Advancing the Simulation and Rendering of the Aurora

Toru Yamaguchi

School of Computer Science

College of Engineering and Physical Sciences

University of Birmingham

2024-25

Abstract

As the understanding of complex plasma phenomena, such as the aurora, continues to advance, realistic simulations emerge as valuable tools. To be able to reproduce aspects of the aurora, it validates our models and enables us to drive further progress. In the field of computer graphics, physically-based fluid simulations have become the benchmark for realistic visualisation of natural phenomena; where these methods have enabled impressive levels of realism in both offline and real-time applications. This project integrates a GPU-accelerated, physically-based plasma simulation and volumetric rendering to visualise the aurora. Capturing the aurora's complex dynamics and producing visually faithful results is one of the primary aims of this work. We achieve this by producing a parallel Particle-in-Cell (PIC) solver and employing volumetric ray-marching to render auroral dynamics with high fidelity in real-time.

Keywords: Physics simulation, Plasma simulation, Ray marching, Real-time rendering

Contents

Abstract **ii**

1 Introduction **v**

2 Literature Review **vi**
2.1 Layout . . . vi
2.2 Auroral Phenomena . . . vi
2.3 Physics Simulation . . . vii
2.4 Aurora Simulation and Visual Techniques . . . viii
2.4.1 Visualisation . . . x

3 Design **xi**
3.1 Methodology . . . xi
3.1.1 Physics Simulation . . . xi
3.1.2 Visualisation . . . xiv
3.2 Software . . . xv

4 Implementation **xvi**
4.1 Introduction to Niagara . . . xvi
4.1.1 Emitters . . . xvii
4.1.2 Dealing with Grids . . . xviii
4.2 Niagara Data Interface . . . xviii
4.2.1 Background . . . xviii
4.2.2 Defining the Data . . . xix
4.3 The Simulation Pipeline . . . xxi
4.3.1 Working with Shaders . . . xxi
4.3.2 Initialising Textures . . . xxii
4.3.3 Shader Function and Shader Parameters Setup . . . xxii
4.3.4 Scatter To Grid . . . xxiii
4.3.5 Computing Charge Density . . . xxvi
4.3.6 Solve Plasma Potential . . . xxvii
4.3.7 Solve Electric Field . . . xxxii
4.3.8 Gather to Particle . . . xxxiii
4.4 Testing . . . xxxv
4.4.1 Initialisation . . . xxxv
4.4.2 Logging . . . xxxvii
4.4.3 Render Target . . . xxxviii
4.4.4 A Stable Solution . . . xxxix
4.4.5 Profiling . . . xlii
4.5 Visualisation . . . xliii
4.5.1 Creating a Particle Density Texture . . . xliii
4.5.2 Render Target . . . xliv
4.5.3 RayMarching . . . xlv

4.6 Project Management l

5 Evaluation **li**
5.1 Performance li
5.2 Realism lii
5.3 Suitability liii

6 Conclusion and Future work **liv**

CHAPTER 1

Introduction

Fluid simulations, or Computational Fluid Dynamics (CFD), is a field of physics simulations that has had immense growth in its realism, numerical accuracy, and computational performance to visualise realistic fluids over the past two decades. This level of progress has inevitably lead to its influence across many domains such as engineering[57] [50] [59], architecture [43] [45] [71] [47], and the entertainment industry [69] [69] [75] [11].

In contrast plasma simulations have lagged behind. Plasmas are governed by the coupled dynamics of charged particles and electromagnetic fields which present additional numeric and stability challenges. Accurate plasma models must resolve complex phenomena such as Debye shielding and wave-particle interactions [23]. This complexity has traditionally constrained high-fidelity plasma simulations to offline environments, meaning outputs to the simulation, such as visualisation, is de-coupled with its execution.

A well known, visually striking phenomenon caused by plasma dynamics is the aurora. This natural, local phenomenon serves as a visually verifiable testing ground for plasma-based simulations. Decades of continued research into the auroras background and physical causes such as with satellites [61] [19], ground-based imagers [26] [27], and in-situ probes [55] [13] has firmly established the integration of plasma based dynamics that produce the dynamism we can observe. A natural step to take when verifying theory beyond data collection, is to attempt to simulate the processes. Previous aurora-based simulations have attempted to incorporate some level of plasma physics [9], however, must rely on a range of simplifying assumptions whilst keeping the model offline due to computational constraints.

This project attempts to address the gap of physically-based, real-time auroral visualisation. The primary contributions are the following:

- A physically based plasma simulation using the Particle-in-Cell method, built to operate completely on the GPU. This involves simulating the electrostatic forces electrons produce which follow the dynamics seen with the aurora.
- A volumetric ray-marching renderer that uses the particle's density within the simulation to visualise the auroral structures.
- An implementation written using Unreal Engine, which allows GPU-based particle simulations built to operate in real-time applications.

Chapter 2 will provide an in-depth analysis of auroral makeup and morphology, physics simulations and in particular plasma simulations, a review of past implementations, and visual techniques; Chapter 3 will share the theory that was implemented, including a list of the specific methods and formulas. It will also cover the reasoning behind the software chosen; Chapter 4 covers the entire implementation, testing, and project management; Chapter 5 evaluates the project based on performance, realism, and its suitability; Chapter 6 provides a conclusion and a discussion on future work.

CHAPTER 2

Literature Review

2.1 Layout

- Section 2.2: Auroral Phenomena
  - To understand the fundamental physical processes behind the aurora and where our current understanding is.
  - To understand their visual makeup and morphology.
- Section 2.3: Physics Simulations
  - To understand the current state of physics simulations.
  - To understand the key philosophies behind simulations.
  - To review key literature for plasma simulation.
- Section 2.4: Aurora Simulations and Visual Techniques
  - To review past implementations of aurora simulations and discuss what can be improved upon

2.2 Auroral Phenomena

The aurora is a visually striking phenomenon that arises when solar-wind driven charged particles are channelled along earth's magnetic field lines and enter into the upper atmosphere, known as the ionosphere. The ionosphere is not a fixed region in space, as it represents the dynamic region in which the density of charged particles rises that spans from 85-1000km [24]. The increase in charged particles is caused by intense solar radiation striking the gases in the upper atmosphere, stripping the electrons away from the neutral particles (such as oxygen, nitrogen) causing them to transform into positively charged ions. This gas-like state of ions and electrons is known as plasma. Charged particles experience electromagnetic forces such as electrons that experience the Lorentz force making them follow magnetic field lines in a helical, gryo motion [56]. As the charged particles start to reach the polar regions, where the magnetic fields lines start to direct down towards earth, they start to experience collisions with neutral particles. This collision rate only increases the lower the altitude as the density of neutral particles increases. A collision with a neutral particle leads to a transfer of energy, where the collided neutral particle becomes 'excited' and upon returning to its ground state, it emits a photon. This photon is the visible aurora that we can observe.

To focus this review, I will consider the two overreaching types of aurora, mainly diffuse and discrete aurora. Diffuse aurora implies there is no specific structure, where the aurora appears as broad, static glows spanning large regions of the nightside polar cap. Paschmann [24] summarises their cause to be from electron pitch angle diffusion by wave-particle interactions or whistler waves. In contrast, discrete aurora appear as narrow arcs or curtains with fine structure. There are two key mechanisms that drive their formation:

1. Inverted-V events: quasi-static parallel electric fields aligned with magnetic field lines. The fields accelerate the electrons into narrow altitude bands, producing sharp arcs [24].

2. Alfvén waves are also known to cause dynamic, small-scale aurora [53] [30] [13].

These phenomena cause distortions within the plasma, giving the aurora their characteristic appearance and morphology. A broad taxonomy for the type of discrete structures are spirals, folds, curls, rays, and corona [40]. Whilst this taxonomy isn't an exact objective classification, with continuous investigations on the different types of aurora [16], it provides a clear framework for verifying that our plasma simulations naturally reproduce the morphologies observed.

2.3 Physics Simulation

Before reviewing the past implementations, I will introduce the field of physics simulations, their underlying complexities, and the current solutions with specific relevance to plasma simulations.

Physics simulations use computational methods to solve mathematical models that describe physical phenomena. These simulations predict the behaviour of natural processes across a wide range of domains, from fundamental scientific research and online educational resources to the realistic water, smoke, and fire effects in films, TV, and video games. Models are derived from fundamental equations of physics, like Newton's laws of motion, the Navier–Stokes equations for fluid flow, and Maxwell's equations for electromagnetism. In order to transform these laws into equations that the computer can solve, you must apply discretisation schemes such as finite-difference, finite-volume, and finite-element methods. To provide some context of the capabilities of physics simulations I will briefly highlight the advancements fluid simulations have made before reviewing the field of plasma simulations.

Fluid simulations were not always designed with visualisation in mind; historically they were designed mostly for analytical use (Analytical Fluid Dynamics) and Experimental Fluid Dynamics (EFD). One of the challenges with reproducing physical phenomena, was finding stable solutions to the underlying mathematical processes. Stam [70] in 1999, introduced an unconditionally stable solution to simulating fluids that allowed for real-time visualisation. Although this was not the first to produce a stable solution for fluid simulation, it managed to show that it was possible to produce real-time visuals of fluids that used physically based methods. Preceding this paper, much of the computation required for solving fluids was thought to be limited to offline methods, which de-couples the simulation to the outputted results. Instead, this solver was used specifically for real-time video games, as it was efficient and unconditionally stable (it would never diverge). This prompted significant research in the combined field of physically based simulations and computer graphics. Fluid simulations for real-time visualisation has improved significantly, with some game engines [35] pre-built with modern fluid simulation capability. This impressive leap in performance and visual fidelity shows the capability of physical simulations and their use for real-time applications.

Plasma simulations have yet to see the level of capability computational fluid dynamics have achieved with regards to visualisation. This lag in progress is not that surprising as plasma phenomena can be far more complex to simulate. The added complexity comes from additional fundamental factors associated with plasma such as incorporating electromagnetic forces, modelling the effects of particles to electric fields, and obeying the many added constraints that govern plasma physics. Space and Astrophysical Plasma simulation [20] provides an excellent overview of the main techniques for simulating plasma. Firstly, I'll touch on the magnetohydrodynamic (MHD) method which avoids the small-scale complexities of plasma by representing the entire plasma as a fluid, that interacts with magnetic and electric fields. It models the plasma using a fluid-like momentum equation combined with the Maxwell's equations to evolve the electromagnetic fields. This enables you to simulate a large-scale collective behaviour of plasmas without individual particle tracking. There have been some MHD-based simulations adjacent to the topic of the aurora: Chaston [22] produced a 3D simulation of the physical mechanisms that underpin the formation of discrete auroral arcs, such as Alfvén waves produced by the ionospheric feedback instability. Although it only indirectly produces how the aurora is formed, it shows the potential effectiveness of MHD as a method to simulate the formation of discrete auroral features. Interestingly, this solution is also time-dependent which further supports this evaluation. This follows a trend of various MHD simulations that are focused primarily on the causes of the aurora as opposed to the aurora itself. Ebihara [31] [32] [33] produced various MHD simulations for modelling auroral substorms, which is a short-lived disturbance in the magnetosphere that can cause a large release of energy, causing auroras to intensify. Other MHD simulations include simulations of the solar wind [67] [39] and instabilities for the formation of the aurora [64]. These simulations are clearly focused on the formation

of the aurora by simulating the external factors. Another observation is that these MHD simulations seem to be only be simulating large-scale systems. However, this raises the concern with the suitability of the MHD method for capturing 'small-scale' motions and morphology of the aurora.

This project will primarily concern itself on being able to model the small-scale, dynamic morphology of the aurora. This leads onto the method, Particle-in-Cell (PiC), which can be thought of as a hybrid Eulerian, Lagrangian approach for simulating collisionless plasmas. It is a kinetic method, where individual particles interact with the electromagnetic field that the particles themselves produce. Each particle interacts with the underlying electromagnetic field represented by a grid, after which the field is re-computed to calculate the new electromagnetic forces at each grid point. The Eulerian side to this method regards simulating the changes with the grid data, whilst the Lagrangian describes the representation of the plasma as individual particles. Although, the plasma is represented with these particles, the particles do not correlate as a 1 to 1 ratio with actual particles, instead we combine many actual particles into a 'superparticle'. There have been many instances of literature regarding PIC simulations for astrophysical phenomena, such as Pohl [62] and Nishikawa [58]. Both provide comprehensive reviews of PiC as a simulation technique for plasmas of varying applications within the astrophysical field. Although auroras are not mentioned, the viability of this method for capturing the small-scale perturbations and dynamic movement has been indirectly shown with examples of successfully simulating other astrophysical phenomena. Space and Astrophysical Plasma simulation [20] contains a great overview of PIC simulations. It helpfully provides insight into important factors that need to be addressed for these simulations: such as the field solver, time steps size (Courant, Friedrichs and Levy (CFL)), interpolation method, particle motion integration, particle initialisation, boundary conditions, and parallelisation. The exact details of how this project deals with these factors will be addressed in the next chapter. In terms of translating this method for actual computational use, I found Plasma Simulations by Example [15] to be a great source to describe the many methods to implement a number of plasma simulations, including MHD and PIC.

2.4 Aurora Simulation and Visual Techniques

There are a few notable past implementations that have attempted to produce simulations of the aurora. These range from physically based methods to purely visual representations. Baranoski produced a number of implementations [10] [8] [9], with subsequent iterations building on from the previous. I will specifically review the most recent iteration, however, to start off I will give a short summary of the progress between the 3 iterations. The first and second implementations [10] [8] are largely the same, with the second refining and detailing the methods used in more detail. It improved the visual and mathematical background and offered insight into how it could be used for educational and scientific applications. Although the first two implementations used techniques that followed the physical nature of the aurora, it largely relied on stochastic and artificial disturbances to produce its final result. With the third paper [9], Baranoski introduced physically-based motion to more realistically simulate the movements of the aurora. A hybrid Eulerian-Lagrangian approach was used, where electron 'beam's interact with a background electric field in order to update its movements. A method used throughout the papers was this representation of the aurora as electron 'beams', which if placed along a narrow path along the xy axis, can simulate the visually recognisable auroral 'curtain' shape. This method of representing electrons as 'beams' is similar to the use of 'superparticles' with the aforementioned PIC approach. With regards to the hybrid Eulerian-Lagrangian method for particle-field interaction, this also has a similar workflow to that of the PIC approach for simulating astrophysical plasmas [42]. Although many instances of the implementation are similar, there are key aspects of the solution that don't align with that of the PIC approach. Examples of missing factors are the boundary conditions and a plasma specific force integration. With a proper force integrator, as seen with previous simulations [15] [23], an electromagnetic energy conserving solver increases the accuracy of simulated particles, which would further improve this simulation. By fully incorporating these factors that PIC simulations require, Baranoski's method for simulating plasma phenomenon would be capable of more physically accurate results.

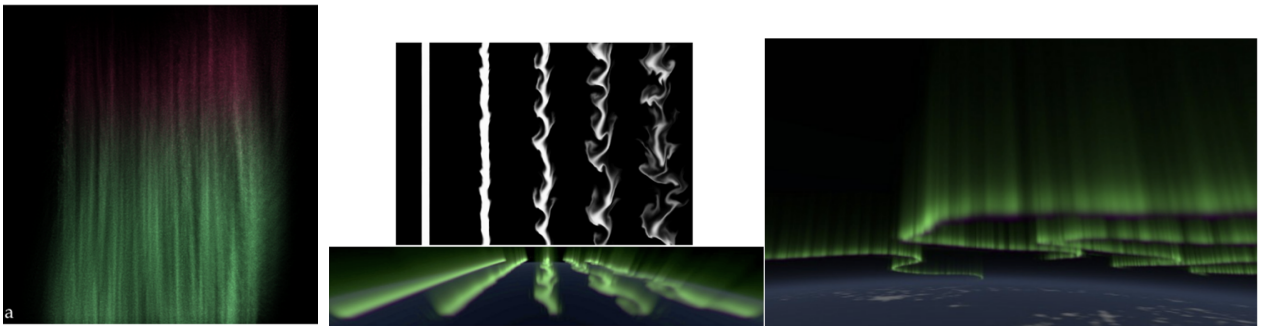
Baranoski's implementations also covered the visualisation of the electron beams to reproduce the aurora. This involved a forward mapping approach of the emission points, which are mapped to the image plane with small perturbations. The pixel is coloured via a spectral lookup table to reproduce the emissions of the aurora. The method is finalised with a Guassian blur applied to the image to mimic the effects of long exposure photography on visualising the aurora. These emission points are computed as the points at which the beam experiences a random disturbance to alter its trajectory. The use of these deflection points as the points for emission, attempts to reproduce the physical process of collisions in the ionosphere. Whilst this method has a physical basis, it is a simplified view of the underlying processes. The method limits the emissions to the positions of the electron beams, which can cause the visuals to overly emphasise the trajectory

Figure 2.1: Baranoski [9], simulation sequence of an auroral rayed band

of the particles. Therefore, I believe the Guassian blur helped to reduce the distinct trajectories and produce a more diffuse look. Figure 2.1 shows an example of the resulting implementation. Baranoski also mentioned the suitability for a parallel-based approach on the GPU, which is something that I will be looking at implementing for this project. Overall, Baranoski provided a foundational technique for simulating the aurora, with an emphasis on the physical processes that mimic its form, morphology, and appearance. Baranoski also provided an additional paper on plasma simulations for general astrophysical phenomena and catered it towards the rendering processes [7]. It emphasised the efforts made with rendering plasma phenomena and the need for collaboration between physical scientists and the computer graphics community. I quite agree with Baranoski's final message here, as seen with the immense growth of computational fluid dynamics, there is certainly huge potential when it comes towards this intersection of fields.

I consider Baranoski's implementation to be the most rigorous, however, there have been several other implementations that have their specific use cases, most of which have been directly influenced by Baranoski. Mills [54] also devised a similar agent-based simulation but differentiated itself by using a different solver and a Monte Carlo based simulation of collisions. Mills solved the Poisson equation using a Fast Fourier Transform (FFT) method, which has been used to solve partial differential equations for fluids [3] before. This differs to the multi-grid method used by Baranoski, but it works well in this context where the simulation is CPU-based and uses periodic boundary conditions. The rendering technique used a very simple mapping of each particle to the screen, avoiding any complex operations, with the result shown with figure 2.22.2.a. Although the visualisation technique could be improved upon, I believe the simulation setup Mills used could be a good starting point for my implementation.

