

#rendering with scenery analysis
SmokeRenderingDataScenery <- read_excel("Smoke_Data_Scenery.xlsx")

#change renderer & scenery column type from char to factor
SmokeRenderingDataScenery$Renderer <- factor(SmokeRenderingDataScenery$Renderer)
SmokeRenderingDataScenery$Scenery <- factor(SmokeRenderingDataScenery$Scenery)

#perform regression test to calculate the resolution, rendering method, and scenery's effect on frame time
#regression formula: Frame-Time = Renderer + scnery + Resolution$^3$ + Renderer * Resolution$^3$ + Renderer * Scenery
lmData_Scenery = lm(formula = Frame_Time ~ I(Resolution^3) * Renderer  + Scenery * Renderer, data =
SmokeRenderingDataScenery )

#view regression results
print(summary(lmData_Scenery))

#---- Visualisation ----#

#separate the two rendering methods into different data frames
RaysCastingData<- SmokeRenderingData[ SmokeRenderingData$Renderer == "Ray-Casting", ]
VoxelsData <- SmokeRenderingData[SmokeRenderingData$Renderer == "Voxels",]

#assign a colour to both renderers
colors <- c("Voxels" = "tomato", "RayCasting" = "steelblue")

#plot the graph as smooth line graph, x-resolution, y-frametime, spit the renderers into different lines
ggplot() +
  geom_smooth(aes(x = RaysCastingData$Resolution, y = RaysCastingData$Frame_Time, color = "RayCasting"),se=F) +
  geom_smooth(aes(x = VoxelsData$Resolution, y = VoxelsData$Frame_Time,colour="Voxels"),se=F) +
  labs(x = "Smoke Simulation Resolution", y = "Frame Time (ms)", color = "Rendering Method") +
  scale_color_manual(values = colors)
```

Fig. 10: Code snippet of the R code used to conduct statistical analysis and visualisation

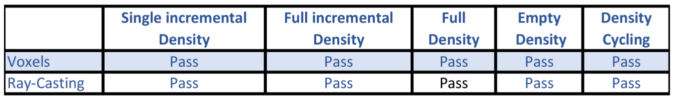
Appendix B
Artefact Unit Testing



```
Test                                        Duration
◢ ✓ Smoke-Testing (12)                      1.2 sec
  ◢ ✓ SmokeTesting (12)                     1.2 sec
    ◢ ✓ SmokeSaving (4)                     1.1 sec
        ✓ Test1_SplittingGrid               859 ms
        ✓ Test2_SplitGridDifferenceTest       5 ms
        ✓ Test3_LoadingSimulation            49 ms
        ✓ Test4_ReadWrite                   204 ms
    ◢ ✓ SmokeTesting (8)                    101 ms
        ✓ Test0_SmokeConstruction             1 ms
        ✓ Test1_SmokeHelperFunctionTests      1 ms
        ✓ Test2_EmptySmokeUpdate             41 ms
        ✓ Test3_SmokeDiffusion               41 ms
        ✓ Test4_SmokeAdvectionUp              4 ms
        ✓ Test5_SmokeAdvectionDown            5 ms
        ✓ Test6_SmokeAdvectionRight           4 ms
        ✓ Test7_SmokeAdvectionForward         4 ms
```

Fig. 11: Artefact unit tests within visual studio's testing framework

Appendix C
Artefact Integration Testing



	Single incremental Density	Full incremental Density	Full Density	Empty Density	Density Cycling
Voxels	Pass	Pass	Pass	Pass	Pass
Ray-Casting	Pass	Pass	Pass	Pass	Pass

Fig. 12: Manual Artefact integration test table

Figure 12 shows an integration test table. As specified earlier, following incremental integration testing means that each time a new feature is added, which could affect the renderers validity, an integration test is taken to verify they are still rendering correctly.

The test runs five different density grids through both renderers, which can be manually identified as correct or not, corresponding to a pass or fail. The first test, single incremental testing, checks the density grid by clearing the grid and adding density to a single cell, which moves to the next cell sequentially each frame. The second test is identical except it doesn't clear the grid each frame, leading to the density filling up row by row. These two test prove density is being rendered correctly at all points, in the grid.

The remaining tests are general checks for full or empty density grids, and also cycling between the two. These are used to identify that the renderer is being updated correctly, and that the bounding smoke box is accurate.

Appendix D
Acknowledgements

I would like to thank my supervisor Brian McDonald for his guidance and support. I would also like to thank the module leader Michael Scott for his dedication to the course, along with his help towards my research. Thanks also go out to my peers within BSc for their encouragement and support.

Appendix E
Artefact Git Repository

https://github.falmouth.ac.uk/DF245193/Comp320-Artefact