Although I will be focusing on physically based simulations, there have been many implementations that produce purely visual renders of the aurora and rely techniques that mimic the aurora motions and morphology. There are still elements of these implementations that can be taken into account especially when considering the rendering process. Lawlor [51] produced a real-time render of the aurora on the GPU, of which many elements are desirable for this project. The real-time aspect hasn't been applicable to any of the previous implementations, due to the computationally intensive calculations required for solving the physical equations. Interestingly, the method for producing the shape of the Aurora was inspired by Baranoski's first two implementations [10] [8], which Baranoski later replaced a more physically accurate method. Lawlor's method involves firstly defining a 2D spline with a fluid Stam-type simulation along it to produce the fluid-like motion that auroras can exhibit. The 2D spline is extruded into 3D space by simply expanding its vertical trajectory. This visually follows a similar result to Baranoski, however this implementation avoids the complex plasma physics by employing this 2D simple fluid solver. We know, from the brief section on fluid simulations, that Stam's fluid solver is efficient enough for real-time applications, which allows for Lawlor's method to follow suit. The most interesting aspect to this implementation is the visualisation technique. Lawlor uses a ray-tracing technique accelerated with distance fields. This was a novel technique for visualising the auroras, although the colouring format followed the same method as Baranoski. Ray-tracing can be described as the opposite to how Baranoski visualised his aurora - forward mapping. Instead of drawing rays from the emission points to the camera, ray-tracing shoots rays from each pixel into the scene and accumulates the light and opacity for that pixel. This method accurately captures the lighting and visual makeup of a scene, especially when you want to capture a volumetric phenomenon. This technique was further optimised using distance fields as shooting rays into areas that don't contain any aurora can be computationally expensive. Figure 2.2.b shows the evolution of the 2D splines being extruded into 3D space to produce the final result. The visualisation technique is something that can be implemented with this project, making sure to adjust



(2.2.a) Mills [54]

(2.2.b) Lawlor [51]

it accumulate through a particle-based simulation rather than the extruded spline. Lawlor taking advantage of the GPU is also something that I will incorporate into the project. Lawlor managed to keep the final result in real-time, by using efficient methods such as the Stam-type fluid solver instead of a complex plasma simulation and accelerating the visualisation process using distance fields. In order for my implementation to stay in real-time, I will also have to find methods for efficiency gains.

2.4.1 Visualisation

The raytracing method used by Lawlor [51] shows the potential with using a GPU-based rendering approach for visualising the aurora. Impressive volumetric rendering examples have been released in the recent years, such as Schneider with Nubis[3] [66] which achieved highly detailed, immersive cloud rendering using an accelerated ray-marching approach with fluid-simulation models. Schneider himself has been producing volumetric based rendering of natural phenomena for over a decade [65]. The ray-marching approach has been used throughout his publications, which involves shooting rays into a scene, stepping through a volume, accumulating data at each step, integrating the data into a resulting pixel colour. Using this approach to visualise the effects of plasma is something that will be explored with this project.

This concludes the literature review. Here, I covered the main topics that helped form the requirements and desires for this project. Some of the fundamental features relating to the implementation have only been covered briefly, as a more in depth discussion of the projects design is left for Chapter 3. As a final discussion, I will mention the possible use cases for this project. Physics simulations have successfully found themselves to be prevalent across many industries and domains; their integration with the graphics community allows an accessible insight into physics processes that underpin natural phenomena. Arguably the biggest way physically based simulations have been used, is with the entertainment industry. Countless examples of computer generated natural phenomena have been used across media, from video games [66] [35] to films and tv [14] [17]. However, a fairly unnoticed application would be for the educational sector. Physics Education Technology (Phet) simulations [2] is an open-source, free, set of simulations that covers a broad range of domains. They cover fundamental physically based phenomena, with some having relevance to this project such as electromagnetism and Maxwells equations. A number of studies show the benefits of bringing physics simulations into the hands of students [5] [6] [76], especially to those that wouldn't usually have access to such resources. Educational resources must also consider the visual and interactive side to learning, as it can greatly enhance the experience. This could be another benefit to choosing a PiC simulation over the MHD, as representing plasmas with kinetic particles is far more visually intuitive to describe the underlying processes behind the aurora as opposed to the fluid based method. Therefore, the incorporation of more advanced physical phenomena such as space weather (auroras, pulsar winds, solar flares) with accompanying visuals to the educational sector would be greatly beneficial.

CHAPTER 3

Design

3.1 Methodology

The methodology will be an implementation independent analysis of the techniques that will be implemented. It includes detailed theory and analysis of the main methods with clear reasoning on their relevance to the simulation. It will also mention any simplifying assumptions that have been made. The section will be split into the physics simulation and visualisation.

3.1.1 Physics Simulation

As an overview, I will implement a Particle-in-Cell (PIC) based plasma simulation on the GPU. PiC requires the interaction of kinetic particles with an underlying grid. Following Mill's [54] implementation, I will be defining a 3D grid as a cube, where each grid cell contains attributes related to computing the electric field. The simulation is broken into many steps, each attempting to solve an equation in order to produce a resulting electric field. One assumption that will be made is modelling the magnetic field as static, this vastly simplifies Maxwells equations as it allows us to model the magnetic field as time invariant. Although the magnetic field is still considered with the Lorentz force, the main forces for capturing the auroras motion will be electrostatic such as natural charge separation. With this assumption, we can derive the electrostatic potential equation:

$$\overrightarrow{E} = -\nabla\phi \tag{3.1}$$

This describes the electric field as the negative gradient of the potential. We can substitute this equation into Guass' law to obtain the Poisson equation:

$$\nabla \cdot (-\nabla\phi) = \frac{\rho}{\epsilon_0} \implies \nabla^2\phi = -\frac{\rho}{\epsilon_0} \tag{3.2}$$

The resulting equation provides a method of computing the potential (ϕ) from the charge density (ρ). Therefore, by computing the charge distribution in the grid, we can derive the potential and then the electric field. I will now describe how this is computed with relevance to GPU execution.

1. To compute the charge distribution for the grid, we simply interpolate each particle's charge to the grid. To do this, a common method is to use linear interpolation or cloud-in-cell [12] (CIC) to 'scatter' the charge to the grid nodes that surround a given particle.

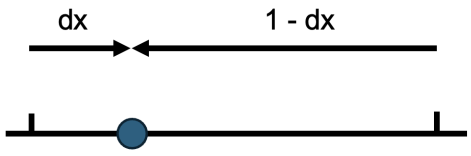


Figure 3.1: A 1D example of the 'scatter' process

Figure 4.13 shows a 1D example of how to compute the linear interpolation of a particle's charge to the surrounding grid nodes. This is easily extendable to three dimensions, which will be done with the implementation. This method more evenly distributes the charge throughout the grid, instead of assigning the entire charge of a particle to the closest grid node. Although it is computationally more expensive it produces much more accurate results than a straight up increment for the closest node, which was implemented by Mills [54].

2. With the resulting charge at the grid nodes, it allows us to solve the Poisson equation eq3.2 to obtain the plasma potential. In order to solve partial differential equations, we use approximation methods, such as the finite difference method. This involves finding the difference between neighbouring potential values and dividing by the spacing between the two.

$$\frac{\partial^2 \phi}{\partial x^2} \approx \frac{\phi_{i+1} - 2\phi_i + \phi_{i-1}}{\Delta^2 x} \tag{3.3}$$

If we substitute this into the Poisson equation and rearrange to solve for the potential, the resulting equation forms the following:

$$\phi_{i+1} - 2\phi_i + \phi_{i-1} = -\frac{\rho_i \cdot \Delta^2 x}{\epsilon_0} \tag{3.4}$$

To solve this, there are a number of different methods to choose from. The literature review found examples of multi-grid [9] and fast-fourier [54] solvers; however, given that I will operate in a GPU context, this complicates the integration of these methods. There exists a class of methods, iterative or convergence-based solvers that work by starting with an initial guess, and improving it over many subsequent runs. They naturally suits a GPU context because they use simple, localised memory access operations that can be efficiently parallelised.

GPUs execution differs greatly to CPU processing, as they contain thousands of cores that are designed for lightweight computations to be executed in parallel. Parallel solutions for the CPU do exist and they are commonly used in scientific computing and High Performance Computing (HPC). However, as part of the visualisation of this project, I will utilise GPU-based real-time rendering which will directly sample data from the simulation; in order to closely integrate the simulation results and the rendering process, I chose to use a GPU-based solver in order to avoid expensive CPU-GPU transfers and to exploit massive parallelism using compute shaders.

Therefore, I will run through how iterative solvers are implemented for GPU-based systems. The most basic method would be to use the Jacobi iteration. Given a grid with the current potential at each node i as ϕ_i and current charge density ρ_i, you re-arrange the discretised Poisson equation 3.4 to find the new potential value for a given grid node. Therefore, the equation used for the compute shader will take the form:

$$\phi_i = \frac{1}{2}(\phi_{i-1} + \phi i + 1 + \frac{\rho_i \cdot \Delta^2 x}{\epsilon_9}) \tag{3.5}$$

Although this is for the 1D example, it maps very easily to higher dimensions, by just accumulating the differences between the neighbours of each axis and dividing by twice the number of dimensions, so dividing by 6 if we operate in 3 dimensions. This formula is ran for multiple iterations so that we can attain a good approximation for the potential value at each grid node.

3. This puts us at the final step, which is to obtain the resulting electric field by solving Guass' law. The equation 3.1 computes the electric field by obtaining the gradient of the potential field. I will use the same finite difference method used before to solve this equation, only this requires a first-order derivation as opposed to the second-order with the Poisson equation. This is very easily parallelisable as I simply run the same equation for every grid node. The equation used will also use the central difference variation of the finite difference method:

$$E_i = -\frac{\phi_{i+1} - \phi_{i-1}}{2\Delta x} \tag{3.6}$$

4. With the electric field calculated for each grid node, the last step is to interpolate the electric field vectors for each particle and integrate it into the particle's motion. Particles under electromagnetic forces experience the Lorentz force:

$$\overrightarrow{F} = q(\overrightarrow{E} + \overrightarrow{v} \times \overrightarrow{B}) \tag{3.7}$$

This is used to calculate particle acceleration by applying Newton's second law: $F = ma$ to form the first-order differential equation for the acceleration:

$$\frac{d\overrightarrow{v}}{dt} = \frac{q}{m}(\overrightarrow{E} + \overrightarrow{v} \times \overrightarrow{B}) \tag{3.8}$$

However, this equation is typically not implemented in a direct way; instead, in order to integrate a method that conserves energy and keeps the motion of particles stable, it is standard to use the Boris pusher. This method splits the integration step into 3 sections. First apply half the electric field force:

$$v^{-} = v + \frac{1}{2} \cdot \frac{q\Delta t}{m}\overrightarrow{E} \tag{3.9}$$

Where q is the charge of the particle and m is the mass. The next step is to calculate the full magnetic field rotation. Instead of applying rotation using sine/cos, which can be expensive, we use Rodrigues' formula to apply the magnetic field rotation. This requires you to compute the rotation vector and a helper function. These are then used to compute two cross product computations.

$$\overrightarrow{t} = \frac{1}{2} \cdot \frac{q\Delta t}{m}\overrightarrow{B} \tag{3.10}$$

$$\overrightarrow{s} = \frac{2\overrightarrow{t}}{1 + |t|^2} \tag{3.11}$$

$$\overrightarrow{v}^{"} = \overrightarrow{v}^{-} + \overrightarrow{v}^{-} \times \overrightarrow{t} \tag{3.12}$$

$$\overrightarrow{v}^{+} = \overrightarrow{v}^{-} + \overrightarrow{v}^{"} \times \overrightarrow{s} \tag{3.13}$$

The final step is to compute the other half of the electric field force and integrate the resulting velocity to the particles position with the simple forward Euler update.

$$\overrightarrow{v} = \overrightarrow{v}^{+} + \frac{1}{2} \cdot \frac{q\Delta t}{m}\overrightarrow{E} \tag{3.14}$$

$$x^{n+1} = x^n + v^n \Delta t \tag{3.15}$$

5. This concludes the necessary theory behind obtaining and integrating the dynamic electric field for particles in the simulation.

There are some additional aspects that must be discussed with the PiC implementation. Firstly, I will discuss boundary conditions. With any PiC simulation, the boundary conditions define how the plasma enters and leaves the domain. The most common are the following: Periodic, where particles that leave one side are re-injected in the opposite end; Dirichlet, sets the potential at the boundaries as a fixed value; Absorbing, absorbs outgoing waves to prevent reflections. To maintain computational efficiency, the periodic boundaries will be chosen for this implementation. As I will have to maintain boundary conditions when solving the field equations, picking ones that align with GPU-based computations should be a priority. If you refer back to the equations that implement the finite difference method, you can see that there is a lot of neighbour indexing to read their values. This would require branching to check that the current node is at the boundary or not. Something that goes along the lines of`if (idx == 0 || idx == N-1`, where N is the number of cells. It is well known that branching should be minimised when working with GPU-based computations [41], and even with thread divergence being handled better with modern GPUs, it still remains a bottleneck to GPU execution. Periodic boundary conditions allow for non-branching execution which presents an efficient solution to computing boundary conditions which is done extensively within the different stages of the solver. Therefore, to calculate boundary conditions without potential for branch divergence, I will simply exploit the modulo operator to wrap any indices that extend further than the number of grid cells. The following equation captures the equation used, it makes sure that any negative indices properly wrap around as well.

$$idx = (x + N) \mod N \tag{3.16}$$

The last aspect to the PiC simulation that I will discuss are the initialisation conditions. For the field attributes, such as charge density, potential, electric field, we assume for them to start at 0. A challenging aspect with PiC simulations is to reduce the noise associated with the particles. Since the number of

'superparticles' is significantly smaller than actual number of particles, it can cause unphysical fluctuations. The initial placement of particles can greatly affect the initial noise of the simulation, therefore a common method to initialise particle positions place particles in an evenly spaced grid, called the 'quiet start'. Since I will be using periodic boundaries, I don't require the injection of new particles into the simulation, which further reduces the complexity with enforcing quasi-neutrality - keeping the charge density neutral. Another assumption I will make with this simulation is that ions form a background charge that serves to neutralise the overall charge; this is because I won't be directly simulating ions. In order to maintain that the overall charge is neutral, after computing the electron's charge density, I subtract the ion charge.

3.1.2 Visualisation

The second part of the project will consist of devising a visualisation algorithm based on the particle simulation. This involves the use of a ray-marching approach, a common technique for rendering volumes for real-time applications. I will describe how this technique can be incorporated into a plasma simulation to visualise the aurora.

The ray-marching technique can be described by shooting rays through each pixel of the screen and evaluating the resulting colour and opacity by checking for intersections through small steps. This technique has been adapted for volumetric rendering by incorporating physically-based equations to imitate how emissive particles interact with light.

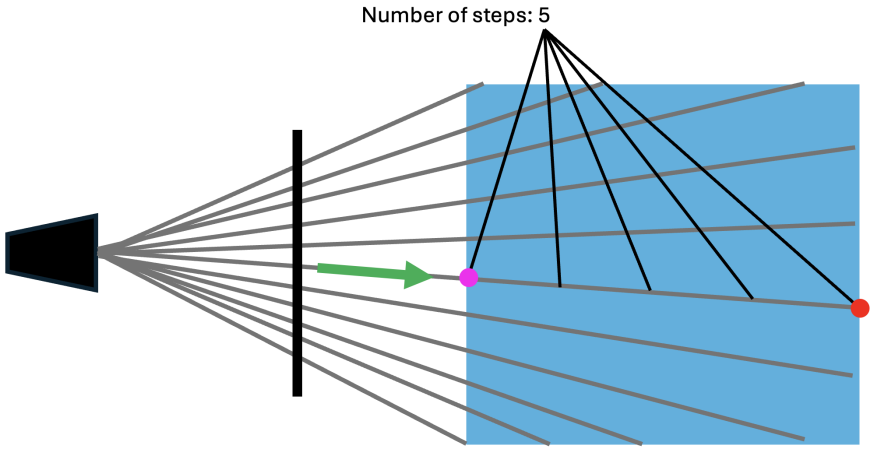


Figure 3.2: Visual for how the ray-march process will work

Figure 3.2 shows a high-level example of how the ray-march algorithm will work with the project from a birds-eye view. For standard ray-marching, you shoot rays through every pixel in the screen, however, with this implementation, I will limit the pixels to those with rays that enter the grid. In order to have a working ray-marching algorithm I will need to define some variables such as the direction of the ray, the entry and exit position of the ray with the grid, and the number of steps. In order to compute the pixel's resulting opacity, we use a simplified version of Beer-Lambert's Law. This law describes the change in 'radiance' as light travels through a medium.

$$T = e^{-distance \times \sigma_a} \tag{3.17}$$

It states that there is an exponential relationship between the transmittance (T) with the distance of the ray as well as the density of the medium (σ_a). For this simulation, I will define a texture that stores the particle's density in 3D space. By doing so, I can sample that texture as I march along the ray to continually update the transmittance value. This will evaluate to the resulting opacity at the given pixel. The colour will also follow a similar process.

3.2 Software

A very important aspect for the project will be to choose the appropriate software. There are many different variables that will affect the implementation, therefore I must be mindful to pick the technology that brings the most benefit to this specific application. I considered two main types of software, mainly low-level graphics APIs and high-level engines. I will list out a few examples from each and their suitability for this project with table 3.1.

Name	**Description and Suitability**
OpenGL	Well known graphics API developed in the early 1990s. It is considered the most friendly graphics API with good documentation. Considered legacy for the current landscape. Lacks modern optimisations and robust multi-threading support. Capable of particle simulations and raymarching via custom GLSL shaders.
Vulkan	Modern low-level graphics API released in 2016. Offers complete and explicit control over the GPU. Built for performance and efficiency; however, it comes with a very steep learning curve and forces the developer to maintain resources and synchronisation manually. Very suitable for advanced visualisation of particle-based simulations.
Unreal Engine	Leading game engine with cutting-edge graphics support and expansive set of development tools. Comes with an advanced particle system with built-in examples of fluid simulations. Resource-intensive, engine constraints can limit complete control over pipeline. Capable of visualisation techniques.
Unity	Popular game engine with user-friendly interface, large community and support. Provides particle system albeit with less flexibility and less low-level control. Offers custom compute shader capability.

Table 3.1: Comparing technology

The main takeaway from this was to find the technology that had both access to modern visualisation features and GPU programming control with a relatively decent learning curve. A worry with choosing a low-level API was that the 'overhead' with learning them would potentially outweigh the time spent actually coding the simulation and visualisation. Graphics APIs are notoriously difficult and time-consuming to learn, they give you access to all the materials but will not provide you with a lot of the basic tools as it is assumed for you to develop them yourself. An example would be the "Hello World" of graphics programming, which is to render a triangle on the screen [29] [60]; it gives you a sense of how low-level we would be working with. This leads onto evaluating the viability of implementing a full physics simulation with visualisation using graphics APIs. Valuable time spent on aspects like rendering a 3D scene or adding a controllable camera would encroach on time working with the simulation, testing parameters, and adjusting the visualisation. My limited experience with OpenGL ended up influencing my decision to implement the project in a more robust engine instead, as the graphics API route would be infeasible without more time and experience.

Therefore, I decided to go with **Unreal Engine** (UE) for the implementation. Unity, whilst being feature rich itself with good compute shader compatibility, lacks the general purpose development with UE. Whilst both engines aren't fully open source with certain aspects like the rendering pipeline hidden, UE is source-available which allows you to access and make changes to the source code. Having access to the source code allows you to further analyse the underlying architecture and to fully grasp what will be required for the implementation. The main concern with using UE would be the fact that it is still considered a 'game engine', meaning a physically based simulation wouldn't exactly fit that description. However, UE has managed to expand and diversify over the years, such as introducing modern fluid simulation techniques [36] which have enabled impressive fluid and gas visual effects. Therefore, it's clear that this engine is capable of handling high performance particle physics on the GPU.

CHAPTER 4

Implementation

- Section 4.1: Introduction to Niagara
- Section 4.2: Niagara Data Interface
- Section 4.3: The Simulation Pipeline
- Section 4.4: Testing
- Section 4.5: Visualisation
- Section 4.6: Project Management

4.1 Introduction to Niagara

Niagara is Unreal Engine's built-in VFX system, providing a rich set of features that allow for real-time simulation and rendering of particle based effects. I will define a Niagara System which will be the 'blueprint' for the simulation. This will encapsulate everything including the simulation, visualisation, and debugging. In Figure 4.1, I have opened the Niagara System to show the Niagara Editor. The following list will cover four important features the editor provides and how they relate to the project. The editor provides many more features that will be detailed in the following sections. However, to see an exhaustive list of features, see the official documentation [37].

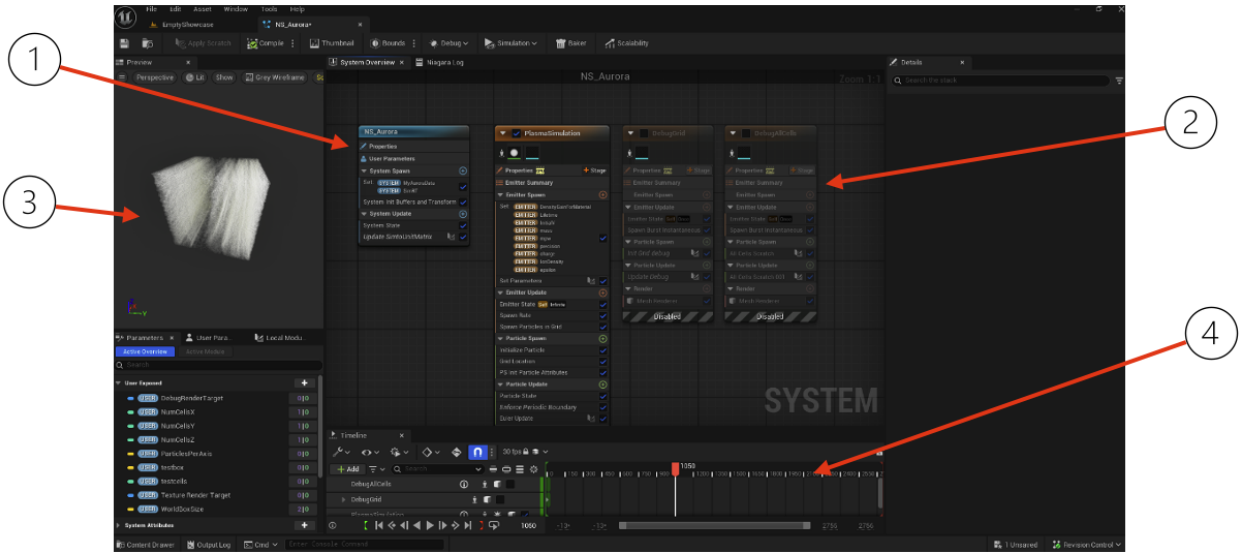


Figure 4.1: Overview of the Niagara Editor

1. Niagara System: The blue node is where the system properties and system-level parameters are defined. I used this to define properties that will be used across emitters.

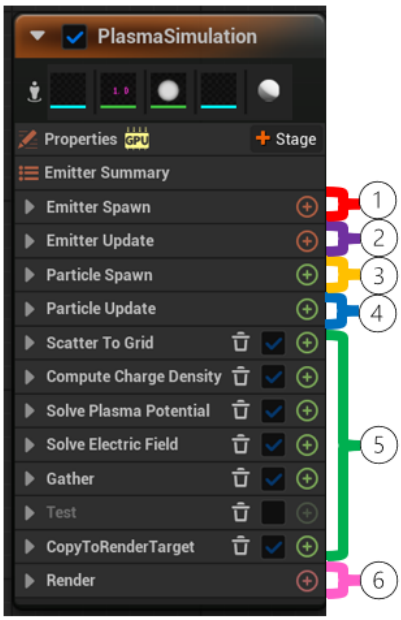
2. Emitters: Emitters are self-contained nodes for particle effects. I will implement separate emitters for different sections of the overall system. Emitters will be covered in more detail in the next section.

3. Viewport: This provides real-time rendering of the simulation. This was used extensively in testing to be able to visually see changes applied to the simulation.

4. Timeline Panel: Managing active emitters and controlling the current time in a run. Used in testing.

4.1.1 Emitters

Niagara offers the Emitter as a modular and highly customisable node in the Niagara system, where I can define what happens to particles at each stage of the simulation. I have created an emitter ("PlasmaSimulation") for my particles, where I define all their properties and where I also define all the custom modules which involve scripts that execute shader code. Figure 4.2.a shows an example of an emitter with its various 'groups'. With figure 4.2.b, I created a simplified visual of a single lifecycle of the emitter, referred to as a 'Tick'. As a quick overview, the groups in the emitter are executed sequentially from top to bottom, with each group containing 'modules' or shader scripts. The main groups will be covered with the following list.



(4.2.a) Compressed view of our emitter

(4.2.b) Lifecycle of the simulation, a "Tick"

1. Emitter Spawn Group: This group defines functions executed when the emitter is first spawned, similar to a constructor. I used it to initialise parameters for the simulation.

2. Emitter Update Group: Here I specify the lifecycle settings for the simulation, known as the 'Emitter State'. This is where I also defined the number of particles to spawn. The particles will be spawned at the start of the simulation upon definition, therefore this group isn't used proceeding that.

3. Particle Spawn Group: This group is called per spawned particle, where we define some important properties for each particle such as their lifetime, spawn location, and initial velocity. As with Emitter Update, this will only be required for the first tick.

4. Particle Update Group: This group is called per particle for every tick in the simulation. Therefore, I use it to enforce the `boundary conditions`, more on this later.

5. Simulation Stages: On top of the default groups, we can define extra custom groups called Simulation Stages. These modules can be understood as separate compute shader calls, where you specify whether you want to iterate over particles or over a data interface. An important aspect is that these groups can be specified to execute multiple times per tick. We will take an in-depth look at how I used simulation stages later with subsection 4.3.

4.1.2 Dealing with Grids

In the Design section, we looked in-depth with Particle-In-Cell (PIC) simulations, where particles interact with grid data in order to advance their positions in the simulation. Consequently, an important aspect of this Niagara System will be to define the properties of the 3D grid, which in turn guides the implementation strategy. The following design decisions were made for the grid:

- It will be defined as a 3D cube with periodic boundary conditions, meaning particles that exit on one side re-enter from the opposite side.
- At each grid node, we store the following attributes: charge density (float), electric potential (float), and electric field (float3). Therefore, this grid must be capable of storing multiple data types and their corresponding values at each grid node.
- The grid data will be stored in GPU memory to enable access through shaders.

The next step was to investigate how to define and interact with grid data within a Niagara System. I knew this was possible, since a similar method was used for Unreal Engine's built-in fluid simulation system. I found that Niagara provides built-in 'data interfaces' that allow data to be stored persistently on the GPU. Initially, I believed I could utilise these interfaces, as options such as *NiagaraDataInterfaceGrid3DCollection* appeared to be suitable. However, after some testing, I found that these interfaces lacked the necessary customisation and abstracted away control over the data. I required control over how and when the grid data was manipulated. As a result, I decided to examine the source code in detail and develop a custom data interface for defining the grid data.

4.2 Niagara Data Interface

What I learnt when analysing the existing Niagara Data interfaces (NDI) was that they weren't only built for configuring data for your Niagara System but it exposes much more functionality and provides much needed insight into the lower-level aspects of a Niagara System. While Unreal Engine generally offers good documentation, community guides, and support, I found that finding resources for developing custom NDIs was not well documented. In fact, I was only able to find a brief introduction [21], a reference guide [38] that primarily relayed comments from the source code, and a blog post [63] describing how a team created a custom NDI for parsing Gaussian splat data. The blog post provided the most insight; however, it was clearly built for a different purpose and it referenced deprecated functions. Due to the lack of documentation, I found myself spending a significant amount of time analysing the existing NDIs to try to understand their structure and behaviour.

4.2.1 Background

Before we go through the implementation, I want to explain how NDIs integrate with the overall system at a high level. Figure 4.3 adapts a diagram from the blog post [63] to illustrate how the NDI exists within the Niagara System and how it interacts with the GPU. The term *render-thread*, used throughout this chapter, refers specifically to execution on the GPU as opposed to the CPU. The NDI will communicate to the GPU via render-thread calls in order to update and sample from grid data. Once particles are updated with the grid data, the rendering pipeline will display the particles to the screen.

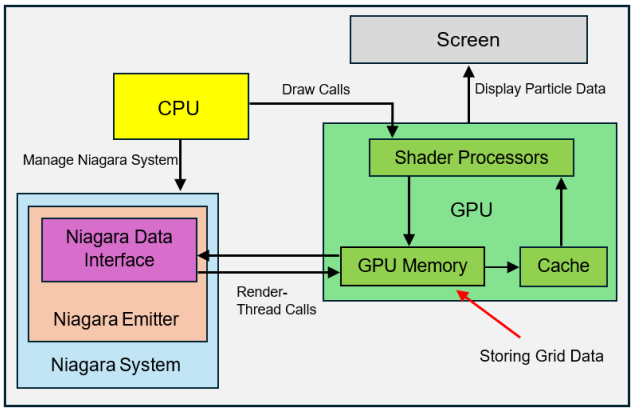


Figure 4.3: Niagara Data Interface integration with the Niagara system

4.2.2 Defining the Data

I will start by defining structs for both the render-thread (GPU data) with figure 4.4.a, and the game-thread (CPU data) with figure 4.4.b. You will notice that Unreal Engine (UE) almost exclusively uses C++ for code and also comes with lots of UE specific data structures and data types. I will try to reduce the occurrence of irrelevant code and boilerplate; however, if the coding style is unclear, the Unreal coding standard [34] will provide a good overview. I will place these structs within the same header file as the main Niagara Data Interface (NDI).

```
struct FNDIAuroraInstanceDataRenderThread
{
    void ResizeBuffers(FRDGBuilder& GraphBuilder);

    FIntVector NumCells = FIntVector(64, 64, 64);
    FVector WorldBBoxSize;
    float CellSize = 0.0f;

    bool bResizeBuffers = false;

    FNiagaraPooledRWTexture PlasmaPotentialTextureRead;
    FNiagaraPooledRWTexture PlasmaPotentialTextureWrite;
    FNiagaraPooledRWTexture NumberDensityTexture;
    FNiagaraPooledRWTexture ChargeDensityTexture;
    FNiagaraPooledRWTexture ElectricFieldTexture;
    FNiagaraPooledRWTexture VectorFieldTexture;
    FNiagaraPooledRWTexture CopyTexture;
};
```

(4.4.a) Render Thread Data (GPU)

```
struct FNDIAuroraInstanceDataGameThread
{
    FIntVector NumCells = FIntVector(64, 64, 64);
    FVector WorldBBoxSize;
    bool bNeedsRealloc = false;
    bool bBoundsChanged = false;
};
```

(4.4.b) Game Thread Data (CPU)

The render-thread struct stores various data on the GPU such as grid properties and the attributes stored at each grid node. The attributes are stored as **FNiagaraPooledRWTexture** data structures. Although it may seem unusual to store data in textures, this approach is commonplace in GPU computing, especially for general-purpose applications (GPGPU). GPUs have naturally evolved from its origin to be highly optimised for storing and sampling from textures. By exploiting this aspect of GPUs, I can efficiently manage and access grid data within the simulation.

In addition to the grid attribute textures, I will store grid properties such as the number of cells and the physical bounding box size of the grid for both the GPU and CPU. The reason I store them on the CPU is to be able to easily test different values for them, from the Niagara Editor. If I only kept the properties on the render-thread, I would have to manually re-build the project every time I wanted to change those properties. This extensibility comes with added effort as I will have to manually propagate the grid properties from the game-thread to the render-thread, which will be explained later.

```
UCLASS(EditInlineNew, Category = "Aurora", meta = (DisplayName = "Aurora Data"))
0 derived Blueprint classes
class AURORA_API UNiagaraDataInterfaceAurora : public UNiagaraDataInterfaceRWBase
{
    GENERATED_BODY()

    UNiagaraDataInterfaceAurora();

protected:
    /* storing game-thread data instances */
    TMap<FNiagaraSystemInstanceID, FNDIAuroraInstanceDataGameThread*> SystemInstancesToProxyData_GT;
}
```

Figure 4.5: Custom Niagara Data Interface class definition

With figure 4.5, I define the actual NDI class. Within this class, I store a map of the game-thread data instances. This means that whenever I instantiate an NDI instance, I should create a new instance of the game-thread struct and store it in this map with the Niagara System ID as the key. Just like the game-thread, I also want a map for instances of the render-thread; however, instead of storing this in the NDI class, this will be defined in a separate struct specifically for managing data on the GPU, called a *Proxy*. This is shown in figure 4.6. Currently, it only stores the map I just described; however, it will be extended with more functionality in a later section.

```cpp
struct FNiagaraDataInterfaceProxyAurora : public FNiagaraDataInterfaceProxyRW
{
    FNiagaraDataInterfaceProxyAurora() {}

    TMap<FNiagaraSystemInstanceID, FNDIAuroraInstanceDataRenderThread> SystemInstancesToProxyData;
};
```

Figure 4.6: Struct designed to manage data on the GPU

Initialising an NDI instance with data: Conveniently, the base class for the Niagara Data Interface contains a function that runs upon instantiating an NDI, `InitPerInstanceData()`. I will use this function to create an instance of the game-thread and render-thread for the given instance and initialise them with default values.

```cpp
bool UNiagaraDataInterfaceAurora::InitPerInstanceData(void* PerInstanceData, FNiagaraSystemInstance* SystemInstance)
{
    FNDIAuroraInstanceDataGameThread* InstanceData = new (PerInstanceData) FNDIAuroraInstanceDataGameThread();
    SystemInstancesToProxyData_GT.Emplace(SystemInstance->GetId(), InstanceData);

    if ((NumCells.X * NumCells.Y * NumCells.Z) == 0 || (NumCells.X * NumCells.Y * NumCells.Z) > GetMaxBufferDimension()) { ... }

    InstanceData->WorldBBoxSize = WorldBBoxSize;
    InstanceData->NumCells = NumCells;

    FNiagaraDataInterfaceProxyAurora* LocalProxy = GetProxyAs<FNiagaraDataInterfaceProxyAurora>();

    ENQUEUE_RENDER_COMMAND(FInitData)(
        [LocalProxy, RT_InstanceData = *InstanceData, InstanceID = SystemInstance->GetId()](FRHICommandListImmediate& RHICmdList)
        {
            check(!LocalProxy->SystemInstancesToProxyData.Contains(InstanceID));
            FNDIAuroraInstanceDataRenderThread* TargetData = &LocalProxy->SystemInstancesToProxyData.Add(InstanceID);

            TargetData->NumCells = RT_InstanceData.NumCells;
            TargetData->WorldBBoxSize = RT_InstanceData.WorldBBoxSize;
            TargetData->CellSize = static_cast<float>(RT_InstanceData.WorldBBoxSize.X / RT_InstanceData.NumCells.X);
            TargetData->bResizeBuffers = true;
        });
    return true;
}
```

Figure 4.7: Initialising game-thread and render-thread data for an NDI instance

In figure 4.7, I first instantiate an instance of the game-thread struct and add it to the corresponding map. I then assign the default bounding box size and number of cells. For the render-thread data, I get the current proxy associated with the NDI and we issue a render-thread command. Unreal Engine separates the game and render thread execution; therefore, in order to communicate from the CPU to the GPU, I must add commands to a queue that will be fed to the render-thread. The instruction,`ENQUEUE_RENDER_COMMAND`, does exactly this. I define a lambda expression containing instructions to (1) get the current proxy, (2) add a new render-thread instance to its map, (3) assign default data and indicate that the textures need to be reset with `bResizeBuffers`.

Before I move onto the main simulation, I wanted to quickly run through other functions defined in the NDI that, although have their purpose, will not be covered in detail any further.

- `DestroyPerInstanceData()` - cleans up the per-instance data on game and render thread when we destroy a Niagara Data Interface instance.
- `PerInstanceTickPostSimulate()` - after every tick, we check if the textures need resizing or the grid size changed and will propagate that the data needs updating on the render-thread.
- `PostInitProperties()` - ran after the constructor and property initialisation, where it is used to register the NDI.
- `Equals()` - helper function for checking if two NDI instances are equal.
- `CopyToInternal()` - helper function for copying the state of an NDI to another.
- `AppendCompileHash()` - adding a hash to the compile process ensuring that changes made forces a recompilation of the shader.

4.3 The Simulation Pipeline

Now that the underlying data for the simulation has been defined, I now have the necessities in order to begin the actual simulation implementation. I will begin with a high-level visual schematic of the simulation process. Afterwards, I will go into detail with how it will be implemented.

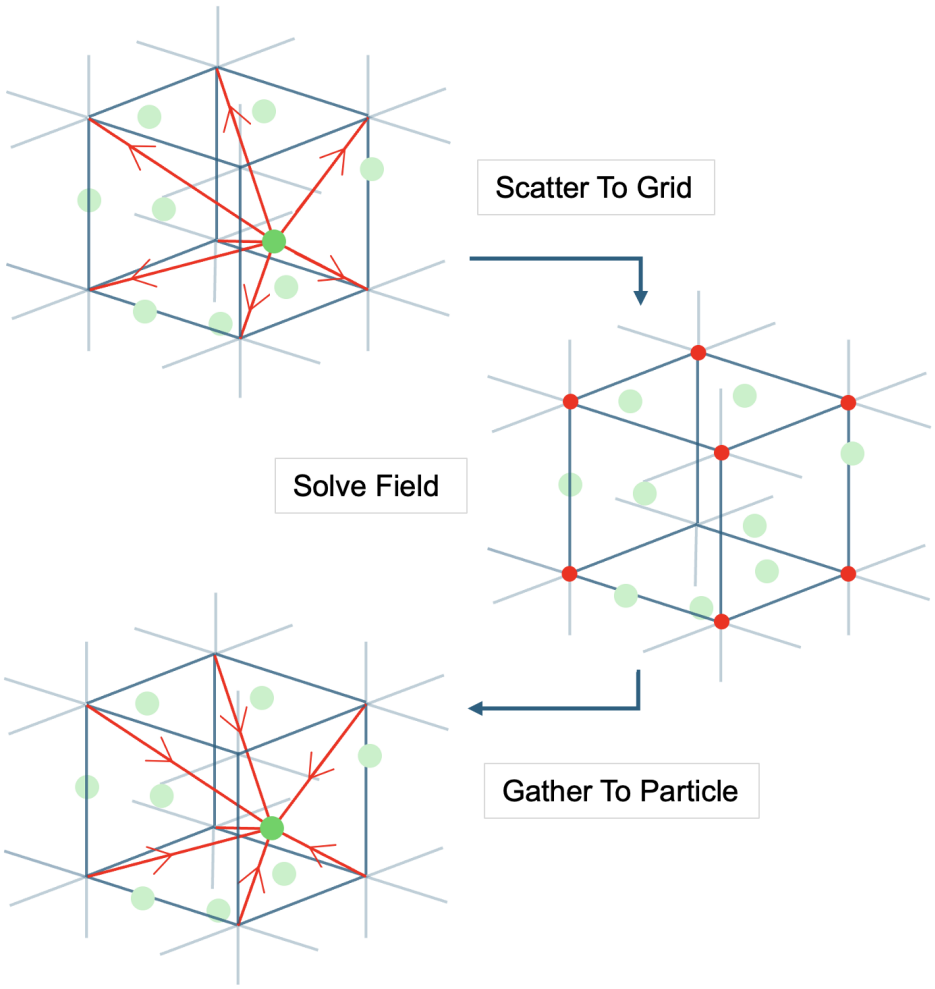


Figure 4.8: Simplified visual of the simulation process

The simulation can be conceptualised in three distinct stages. In the first stage, we iterate over each particle and scatter their density to the surrounding grid nodes. The second stage, executed for each grid node, consists of three separate shader calls in order to solve the electric field. Once the electric field is computed, the final stage involves iterating through each particle, gathering the electric field from the surrounding grid nodes, and integrating that force into the particle's motion. In order to implement these stages I must execute them as `Simulation Stages`, as referenced with the green modules in figure 4.2.a. Since shaders have not been discussed previously, the next section introduces them and how they will be used to implement these stages.

4.3.1 Working with Shaders

Compute Shaders were introduced to GPUs to enable general-purpose, massively parallel processing. They are extensively used in particle simulations, since the instructions are executed for each particle, we can just execute them in parallel (SIMD). Compute shaders will be written in High-Level Shading Language (HLSL), which we define in our custom NDI. Before I get onto the actual shader code, I will run through how I implemented compute shader capability in the Niagara system. Unreal Engine abstracts away defining some of the boilerplate for dispatching the compute shader; however, there are still important aspects that must be incorporated into the NDI to make it compute shader ready. For further insight into the dispatch mechanism implemented by Unreal Engine, I found the `NiagaraGpuComputeDispatch.cpp` file in the source code to be particularly useful.

```
virtual bool CanExecuteOnTarget(ENiagaraSimTarget Target) const override { return Target == ENiagaraSimTarget::GPUComputeSim; }
virtual ENiagaraGpuDispatchType GetGpuDispatchType() const override { return ENiagaraGpuDispatchType::ThreeD; }
virtual FIntVector GetGpuDispatchNumThreads() const { return FIntVector(4, 4, 4); }
```

Figure 4.9: Configuring compute shader settings

In figure 4.9, I have outlined several critical settings that I set inside the custom NDI in order to enable compute-shader capable, GPU-based simulation. First, I set the simulation target to GPU compute, ensuring that it operates with the GPU rather than the CPU. Next, the dispatch type is explicitly defined to be 3D, meaning the compute shader will operate over three spatial dimensions, which is important when dealing with volumetric data as this simulation will. Additionally, the last function specifies the number of threads to be executed in each dimension, thereby defining the thread-group size. These combined settings ensure support for compute shaders that iterate over 3D data.

4.3.2 Initialising Textures

```
ElectricFieldTexture.Release();
const FRDGTextureDesc EFieldTextureDesc = FRDGTextureDesc::Create3D(
            FIntVector(NumCells.X, NumCells.Y, NumCells.Z),
            PF_A32B32G32R32F,
            FClearValueBinding::Black,
            ETextureCreateFlags::ShaderResource | ETextureCreateFlags::Dynamic | ETextureCreateFlags::UAV);
```

Figure 4.10: A snippet of the `ResizeBuffers()` function in the proxy

With the compute shaders expecting 3D data to iterate over, I will now briefly cover how to initialise our textures to be 3D - volumetric. There are certain advantages that comes with dispatching threads in 3D and having 3D data, such as better memory coalescing, easier indexing, and better thread utilisation. For example, certain stages in the simulation require grid nodes to access data from neighbouring nodes; with threadgroups in 3D, their accesses to neighbour nodes are more likely to be coalesced (combined into one memory transaction), improving performance. Figure 4.10 shows a snippet from the `ResizeBuffers()` function defined in the render-thread struct, which outlines the texture initialization process. I will define a 'texture description', which specifies the texture's size, pixel format, default value, and usage flags. For example, the Electric Field texture, which stores vector data, is packed into a `PF_A32B32G32R32F` pixel format (four 32-bit floats representing RGBA channels). A clear value of 'Black' represents zeroed out data. Lastly, the usage flags indicate that the texture is shader-accessible, updated every frame (dynamic), and that we require an Unordered Access View (UAV) of the data. UAVs are required if I want write-access to the textures, which applies to all the textures.

4.3.3 Shader Function and Shader Parameters Setup

Each simulation stage will be captured within a shader function - a function executed on the GPU. For each shader function, I must add important information to two functions, `GetFunctionsInternal()`, where I add the function's signature, and `GetFunctionHLSL()`, where I add the actual shader code. On top of the individual inputs/outputs for the shader functions, I will also define shader parameters. These parameters can be accessed from any shader function and require a specific method of definition and assignment.

```
BEGIN_SHADER_PARAMETER_STRUCT(FShaderParameters, )
    SHADER_PARAMETER(FIntVector,                          NumCells)
    SHADER_PARAMETER(FVector3f,                           CellSize)
    SHADER_PARAMETER(FVector3f,                           WorldBBoxSize)
    SHADER_PARAMETER_RDG_TEXTURE_UAV(RWTexture3D<float>,  PlasmaPotentialWrite)
    SHADER_PARAMETER_RDG_TEXTURE_SRV(Texture3D<float>,    PlasmaPotentialRead)
    SHADER_PARAMETER_RDG_TEXTURE_UAV(RWTexture3D<uint>,   NumberDensityWrite)
    SHADER_PARAMETER_RDG_TEXTURE_SRV(Texture3D<uint>,     NumberDensityRead)
    SHADER_PARAMETER_RDG_TEXTURE_UAV(RWTexture3D<float>,  ChargeDensityWrite)
    SHADER_PARAMETER_RDG_TEXTURE_SRV(Texture3D<float>,    ChargeDensityRead)
    SHADER_PARAMETER_RDG_TEXTURE_UAV(RWTexture3D<float4>, ElectricFieldWrite)
    SHADER_PARAMETER_RDG_TEXTURE_SRV(Texture3D<float4>,   ElectricFieldRead)
END_SHADER_PARAMETER_STRUCT()
```

Figure 4.11: Struct to define all shader parameters

In figure 4.11, I have included how to define the shader parameters. I firstly specify the grid properties, then for each texture I specify an Unordered Access View (UAV) and a Shader Resource View (SRV). If I want to access a texture for reading, I use the SRV parameter, for writing I specify the UAV. The reason why it's important to specify read/write access is not only for readability but it mitigates the potential issue of race conditions - reading/writing to the same memory location.

4.3.4 Scatter To Grid

I will start with the "ScatterToGrid" stage as the first shader function. Shown in figure 4.12, I must first define the function signature. A function signature is just a formal description of the function. Inputs and outputs are both specified as parameters in shader functions in UE. I will also specify important details about the function such as whether it will be a write-only function, and that it only executes on the GPU.

```
FNiagaraFunctionSignature ScatterToGridSig;
ScatterToGridSig.Name = ScatterToGridFunctionName;
ScatterToGridSig.Inputs.Add(FNiagaraVariable(GetClass(), TEXT("Aurora")));
ScatterToGridSig.Inputs.Add(FNiagaraVariable(FNiagaraTypeDefinition::GetVec3Def(), TEXT("InPosition")));
ScatterToGridSig.Inputs.Add(FNiagaraVariable(FNiagaraTypeDefinition::GetMatrix4Def(), TEXT("matrx")));
ScatterToGridSig.Inputs.Add(FNiagaraVariable(FNiagaraTypeDefinition::GetFloatDef(), TEXT("mpw")));
ScatterToGridSig.bMemberFunction = true;
ScatterToGridSig.bRequiresContext = false;
ScatterToGridSig.bRequiresExecPin = true;
ScatterToGridSig.bSupportsCPU = false;
ScatterToGridSig.bWriteFunction = true;
ScatterToGridSig.bSupportsGPU = true;
OutFunctions.Add(ScatterToGridSig);
```

Figure 4.12: Scatter function signature

```
void {FunctionName}(float3 InPosition, float4x4 matrx, float mpw, float precision) {
    // 1. converting the particle's position from world-space to unit-space
    float3 UnitIndex = mul(float4(InPosition, 1.0), matrx).xyz;
    float3 IndexF = UnitIndex * {NumCells} - 0.5f;

    // 2. referencing shader parameters with local variables for readability
    int GridSizeX = {NumCells}.x;
    int GridSizeY = {NumCells}.y;
    int GridSizeZ = {NumCells}.z;

    // 3. getting surrounding grid indices and fractional distances
    int i = ((int(IndexF.x) % GridSizeX) + GridSizeX) % GridSizeX;
    int j = ((int(IndexF.y) % GridSizeY) + GridSizeY) % GridSizeY;
    int k = ((int(IndexF.z) % GridSizeZ) + GridSizeZ) % GridSizeZ;
    int i1 = (i + 1) % GridSizeX;
    int j1 = (j + 1) % GridSizeY;
    int k1 = (k + 1) % GridSizeZ;
    float di = IndexF.x - i;
    float dj = IndexF.y - j;
    float dk = IndexF.z - k;

    // 4. computing the weighted density for each grid node
    float w000 = precision * mpw * (1.0f - di) * (1.0f - dj) * (1.0f - dk);
    float w100 = precision * mpw * di * (1.0f - dj) * (1.0f - dk);
    float w110 = precision * mpw * di * dj * (1.0f - dk);
    float w010 = precision * mpw * (1.0f - di) * dj * (1.0f - dk);
    float w001 = precision * mpw * (1.0f - di) * (1.0f - dj) * dk;
    float w101 = precision * mpw * di * (1.0f - dj) * dk;
    float w111 = precision * mpw * di * dj * dk;
    float w011 = precision * mpw * (1.0f - di) * dj * dk;

    // 5. atomically adding the weightings to surrounding grid nodes
    InterlockedAdd({NumberDensityWrite}[uint3(i,j,k)], uint(w000));
    InterlockedAdd({NumberDensityWrite}[uint3(i1,j,k)], uint(w100));
    InterlockedAdd({NumberDensityWrite}[uint3(i1,j1,k)], uint(w110));
    InterlockedAdd({NumberDensityWrite}[uint3(i,j1,k)], uint(w010));
    InterlockedAdd({NumberDensityWrite}[uint3(i,j,k1)], uint(w001));
    InterlockedAdd({NumberDensityWrite}[uint3(i1,j,k1)], uint(w101));
    InterlockedAdd({NumberDensityWrite}[uint3(i1,j1,k1)], uint(w111));
    InterlockedAdd({NumberDensityWrite}[uint3(i,j1,k1)], uint(w011));
}
```

Figure 4.13: Scatter function HLSL code

Figure 4.13 shows the first look of a shader function written in HLSL. HLSL has C-like syntax which eases the writing process. The function specific parameters are specified within the function definition just like normal functions. On the other hand, shader parameters, the function independent parameters, are referenced within the function body with curly braces around them - *NumCells*, *OutputNumberDensity*.

On a side note, in order to actually assign the render-thread data to the shader parameters, I have to define an extra function, `SetShaderParameters()`. Figure 4.14 shows a snippet of this function and how I connect the shader parameters to their actual data. This snippet is for the scatter function, where I first get the render-thread data from the proxy, set the `NumCells` parameter, then assign a UAV for the write-accessible shader parameter for the `NumberDensity` texture. For this function, I will only require a write-access view for the texture; therefore, for the read-access parameter, i just set it to a default SRV texture. There are many other shader parameters to assign values, but this snippet should capture the assignment process.

```
FNDIAuroraInstanceDataRenderThread* InstanceData = DIProxy.SystemInstancesToProxyData.Find(Context.GetSystemInstanceID());
ShaderParameters->NumCells = InstanceData->NumCells;
ShaderParameters->NumberDensityWrite = InstanceData->NumberDensityTexture.GetOrCreateUAV(GraphBuilder);
ShaderParameters->NumberDensityRead = Context.GetComputeDispatchInterface().GetBlackTextureSRV(GraphBuilder, ETextureDimension::Texture3D);
```

Figure 4.14: Snippet from `SetShaderParameters()`

Now I will run through the scatter function code, refer back to the commented sections in figure 4.13:

1. This shader function will iterate over each particle in parallel. In order to execute the 'scatter' method, I need to find the 6 surrounding grid nodes for the current particle. I'll pass the particle's world position as an input to the function; however, in order to relate the position to the grid, it will involve applying a world-to-unit matrix transformation. With the matrix applied to the particle's position, the returned value will be the normalised position of the particle relative to the grid. The full details of how I defined this matrix will be covered in the following section. I then re-scale the normalised coordinates from the [0..1] range to the number of cells range, so [0..64] if there are 64 cells in xyz.

2. With this being the first HLSL code we go through, I wanted to briefly cover creating local variables for shader parameter values for readability. This will typically be done for the grid properties that will be repeatedly called throughout the shader.

3. Given the re-scaled grid indices with `IndexF`, the next job is to find the neighbouring nodes. The integer part of the indices represents the lower corner grid node and from there I'll make increments to get the neighbouring nodes. For example, if `IndexF` was (24.4, 53.4, 3.1), our lower grid node would be (24, 53, 3) stored in `(i,j,k)` and the neighbouring indices stored in `(i1,j1,k1)`. The modulo operator % enforces that grid node's at the boundaries will wrap around to get their neighbour node. Something to notice is that I have an extra set of variables `(di,dj,dk)` which represents the floating point distances between the lower corner and the neighbour nodes. If I apply it to the example, the fractional distances would be (0.4, 0.4, 0.1).

4. I use the fractional distances to apply a 'weighting' to the scatter process. If the particle is closer a grid node, it will scatter more of its macro-particle weight (mpw) to that grid node. The multiplication of the 'precision' value will be explained next.

5. The final section would involve actually adding the weighting values to the surrounding grid nodes. However, when working with parallel execution, potential writes to the same memory location is an easy cause for race conditions. If two or more threads want to write a new value to a grid node's density attribute, there is a chance that the one of the contributions will be lost if these write instructions aren't carefully handled. To mitigate this, I simply use an atomic add operation, `InterlockedAdd()` which completes the add instruction in a thread-safe manner [1]. There is a caveat to this function which is the fact that it can only atomically add the `int` and `uint` data types. If I simply converted the current `float` value to a `uint`, it would lose much of its precision especially if the mpw is a smaller value. In order to conserve precision, I will multiply the value to scatter by a large 'precision' value. Upon reading, I convert the `uint` back to a float and divide by the same 'precision' value. I make the precision value an input to the function to keep it adjustable, as I also want to be careful of overflowing. Unreal Engine's built-in `NiagaraDatatInterfaceRasterizationGrid3D` applies a similar process in order to allow atomic floating point addition.

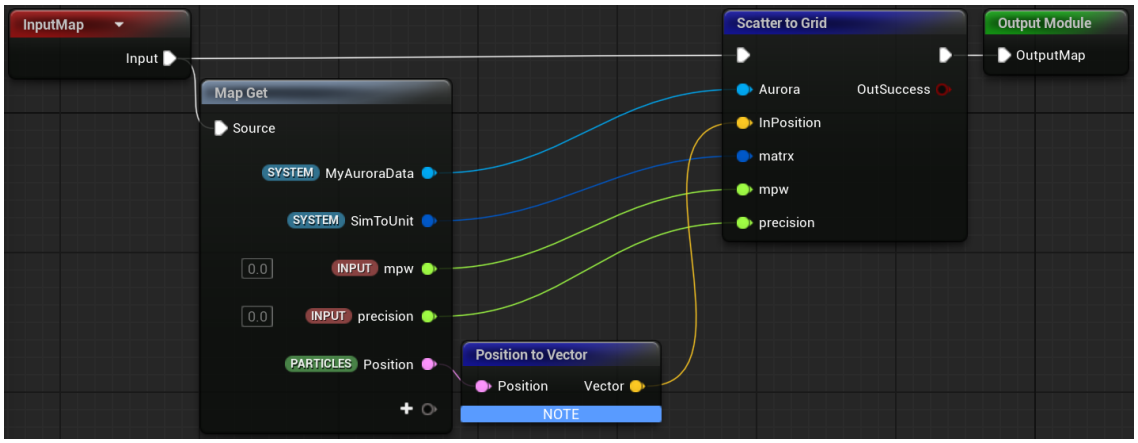


Figure 4.15: Niagara Module Script for Scatter stage

Implementing the Scatter function. Now that I have defined the shader function, all that's left to do is to create a simulation stage in the emitter and to call the function. Figure 4.15 shows a Niagara Module Script which is written using Unreal Engine's node graph. As an overview, execution generally occurs from left to right with inputs defined inside the `Map Get` node. I call the Scatter to Grid function, as Unreal Engine will automatically create a node for the functions you define. I then pipe the relevant inputs to the function's input pins. These include the NDI instance, the world-to-unit matrix, macro-particle weight, precision value, and finally the position of the particle.

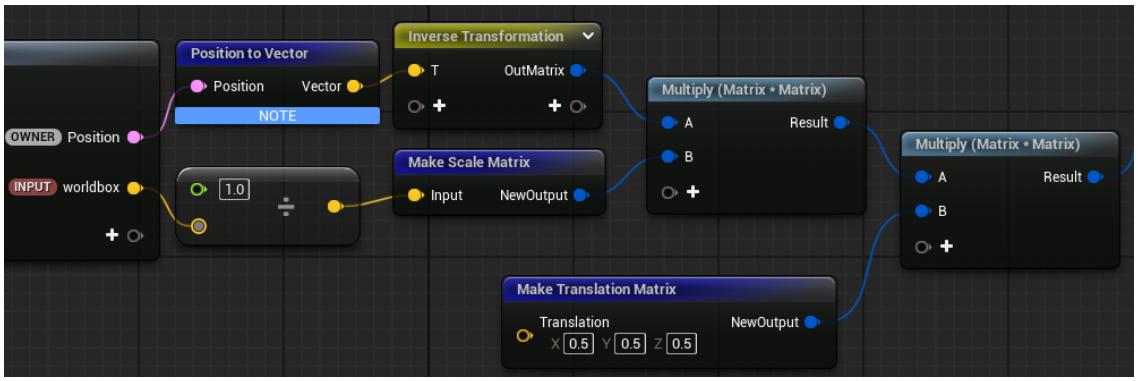
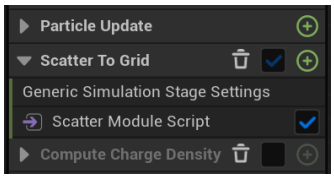
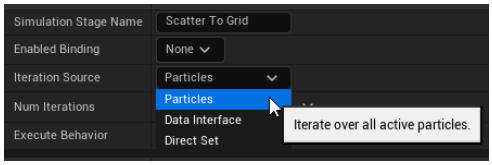


Figure 4.16: Computing the world-to-unit matrix

With figure 4.16, I have shown the process of defining the world-to-unit matrix. I first create an inverse translation matrix which will essentially move the origin to the grid's centre. I then normalise the indices to a [-0.5,0.5] range by applying an inverse scale-matrix of the bounding box of the grid. To move it into a [0..1] range, I apply a final translation of (0.5,0.5,0.5). If you apply this matrix to the world position of a particle, it should correctly transform it into unit space to the grid.



(4.17.a) The 'Scatter' Sim Stage in the Emitter



(4.17.b) Sim Stage Settings

In Figure 4.17.a, I have shown a snippet of the simulation stage, 'Scatter To Grid', being slotted in just after the Particle Update group as our first simulation stage. It runs the Niagara Module Script which I defined in figure 4.15. On the right, with figure 4.17.b, I have the shown the settings for this simulation stage. The 'Iteration Source' choice in particular is important as it determines what to iterate over. For this stage I specify to iterate over Particles; however, in some of the following stages, we will want to iterate over the grid nodes instead, in which case I would choose 'Data Interface' as the iteration source.

4.3.5 Computing Charge Density

If we follow figure 4.8, the stage after 'scatter' is called 'Solve Field'. This field refers to the electric field and it will be slightly more complicated than the single shader function required for the scatter stage. 'Solve Field' will be split into 3 main steps, each with their own shader function and simulation stage.

1. Compute Charge Density: converts the calculated number density into charge density
2. Solve Plasma Potential: solve Poisson equation to obtain the electric potential.
3. Solve Electric Field: use the finite difference method (FDM) to convert potential into the resulting Electric Field.

I will cover the first step here, which will be a relatively lightweight stage. In figure 4.18, I included the function signature for computing the charge density.

```
FNiagaraFunctionSignature ComputeChargeDensitySig;
ComputeChargeDensitySig.Name = NumberDensityToChargeFunctionName;
ComputeChargeDensitySig.Inputs.Add(FNiagaraVariable(FNiagaraTypeDefinition(GetClass()), TEXT("Grid")));
ComputeChargeDensitySig.Inputs.Add(FNiagaraVariable(FNiagaraTypeDefinition::GetFloatDef(), TEXT("Charge")));
ComputeChargeDensitySig.Inputs.Add(FNiagaraVariable(FNiagaraTypeDefinition::GetFloatDef(), TEXT("IonDensity")));
ComputeChargeDensitySig.Inputs.Add(FNiagaraVariable(FNiagaraTypeDefinition::GetFloatDef(), TEXT("Epsilon")));
ComputeChargeDensitySig.Inputs.Add(FNiagaraVariable(FNiagaraTypeDefinition::GetFloatDef(), TEXT("mpw")));
ComputeChargeDensitySig.Inputs.Add(FNiagaraVariable(FNiagaraTypeDefinition::GetFloatDef(), TEXT("precision")));
ComputeChargeDensitySig.bMemberFunction = true;
ComputeChargeDensitySig.bRequiresContext = false;
ComputeChargeDensitySig.bSupportsCPU = false;
ComputeChargeDensitySig.bSupportsGPU = true;
ComputeChargeDensitySig.bRequiresExecPin = true;
OutFunctions.Add(ComputeChargeDensitySig);
```

Figure 4.18: Function Signature for Computing the Charge Density

```
void {FunctionName}(float Charge, float IonDensity, float Epsilon, float mpw, float precision)
{
    int3 GridSize = {NumCells};
    float CellVol = {CellSize}.x * {CellSize}.y * {CellSize}.z;

    #if NIAGARA_DISPATCH_TYPE == NIAGARA_DISPATCH_TYPE_THREE_D
        const int IndexX = GDispatchThreadId.x;
        const int IndexY = GDispatchThreadId.y;
        const int IndexZ = GDispatchThreadId.z;
    #else
        const uint IndexX = Linear % GridSize.x;
        const uint IndexY = (Linear / GridSize.x) % GridSize.y;
        const uint IndexZ = Linear / (GridSize.x * GridSize.y);
    #endif

    float NumDens = (float({NumberDensityRead}.Load(uint4(IndexX, IndexY, IndexZ, 0))) / precision);
    float toDensity = NumDens / CellVol;
    float ChargeDensity = ((toDensity * Charge) - IonDensity) / Epsilon;
    {ChargeDensityWrite}[uint3(IndexX, IndexY, IndexZ)] = ChargeDensity;
}
```

Figure 4.19: Compute Charge Density code

- After defining some initial variables, I want to define variables for 3D indices of the current grid node. A technique used when dispatching in 3D is to use the threadID as the indices. This is one of the benefits I mentioned previously for dispatching in 3D for volumetric data. The preprocessor enforces that if I decided to dispatch in 1D, that it would involve having to manually convert the 1D index into its 3D components which adds some overhead.
- The last section involves the actual computation. When I read data from the number density texture, I have to make sure to run the reverse process I applied for scatter. This involves converting the value to a float and dividing by the precision. To read from a texture, I use the `Load()` function. The result will be divided by the cell volume to get the density; it will be multiplied by the charge of the particle; I then subtract the ion density to ensure quasi-neutrality, since I won't be manually simulate ions. The result is then divided by `Epsilon` which represents the permittivity of a vacuum and is done as part of solving the Poisson equation. The result is stored in the charge density texture.

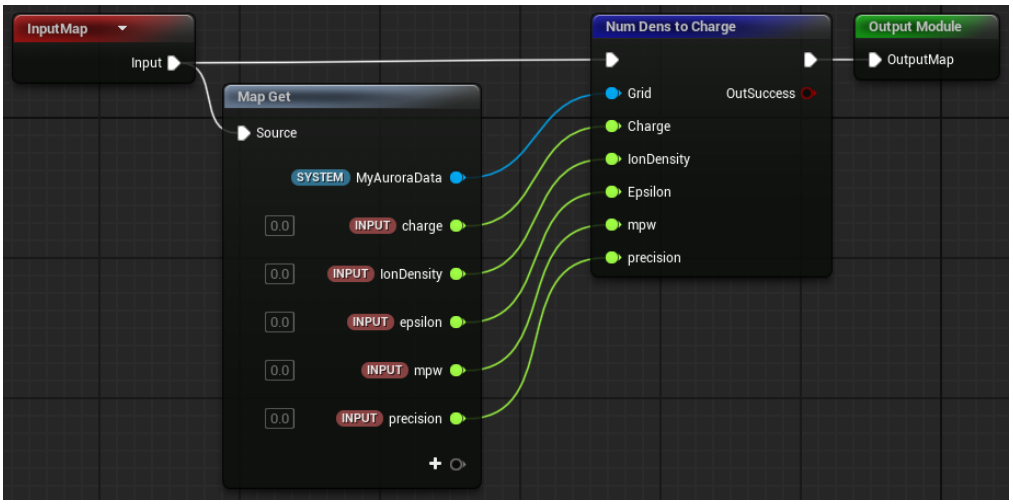


Figure 4.20: Niagara Module Script for Computing Charge Density

4.3.6 Solve Plasma Potential

The next stage is considered one of the more complicated simulation stages. It is in charge of solving the Poisson equation to obtain the potential at each grid node. I will run through the implementation process which include some of the iterations I tested.

```
void {FunctionName}()
{
    int3 GridSize = {NumCells};
    float dx2 = {CellSize}.x * {CellSize}.x;

    const int IndexX = GDispatchThreadId.x;
    const int IndexY = GDispatchThreadId.y;
    const int IndexZ = GDispatchThreadId.z;

    // getting neighbour indices and their potential values
    int ileft = (IndexX - 1 + GridSize.x) % GridSize.x;
    int iright = (IndexX + 1) % GridSize.x;
    int idown = (IndexY - 1 + GridSize.y) % GridSize.y;
    int iup = (IndexY + 1) % GridSize.y;
    int iback = (IndexZ - 1 + GridSize.z) % GridSize.z;
    int ifront = (IndexZ + 1) % GridSize.z;

    float left = {PlasmaPotentialRead}.Load(uint4(ileft, IndexY, IndexZ, 0));
    float right = {PlasmaPotentialRead}.Load(uint4(iright, IndexY, IndexZ, 0));
    float down = {PlasmaPotentialRead}.Load(uint4(IndexX, idown, IndexZ, 0));
    float up = {PlasmaPotentialRead}.Load(uint4(IndexX, iup, IndexZ, 0));
    float back = {PlasmaPotentialRead}.Load(uint4(IndexX, IndexY, iback, 0));
    float front = {PlasmaPotentialRead}.Load(uint4(IndexX, IndexY, ifront, 0));

    // executing the jacobi update
    float ChargeD = {ChargeDensity}.Load(uint4(IndexX, IndexY, IndexZ, 0));
    float NewPhi = (left + right + down + up + back + front + dx2 * ChargeD) / 6.0f;
    {PlasmaPotentialWrite}[uint3(IndexX, IndexY, IndexZ)] = NewPhi;
}
```

Figure 4.21: Simple Jacobi Solver

Jacobi Iteration Method: I started with a basic method to solve the Poisson equation, shown with figure 4.21. It involves sampling the charge density from the neighbour grid nodes, accumulating them, adding the squared cell size, and dividing by 6. Something you might notice, is that it seems like I'm reading and writing from the same `PlasmaPotential` texture. The following section explains how I prevent race conditions:

The method I used was to define two separate textures. In one iteration I read from one and write to the other. Then I would 'swap' the textures, so that in the next iteration, the texture I just wrote to would now be read from. This doesn't require any atomic operations because there aren't any race conditions with the writing stage specifically. Figure 4.22 shows a simplified visual of this process. The question now is how do I get the textures to swap between each iteration.

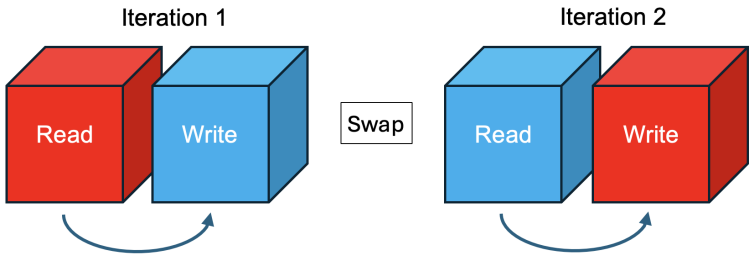


Figure 4.22: Visual of the swapping mechanism

```
struct FNiagaraDataInterfaceProxyAurora : public FNiagaraDataInterfaceProxyRW
{
    FNiagaraDataInterfaceProxyAurora() {}

    // controlling execution
    virtual void ResetData(const FNDIGpuComputeResetContext& Context) override;
    virtual void PreStage(const FNDIGpuComputePreStageContext& Context) override;
    virtual void PostStage(const FNDIGpuComputePostStageContext& Context) override;
    virtual void PostSimulate(const FNDIGpuComputePostSimulateContext& Context) override;

    // how many elements we iterate over in 3D
    virtual void GetDispatchArgs(const FNDIGpuComputeDispatchArgsGenContext& Context) override;

    TMap<FNiagaraSystemInstanceID, FNDIAuroraInstanceDataRenderThread> SystemInstancesToProxyData;
};
```

Figure 4.23

Up until now I have omitted certain details about how I can further customise the execution process. Defining a custom NDI provides access to important functions that give you much more control with simulation stages. The proxy was introduced with figure 4.6, where it currently stores instances of the GPU data; however, in this section, I will build upon this struct by defining several more functions.

1. `Controlling Execution:` The following functions provide the ability to run instructions between stages and iterations. For example, with the current Jacobi solver, I need to find a way to swap textures in between iterations when running the 'solve plasma potential' stage. Another use would be to make sure to reset the Number Density texture at the end of each tick. Unlike the other textures, the values from this texture aren't overridden but simply added to; therefore, to avoid accumulating density from previous ticks, we must reset it. To give a high-level overview of how these functions are incorporated into the simulation tick, refer to figure 4.24, which expands on figure 4.2.b.

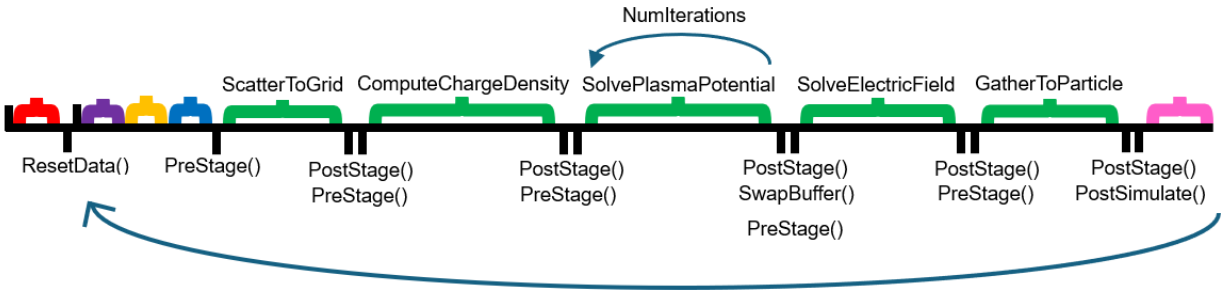


Figure 4.24: More detailed view of a simulation tick

- `ResetData()` - This function is called once, at very beginning of the simulation to clear the textures to their default values. It runs several `AddClearUAVPass()` functions which efficiently clear out the texture to a specified value.
- `PreStage()` - This function allows you to specify instructions to run just before a stage or iteration. For my simulation, it will be used to check if the textures require resizing and subsequently runs the `ResizeBuffers()` function if we require it.
- `PostStage()` - This function allows you to specify instructions to run after a stage or iteration. This is used to implement the texture swapping mechanism as part of the `SolvePlasmaPotential` simulation stage. It first checks if we are within the correct stage, if we are we grab the render-thread data and swap the pointers so that they point to the opposite texture. This is efficient and doesn't require swapping any of the data.

```
void FNiagaraDataInterfaceProxyAurora::PostStage(const FNDIGpuComputePostStageContext& Context)
{
    if (Context.GetSimStageData().StageMetaData->SimulationStageName == TEXT("Solve Plasma Potential"))
    {
        FNDIAuroraInstanceDataRenderThread* ProxyData = SystemInstancesToProxyData.Find(Context.GetSystemInstanceID());
        Swap(ProxyData->PlasmaPotentialTextureRead, ProxyData->PlasmaPotentialTextureWrite);
    }
}
```

Figure 4.25: Implementing texture swapping in `PostStage()`

- `PostSimulate()` - This function is ran at the end of each tick and is used to finalise data for the next tick. Here I will run an `AddClearUAVPass()` to the Number Density texture in order to reset it.

2. `GetDispatchArgs`: For the following simulation stages: `ComputeChargeDensity`, `SolvePlasmaPotential`, and `SolveElectricField`, I want to iterate over each grid node as opposed to every particle. This requires to set the Iteration Source to 'Data Interface', seen in figure 4.17.b. However, I will have to manually set the number of elements to iterate over with the `GetDispatchArgs()` function. It's quite simple as I just set it to the number of cells in 3D.

Optimising: The Jacobi Solver is an iterative algorithm and therefore requires multiple iterations to converge successfully. Currently, I can set the number of iterations but this number is something we can only predefine. To improve the algorithm, I can try to add a convergence check within the shader. Adding a convergence check would be easy if I was implementing this on the CPU; however, executing this on the GPU complicates things as I don't have access to global synchronisation. If HLSL provided a barrier for all threads, it would allow easy integration for the convergence check but it isn't so simple. I also can't directly set the shader to run infinitely until it converges, there has to be maximum.

With all this in mind, the method I attempted to use was ultimately too constrained to use. However, I will run through it so as to inform the final iteration of the potential solver shader. For this attempt, I decided to put the the convergence check at the beginning of the shader, this is because I know that the data will be synchronised between compute shader calls. This is essentially the same as checking for convergence for the previous iteration's results. If convergence has successfully completed, the shader exits early without running any of the other computations. Figure 4.26 shows the attempt.

To provide some more context with this implementation, the important part is to compute the 'residual' value for a given iteration. The residual is simply the difference between the newly calculated potential value and the previous potential value. In order to store the residual values between iterations, I will create new shader parameters, `MaxResidualWrite` and `MaxResidualRead`. Just like with the potential textures, I will be reading and writing within the same shader; therefore, I will require separate textures and implement swapping for these as well. This method also requires you to set the initial max residual for the first iteration to a value higher than the tolerance, so we don't converge immediately. I will run through the two added steps in the code with figure 4.26.

1. At the start of the shader, I immediately check the previous iteration's maximum residual value, if it's successful we exit the shader immediately skipping the rest of the computations.

2. If the convergence isn't successful yet, I execute the algorithm as normal and compute the new potential. I calculate the new residual and atomically assign it to the `MaxResidual` parameter if it's greater than the current residual.

```
void {FunctionName}(float Tolerance)
{
    // 1. convergence check with early exit if successful
    float CurrentMaxResidual = asfloat({MaxResidualRead}[0]);
    if (CurrentMaxResidual < Tolerance)
        return;

    int3 GridSize = {NumCells};
    float dx2 = {CellSize}.x * {CellSize}.x;

    const int IndexX = GDispatchThreadId.x;
    const int IndexY = GDispatchThreadId.y;
    const int IndexZ = GDispatchThreadId.z;

    // getting neighbour's potential values
    float left = {PlasmaPotentialRead}.Load(uint4((IndexX - 1 + GridSize.x) % GridSize.x, IndexY, IndexZ, 0));
    float right = {PlasmaPotentialRead}.Load(uint4((IndexX + 1) % GridSize.x, IndexY, IndexZ, 0));
    float down = {PlasmaPotentialRead}.Load(uint4(IndexX, (IndexY - 1 + GridSize.y) % GridSize.y, IndexZ, 0));
    float up = {PlasmaPotentialRead}.Load(uint4(IndexX, (IndexY + 1) % GridSize.y, IndexZ, 0));
    float back = {PlasmaPotentialRead}.Load(uint4(IndexX, IndexY, (IndexZ - 1 + GridSize.z) % GridSize.z, 0));
    float front = {PlasmaPotentialRead}.Load(uint4(IndexX, IndexY, (IndexZ + 1) % GridSize.z, 0));

    // executing the jacobi update
    float ChargeD = {ChargeDensity}.Load(uint4(IndexX, IndexY, IndexZ, 0));
    float OldPhi = {PlasmaPotentialRead}.Load(uint4(IndexX, IndexY, IndexZ, 0));
    float NewPhi = (left + right + down + up + back + front + dx2 * ChargeD) / 6.0f;
    {PlasmaPotentialWrite}[uint3(IndexX, IndexY, IndexZ)] = NewPhi;

    // 2. calculating new residual value
    float Residual = abs(NewPhi - OldPhi);
    uint ResidualAsuint = asuint(Residual);
    InterlockedMax({MaxResidualWrite}[0], ResidualAsuint);
}
```

Figure 4.26: Potential Optimisation for Solving Plasma Potential

Optimisation Suitability: At first glance, this improves the current implementation because once the algorithm has converged, we drastically reduce the computation for any further iterations. However, there are a few bottlenecks with this solution. Firstly, dynamically stopping shader execution is not possible with the current workflow, therefore I would still have to execute all the iterations even if we have converged. I investigated certain ways to see we could stop execution of the shader dynamically; however, changing simulation stage information dynamically wasn't possible with the current model. Therefore, I will have to execute the maximum number of iterations either way which creates a bottleneck from the overhead with dispatching shaders and synchronisation.

The second bottleneck comes from the use of the atomic operation. Before I talk about its efficiency, I wanted to briefly comment on a neat trick used to overcome the constraint of having to use uint/int data types with atomic functions. The `asuint()` function allows you to store a value within the format of a uint. I used this with the float residual value to maintain the precision. When reading, I use `asfloat()` to convert the value back to a float. The reason why I couldn't use this trick with the scatter stage, and instead used a precision multiplier is because this doesn't work when applying computation such atomic add operations. With comparing for the 'Max' value, the comparison operators still work. The comparison only works with positive values since we use `uint`, but since the residual value is calculated with `abs()` this is fine. Although these optimisations are overcome certain issues, the bottleneck has to do with using the atomic operation in the first place. An atomic operation ensures a guaranteed write but it does so using locks. Since every thread wants to check their residual value against the current maximum, that means all threads are contending with the same memory location, causing a hotspot. This effectively degrades performance, as these threads will run the atomic operation sequentially, defeating the point of parallelism.

Final Method: Instead, I decided to try to optimise the solver itself to reduce the time for convergence. I will implement the red-black Gauss Seidel method. I'll run through how Guass-Seidel simply improves upon the Jacobi solver, and then explain why we require this red-black scheme to overcome issues when implementing the algorithm in a parallel context.

In a CPU, sequential based approach, the Guass Seidel approach essentially makes the simple change from having separate textures to reading and writing to the same texture. By doing so, you naturally use more updated values during an iteration, instead of only using the previous iteration's values. However, I have covered why this isn't possible as this isn't a sequential context and it invokes the possibility for race conditions. The red-black scheme separates the solver into two separate calls; the first call solves the potential for half of the grid, the second call solves for the other half. Even though it may seem like this is halving the throughput, the second half of the solver updates the potential using updated potentials rather than the old values, effectively speeding up convergence. I separate the grid nodes into halves, 'red' and 'black'. Figure 4.27 shows how the nodes are separated. Nodes shouldn't read from a node that will be written to, therefore by alternating execution of nodes, we don't violate this rule. This also means that I don't require separate texture for the potential any longer. This not only reduces memory consumption but it reduces overhead of having to swap the buffers.

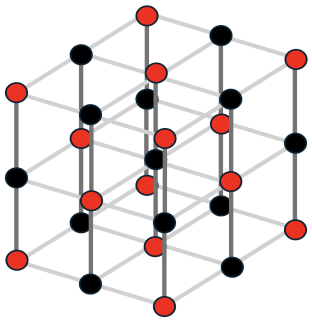


Figure 4.27: Visual for the red, black separation of grid nodes

```
void {FunctionName}()
{
    const int IndexX = GDispatchThreadId.x;
    const int IndexY = GDispatchThreadId.y;
    const int IndexZ = GDispatchThreadId.z;

    // Checking if the current node is red/black
    if ((IndexX + IndexY + IndexZ) % 2 == {RedBlackCheck}) return;

    int3 GridSize = {NumCells};
    float dx = {CellSize}.x;
    float dx2 = dx * dx;

    int ileft = (IndexX - 1 + GridSize.x) % GridSize.x;
    int iright = (IndexX + 1) % GridSize.x;
    int idown = (IndexY - 1 + GridSize.y) % GridSize.y;
    int iup = (IndexY + 1) % GridSize.y;
    int iback = (IndexZ - 1 + GridSize.z) % GridSize.z;
    int ifront = (IndexZ + 1) % GridSize.z;
    float left = {PlasmaPotentialRead}.Load(uint4(ileft, IndexY, IndexZ, 0));
    float right = {PlasmaPotentialRead}.Load(uint4(iright, IndexY, IndexZ, 0));
    float down = {PlasmaPotentialRead}.Load(uint4(IndexX, idown, IndexZ, 0));
    float up = {PlasmaPotentialRead}.Load(uint4(IndexX, iup, IndexZ, 0));
    float back = {PlasmaPotentialRead}.Load(uint4(IndexX, IndexY, iback, 0));
    float front = {PlasmaPotentialRead}.Load(uint4(IndexX, IndexY, ifront, 0));

    float ChargeD = {ChargeDensityRead}.Load(uint4(IndexX, IndexY, IndexZ, 0));
    float NewPhi = (left + right + down + up + back + front + dx2 * ChargeD) / 6.0f;

    {PlasmaPotentialWrite}[uint3(IndexX, IndexY, IndexZ)] = NewPhi;
}
```

Figure 4.28: Solving Plasma Potential with Red-Black Guass Seidel

Figure 4.28 shows the resulting solver shader code. It references a new shader parameter `RedBlackCheck` which just alternates between 0 and 1. I set this parameter with the alternating values in the `SetShaderParameters()` function, shown in figure 4.29. Unfortunately, the convergence check from the previous attempt wasn't utilised; however, it provided important insight and limitations to the current model that will be evaluated later. This concludes the solve potential stage.

```
ShaderParameters->RedBlackValue = InstanceData->Counter;
InstanceData->Counter = (InstanceData->Counter + 1) % 2;
```

Figure 4.29: Setting the RedBlackCheck shader parameter

4.3.7 Solve Electric Field

The final part to the 'Solve Field' stage, involves computing the electric field by solving Guass's law as part of the Maxwells equations - $\nabla \cdot E = \frac{\rho}{\epsilon_0}$.

```
void {FunctionName}()
{
    int3 GridSize = {NumCells};
    float dx = {CellSize}.x;
    float inv2dx = 1.0f / (2.0f * dx);

    const int IndexX = GDispatchThreadId.x;
    const int IndexY = GDispatchThreadId.y;
    const int IndexZ = GDispatchThreadId.z;

    int ileft = (IndexX - 1 + GridSize.x) % GridSize.x;
    int iright = (IndexX + 1) % GridSize.x;
    int idown = (IndexY - 1 + GridSize.y) % GridSize.y;
    int iup = (IndexY + 1) % GridSize.y;
    int iback = (IndexZ - 1 + GridSize.z) % GridSize.z;
    int ifront = (IndexZ + 1) % GridSize.z;
    float left = {PlasmaPotentialRead}.Load(uint4(ileft, IndexY, IndexZ, 0));
    float right = {PlasmaPotentialRead}.Load(uint4(iright, IndexY, IndexZ, 0));
    float down = {PlasmaPotentialRead}.Load(uint4(IndexX, idown, IndexZ, 0));
    float up = {PlasmaPotentialRead}.Load(uint4(IndexX, iup, IndexZ, 0));
    float back = {PlasmaPotentialRead}.Load(uint4(IndexX, IndexY, iback, 0));
    float front = {PlasmaPotentialRead}.Load(uint4(IndexX, IndexY, ifront, 0));

    // finite difference
    float Ex = -(left - right) * inv2dx;
    float Ey = -(down - up) * inv2dx;
    float Ez = -(back - front) * inv2dx;

    {ElectricFieldWrite}[uint3(IndexX, IndexY, IndexZ)] = float4(Ex, Ey, Ez, 0.0f);
}
```

Figure 4.30: Solving for the Electric Field

Figure 4.30 shows the resulting code. I want to find the negative gradient of the potential. A simple and popular method, finite difference, finds the negative difference from the neighbouring potential values. You then divide the result by {2dx}; however, in order to avoid 3 division computations, I instead calculate the inverse of {2dx} and multiply that to each axis. Given that the previous stage properly converged, this should result in a stable vector for the particles to interpolate in the next stage.

4.3.8 Gather to Particle

The last stage, 'gather', can be described as the inverse of the scatter stage. As the scatter stage uses trilinear interpolation, I use the same method to sample the surrounding grid nodes.

```
void {FunctionName}(float3 InPos, float4x4 Matrx, out float3 OutVector) {
    float3 UnitIndex = mul(float4(InPos, 1.0f), Matrx).xyz;
    float3 IndexF = UnitIndex * {NumCells} - 0.5f;

    int3 GridSize = {NumCells};

    int i = ((int(IndexF.x) % GridSize.x) + GridSize.x) % GridSize.x;
    int j = ((int(IndexF.y) % GridSize.y) + GridSize.y) % GridSize.y;
    int k = ((int(IndexF.z) % GridSize.z) + GridSize.z) % GridSize.z;
    int i1 = (i + 1) % GridSize.x;
    int j1 = (j + 1) % GridSize.y;
    int k1 = (k + 1) % GridSize.z;
    float di = IndexF.x - i;
    float dj = IndexF.y - j;
    float dk = IndexF.z - k;

    float3 ef000 = {ElectricFieldRead}.Load(uint4(i,j,k,0)).xyz;
    float3 ef100 = {ElectricFieldRead}.Load(uint4(i1,j,k,0)).xyz;
    float3 ef110 = {ElectricFieldRead}.Load(uint4(i1,j1,k,0)).xyz;
    float3 ef010 = {ElectricFieldRead}.Load(uint4(i,j1,k,0)).xyz;
    float3 ef001 = {ElectricFieldRead}.Load(uint4(i,j,k1,0)).xyz;
    float3 ef101 = {ElectricFieldRead}.Load(uint4(i1,j,k1,0)).xyz;
    float3 ef111 = {ElectricFieldRead}.Load(uint4(i1,j1,k1,0)).xyz;
    float3 ef011 = {ElectricFieldRead}.Load(uint4(i,j1,k1,0)).xyz;

    // linearly interpolate the neighbouring electric field vectors
    float3 ef00 = lerp(ef000,ef100,di);
    float3 ef10 = lerp(ef010,ef110,di);
    float3 ef01 = lerp(ef001,ef101,di);
    float3 ef11 = lerp(ef011,ef111,di);
    float3 ef0 = lerp(ef00,ef10,dj);
    float3 ef1 = lerp(ef01,ef11,dj);
    OutVector = lerp(ef0, ef1, dk);
}
```

Figure 4.31: Gather Electric Field for each particle

1. Just like the scatter stage, I transform the particle's position to its grid indices. I then then get the neighbouring node indices and the fractional distance of the particle. The last part of the code is the important part where I linearly interpolate between the neighbours using the `lerp()` function.

2. One thing to notice with this function is the use function outputs. With all previous shader functions, I have yet to require a specific output. Here I want to use the resulting vector to compute the resulting motion of the particle.

3. With the Gather stage implementation, it will involve slightly more than just running the shader function. Figure 4.32 shows a snippet of how to call the gather function and integrate that into the velocity. This involves creating a variable for the electric field and subsequently using that as the output parameter to the gather function. I then implement the Boris Integrator, which involves a 3 step process; apply half the electric force, then the magnetic force, then the other half of the electric force. The actual particle update is done in a separate Niagara Module Script, shown with figure 4.33. This implements a simple forward Euler integration step, to calculate the new position of the particle.

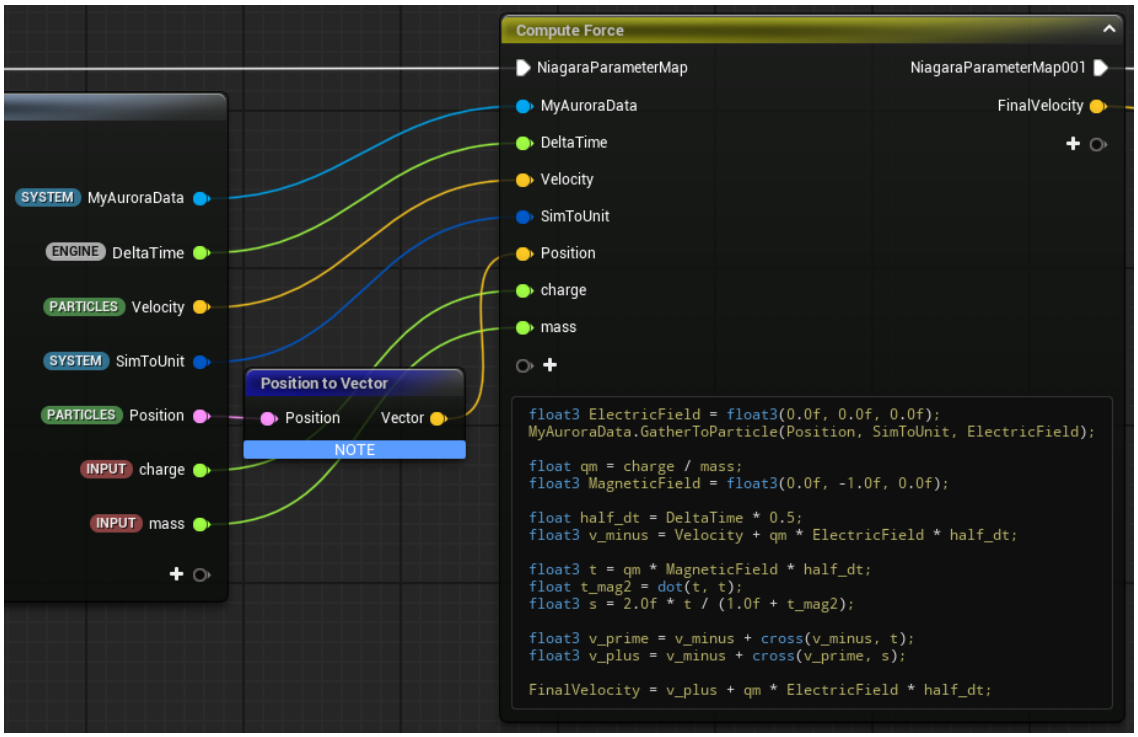


Figure 4.32: Niagara Module Script for Gather stage

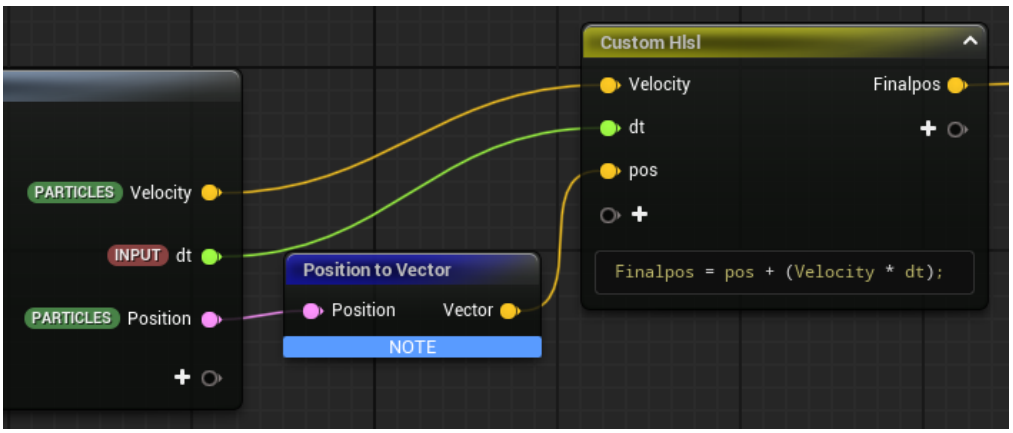


Figure 4.33: Forward Euler integration step

4.4 Testing

1. Initialisation
2. Logging
3. Render Target
4. Stable Solution
5. Profiling

4.4.1 Initialisation

I have set up all the required simulation stages which will execute the steps for a PIC simulation on the GPU. There are some other essential aspects that must be implemented in order to successfully have a running simulation, such as important properties or parameters. With table 5.1, I have defined a list of properties/parameters that exist within different levels of scope in the simulation.

Table 4.1: Niagara Simulation Properties

Property Scope	**Property Name**	**Explanation**
System-Wide		At the highest level of our Niagara System, we have the system properties. These have system-wide scope and any of the emitters can access the properties defined here.
	Fixed Bounds	This is the **bounding box** for the Niagara System which constitutes what gets rendered to the screen; anything outside this box will be culled. I will set it to the size of the grid since I shouldn't need any rendering past the grid.
	Delta Time	The **delta Time** indicates the time allotted for each simulation increment. Physical simulations will often use fixed time-steps for stability and deterministic results which I will follow as well.
	Aurora Data	This is the name of our custom NDI instance. I actually define it at the system-level because the same NDI instance will be referenced in the emitters (I will define multiple emitters).
	World-To-Unit matrix	Mentioned before, but I define the matrix that transforms a world position into unit space relative to our grid at the system-level. This will be updated every tick.
User Parameter		These are similar to System Properties; however, they are read-only within the system but can be modified externally outside the Niagara System.
	NumCells	The number of cells within the grid.
	WorldBBoxSize	The physical size of the bounding box for the grid.
	RenderTarget	I will define Render Targets in section. 4.4.3.
Emitter-Wide		Properties defined within an emitter are only accessible within that emitter. These will usually consist of simulation specific parameters.
	Particle Lifetime	I avoid infinite lifetimes by setting a maximum; the emitter resets once the lifetime has been reached.
	Initial Velocity	Particles will be spawned with an initial velocity, consisting of a small random direction in xy and the majority in the z direction.
	Particle Mass	I will set a mass value for the particle which will affect how strong forces act on it.

Continued on next page

Table 4.1 – *Continued*

Property Type	Property Name	Explanation
	Macro-particle weight	Each particle represents a 'superparticle' and therefore I'll specify the amount of electrons each particle represents.
	Precision	This value is used, in the scatter/compute-charge-density stage, when converting between a float and a uint to preserve the float value's precision. I keep it as a parameter for adjustment.
	Particle Charge	Each particle will have a charge value which affects the forces acting on it.
	Ion Density	This value is subtracted from the overall charge density to maintain quasi-neutrality since I don't directly simulate ions.
	Epsilon	The vacuum permittivity constant. Essentially how much electric field is generated per unit charge density. This is used when solving the Poisson equation.

By defining all the parameters, it allows for testing the simulation with different values and adjusting them dynamically. Additionally, I will briefly mention how I enforce periodic boundaries for the particles as one of the last necessary steps to produce a minimal simulation. When particles exit from one side, they should re-enter from the opposite end. I implement this in a Niagara Module Script slotted into the 'Particle Update' group. Figure 4.34 shows the simple calculation of the new particle position. I take the world position and local position (position relative to the centre of the emitter) of each particle. I then subtract half the bounding box from the local position and divide by the bounding box to take the range of the particles position to [0..1]. If the resulting position is negative, `floor()` returns -1 and the final computation adds the box size to the position; if the resulting position is positive, `floor()` returns 1 and the final computation subtracts the box size to the position. Any value within [0..1] is safely within the bounding box and remains unchanged.

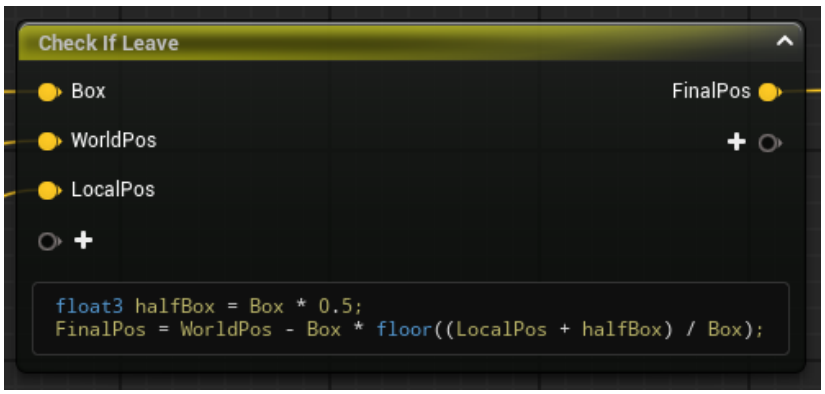
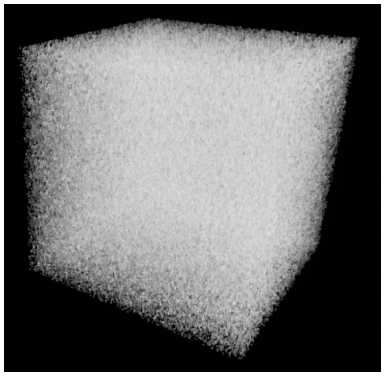


Figure 4.34: Snippet of the Niagara Module Script to enforce Periodic Boundaries

Stage Name	Avg Instances	Avg (us)	Max (us)
ParticleSpawnUpdate	9.9	1,309	2,240
Total	1.2	1,309	2,240

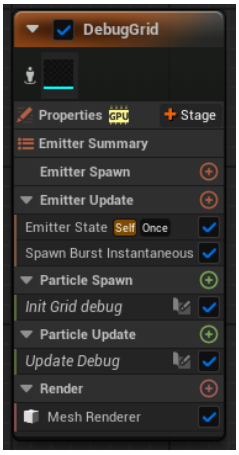
(4.35.a) Performance of Minimal Implementation



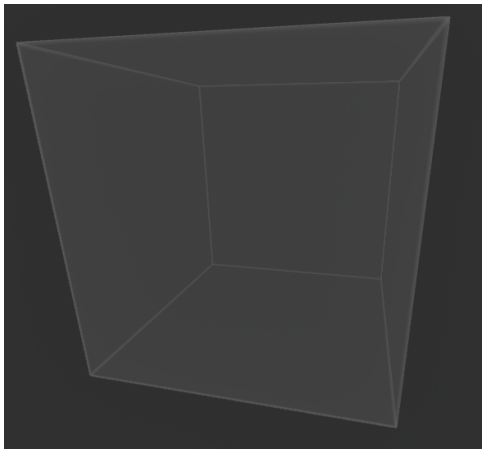
(4.35.b) Visual of Minimal

Figure 4.25.a, shows the current performance for a minimal solution. Minimal here means no physics calculations. It simply spawns particles, give them an initial downwards trajectory and enforces boundary conditions.

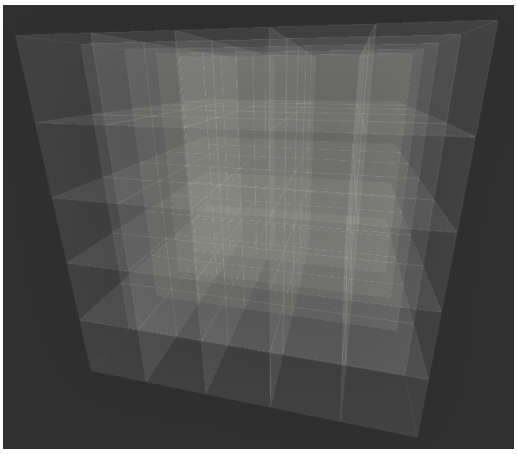
Debug Emitters Previously, I mentioned that I would add multiple emitters to our Niagara System. Here is where I will be doing that in order to help with debugging the grid. Niagara particles are extremely flexible, they can represent electrons in a plasma but they can also be configured to visually show a grid if you attach a see-through mesh and scale it correctly. Figure 4.36.a shows the new Emitter I defined for visualising the grid. I spawn a single particle with a translucent cube mesh and scale it to the bounding box of the grid. The result is shown in figure 4.36.b. Figure 4.36.c shows another emitter that spawns as many particles as there are cells in the grid and renders each grid cell with the same translucent cube mesh. I decided not to use this as the main debug view, as it can get quite obscured especially when increasing the number of cells (the figure only shows 5^3 cells). In the appendix with figure 6.1, it shows the Module Script for setting the debug grid particle properties.



(4.36.a) Emitter



(4.36.b) Debug visual For Entire Grid



(4.36.c) Adjusting Emitter to show Cells

4.4.2 Logging

In order to verify that the simulation is executing as expected, a simple technique would be to attach log messages to different areas of execution. By capturing the order of log messages, I can discern exactly the execution process of the simulation. Important aspects, such as texture resetting and red-black counter alternating will be explicitly logged as they are fundamental to the execution process. To help clarify, I will clearly set-out exactly what will be logged with the following table, including the message and the function we add the log to:

Intention	**Debug Message**	**Function**
NumCells adjusting	Setting Number of Cells to (%d,%d,%d)	**SetNumCells()**
Grid size adjusting	Setting Bounding Box to (%f,%f,%f)	**PerInstanceTickPost..**
Resize/Reset textures	Reset/Resize textures	**PreStage()**
Initialising NDI	Initialising an NDI instance with id: %u	**InitPerInstanceData()**
Red-black counter	Counter: %d	**SetShaderParameters()**
Texture clearing	Clearing all textures	**ResetData()**
Number Density clearing	Clearing Number Density	**PostSimulate()**
Scatter pre-stage	PreStage: Scatter To Grid	**PreStage()**
Charge Density pre-stage	PreStage: Compute Charge Density	**PreStage()**
Solve Potential pre-stage	PreStage: Solve Plasma Potential idx: %u	**PreStage()**
Solve EField pre-stage	PreStage: Solve Electric Field	**PreStage()**
Gather pre-stage	PreStage: Gather to Particle	**PreStage()**

Table 4.2: Tracking our data

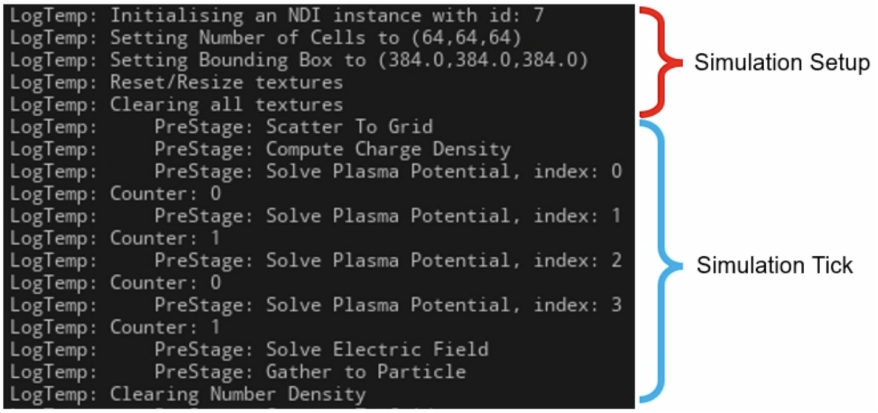


Figure 4.37: Log results from Initialisation and first tick

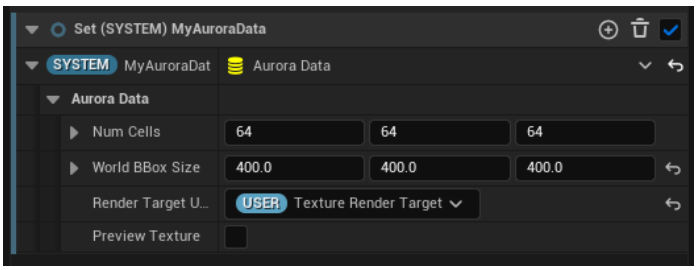
Figure 4.37 shows the result of the log when initialising the Emitter and running its first tick. The simulation setup section correctly sets the number of cells and bounding box from their default values, it then makes sure to update the textures to the new number of cells with the 'Reset/Resize textures' log. For the simulation tick itself, we can see that each stage is successfully being executed in the correct order. With debugging the counter value, we can see that it correctly alternates between 0 and 1, which enables the red/black method to execute as planned. Lastly, clearing the number density texture is being correctly executed at the end of simulation tick, in order to have a cleared texture for the next tick.

4.4.3 Render Target

For further debugging, projecting the different textures onto a 'Render Target' is something that can be achieved. By copying the current data to an external render target, we can visualise, in real-time, the changes in the data when executing the simulation. I will expose an external property to the user, from which they can attach a render target to hold the copied data. Figure 4.38.a shows the new `UPROPERTY`s that the user can set when they initialise the Niagara Data Interface.

```
UPROPERTY(EditAnywhere, Category = "AuroraData")
Changed in 0 Blueprints
FNiagaraUserParameterBinding RenderTargetUserParameter;

WITH_EDITORONLY_DATA
UPROPERTY(EditAnywhere, Category = "AuroraData")
Changed in 0 Blueprints
uint8 bPreviewTexture : 1;
```

(4.38.a) Emitter



(4.38.b) Debug visual For Entire Grid

In order to have the data copied to the render target, I will have to adjust a few parts of the NDI. I will summarise the changes below as there are quite a lot of small tweaks to existing functions.

1. Adjust `Render Thread struct` to add a render target texture reference. This is so that we know where to copy the data to.

2. Adjust `Game Thread struct` to include the render target user parameter binding and a pointer to the render target. I also define a function `UpdateTargetTexture()` here.

3. Create `UpdateTargetTexture()`, this is called upon initialisation to adjust the render target's size, pixel format, clear colour.

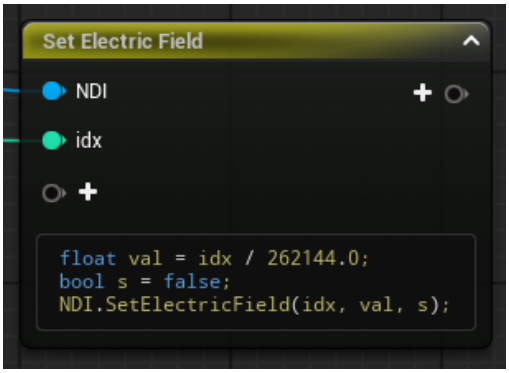
4. Adjust `InitPerInstanceData()` to include initialisation of the texture, includes calling the `UpdateTargetTexture()` function. On the render-thread, I set the set the render target texture reference to the texture's `TextureRHI` from the game-thread.

5. Define `PerInstanceTick()`, this function is called at the start of each tick and checks if I need to update the texture reference on the render-thread. If I do, it sends a render-thread call to update it.

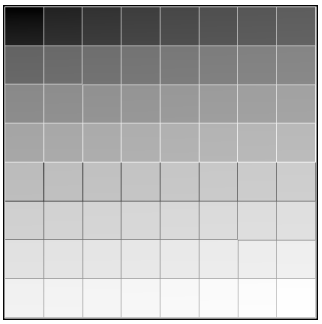
```
if (ProxyData->RenderTargetToCopyTo != nullptr)
{
    ProxyData->ElectricFieldTexture.CopyToTexture(GraphBuilder, ProxyData->RenderTargetToCopyTo, TEXT("RTCopy"));
}
```

Figure 4.39: Copying ElectricField to Render Target code

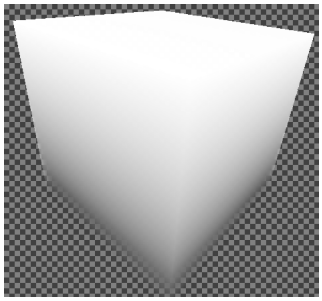
6. Adjust `PostSimulate()`, where the actual copying is done here. Since this function executes at the end of the tick, I know that its values will be fully updated in order to copy safely. Figure 4.39 shows the code for this function. The texture data types conveniently have a built-in copy function.



(4.40.a) Manually setting EField



(4.40.b) Render Target view



(4.40.c) Render Target view in 3D

In order to test the render target is correctly displaying the values at a given texture (Electric Field), I will manually set the values for that texture and view the results. On a side note, when creating all the textures used in this simulation, I defined `Get()` and `Set()` functions for each of them for cases such as this. Therefore, I will utilise the the set function for the electric field here. This value will be the 1D index divided by the total number of cells which results in a value that ranges from [0..1]. I show both the 2D and 3D view of the render target with figures 4.40.b and 4.40.c. The render target visualises the data in the ElectricField in real-time, which is helpful for debugging.

4.4.4 A Stable Solution

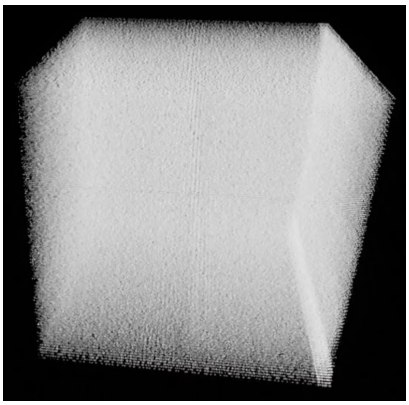
Before I move onto the visualisation, I will first configure a stable state of the simulation. This involves setting the specific parameters used for the solution. This part of the implementation took quite a while to get right, as it involves balancing physical accuracy and efficiency. Small changes to the parameters could completely destabilise the solution and cause completely unphysical results, whilst compute power constraints limited my ability to produce results that were completely noise free. With this 'stable' solution, I prioritised having an efficient, real-time solution that still maintained the physical motion of the aurora. First, I will layout the set of parameter values that I used for this solution, shown in table 4.3.

Parameter	**Type**	**Value used**
NumCells	int32	(64,64,64)
WorldBBoxSize	float	(384, 384, 384)
Particle Count	int32	512,000
Particle Lifetime	float	70 (s)
Initial Velocity	float	([-1..1], [-30..-50], [-1..1]) (cm/s)
Mass	float	9.109e-23 (kg)
Macro-particle weight	float	1.106e7
Precision	float	10.0
Charge	float	-1.602e-19 (C) (**PA**)
IonDensity	float	1.602e-11 (C)
Epsilon	float	8.854e-10 (F/cm) (**PA**)
Delta time	float	0.001 (s)

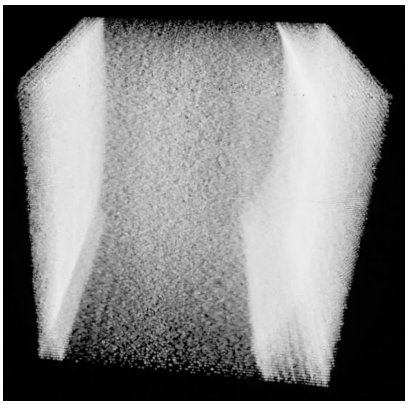
Table 4.3: Simulation Parameters

For the data types, a standard floating point (fp) type was used over the more expensive double precision fp. Although it would increase accuracy, studies (CITE) show that double precision on the GPUs performs much worse compared to single precision fp. Since efficiency is a primary concern, I decided to go with single precision fp, which is common with many GPU-based simulations(CITE). I have attempted to use physically accurate (**PA**) parameter values where possible, such as with the charge and the vacuum permittivity. Other parameters such as the mass and ion density have been adjusted from their physically accurate values to avoid unstable results. The delta time parameter served as quite a bottleneck for my solution, as a value that's too large can easily allow particles to propagate unphysical movement. Therefore, I had to manually increase the particle's mass to avoid uncontrollable speeds.

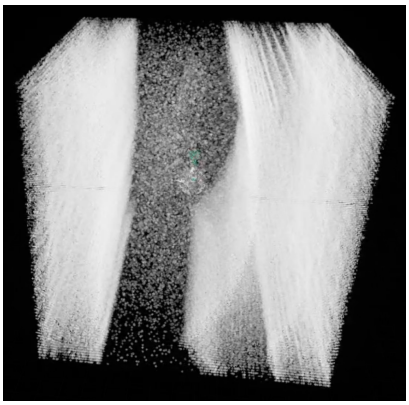
Below, you can find one of the first solutions I produced with the parameters I mentioned before. I found that the motion of particles followed that of the 'curtains' that usually form with auroras. The particles start off in a uniformly distributed grid with equal spacing, so as to avoid unstable motion from the initial proximity of particles if positions were randomised.



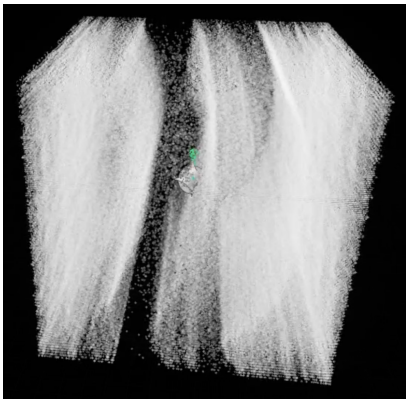
(4.41.a)



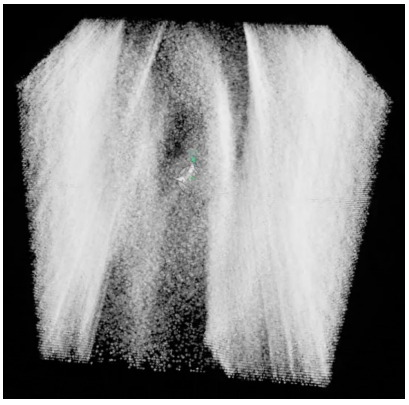
(4.41.b)



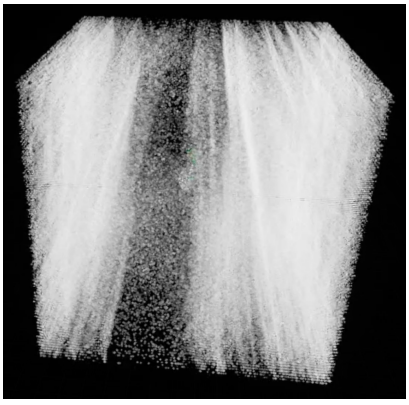
(4.41.c)



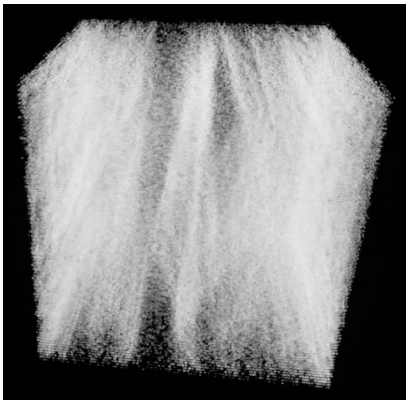
(4.42.a)



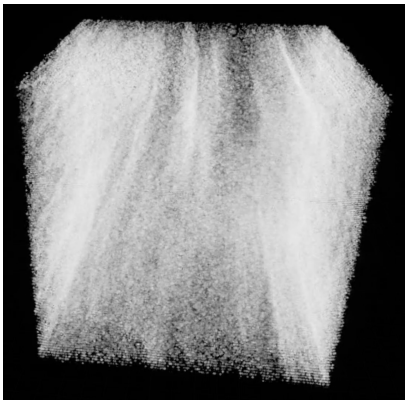
(4.42.b)



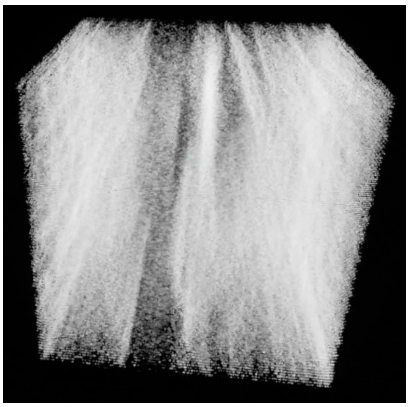
(4.42.c)



(4.43.a)

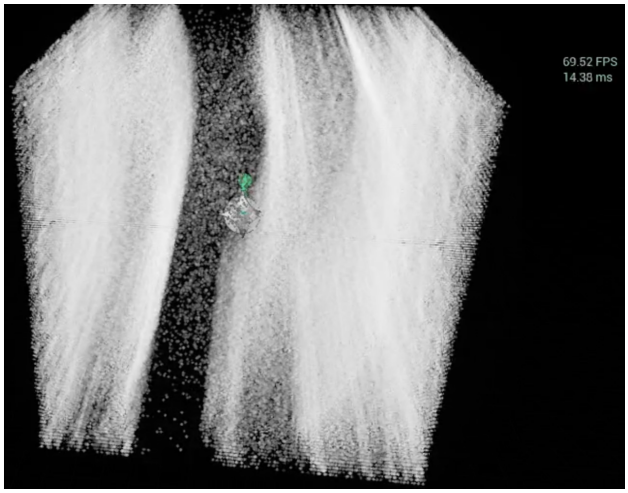


(4.43.b)

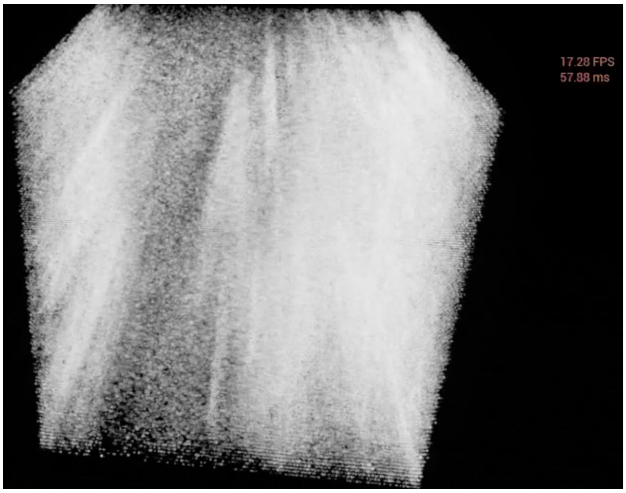


(4.43.c)

Performance: This solution ranged in efficiency. Although the average fps stayed at around 60 fps, there were specific times within the run that strained the performance and reduced the fps. The following figures demonstrate the difference in performance peaks and troughs; where performance didn't usually exceed the 70fps mark whilst the micro-seconds per frame sometimes spiked and caused the fps to drop to lows of 17fps. This indicates that whilst the simulation is performant on the whole, it isn't perfectly stable.



(4.44.a)



(4.44.b)

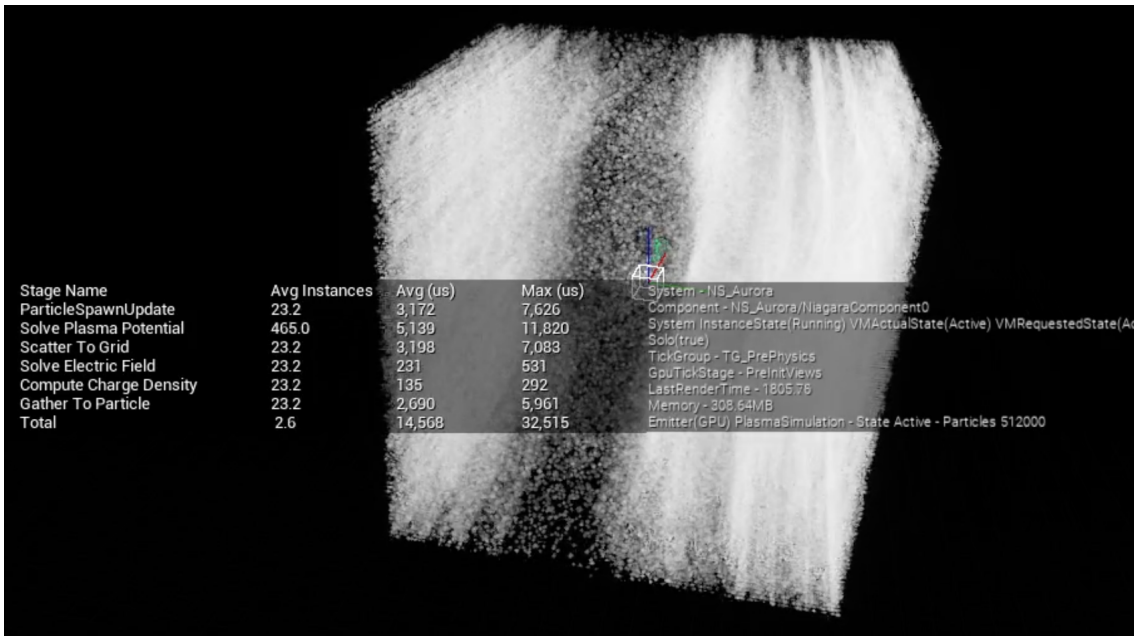


Figure 4.45: Performance breakdown of the different compute shaders calls

Figure 4.45 shows the average and maximum microseconds for the simulations stages. Although I only showed this one screenshot, this set of results also varied, with the 'scatter' stage sometimes overtaking the potential solver with the microseconds per execution. The results are, on the whole, unsurprising; the convergence algorithm iterates over the grid nodes 20 times with this solution, causing the time to surpass the others; the scatter stage comes in next. This stage iterates over all 512,000 particles and requires many atomic operations, causing it to be quite computationally intensive; surprisingly the particle spawn/update stage is next, this is something I will mention later; the gather stage comes after the particle update. This stage also iterates over each particle and requires a host of calculations to gather the electric field for each particle; the compute charge density and solve electric field are last and have noticeably lower execution times. They're both simple and iterate over the least number of elements. Overall, the distribution of performance aligns with initial expectations. The Particle spawn/update stage did confuse me with its time for execution as I didn't include any heavy computations or modules. The next section briefly covers why this was the case.

4.4.5 Profiling

Unreal Engine provides many tools for profiling your solution. The Niagara Debugger, shown previously with figure 4.45, gives a real-time breakdown of how long the stages take. It has other features such as viewing the memory consumption of the overall Niagara System, and visualising the performance as a graph. Unreal Engine offers advanced real-time statistics of the system as well, including the a breakdown of performance of every component grouped by game-thread and render-thread, detailed memory breakdowns and more.

Another tool I used a lot was **Renderdoc** [4]. Renderdoc is an open source debugger for capturing specific frames and detailing pretty much all the information you would want from memory. Unreal Engine provides a built-in plugin for quick capturing of frames during the run. Renderdoc was used extensively to capture the data from the textures and to debug the values to see what could be causing issues. Here is a simple workflow of how I used it to debug that the shaders were working as intended.

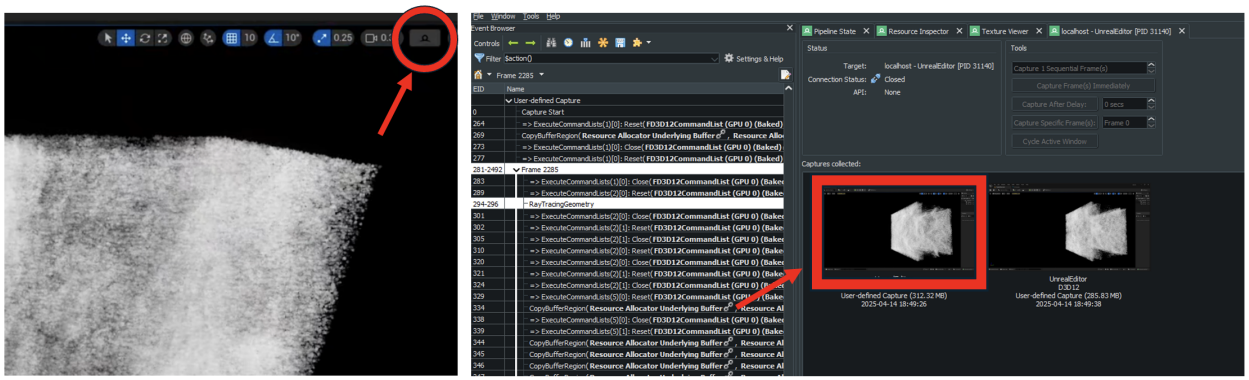
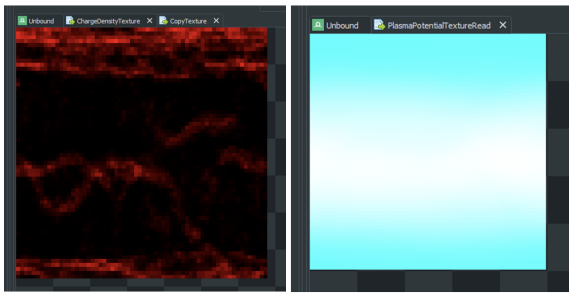
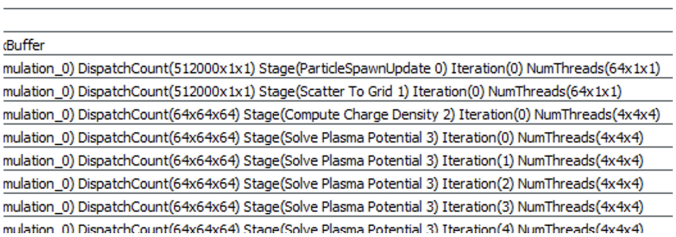


Figure 4.46: Renderdoc workflow

In figure 4.47.a, I have included screenshots of textures for the particle density and the electric potential. It's clear that the density is accurately capturing the areas where particles are. With the potential texture, there is a smooth gradient across the texture which shows that the potential is converging well. If there were large differences between areas of the texture, that would signify that there would need to be more iterations in the convergence solver.



(4.47.a) Texture Debugging



(4.47.b) Renderdoc Pipeline

Renderdoc also has the 'event browser', which details the order of events on the GPU. I traversed through the long list and found where the simulation took place. Here I verify that dispatching over grid data is done in 3 dimensions, with the `NumThreads` being (4x4x4). Particle dispatches will naturally take longer since it will be iterating over 512,000 elements as opposed to 262,144 elements.

In this section I covered the main ways I tested the solutions, with various methods of debugging the data and underlying processes. Other pieces of software I used were the AMD GPU profiler and Unreal Insight. These only confirmed the results shown in this section and therefore I haven't included them here.

4.5 Visualisation

Now that a stable solution has been established, I will move onto the final section of the implementation. Observing the success volumetrics have had with visualising fluids and gas, I will attempt to do the same here with this simulation. At a high-level I will using the ray-marching technique to visualise the density of particles as a volume with added features. Overall, the resources for ray-marching are better with Ryan Bruck's tutorial [18] being a particular example that was very insightful. I will start with a high-level overview of how the visualisation implementation will work:

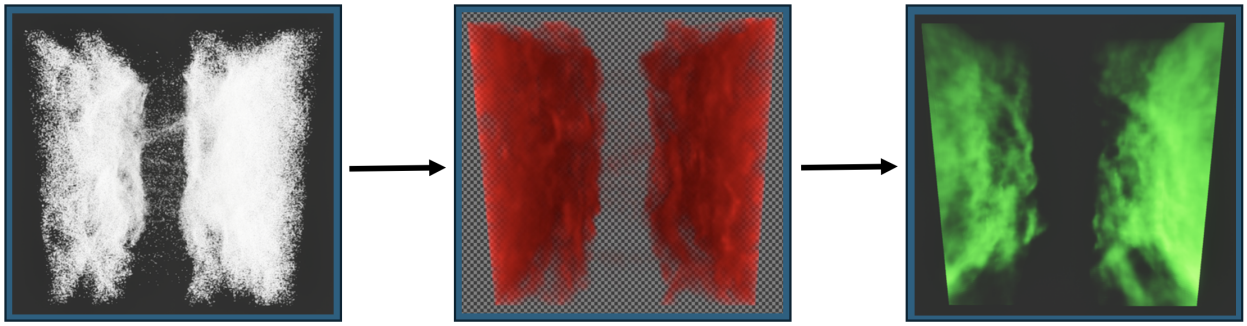


Figure 4.48: A high-level overview of the visualisation implementation

1. The first stage has been mostly completed. I want to get a texture that stores the average number of particles around each grid node. Conveniently I required to do exactly that in the scatter stage. However, in order to convert that macro-particle weight value into the number of particles I will just have to divide the values by the mpw.
2. Currently I don't have anywhere to store the computed values, therefore I will create a new texture to do so. This new texture can be copied to a volume render target which can subsequently fed to the material shader.
3. With the volume texture accessible in the material, I will raymarch through the texture, accumulate values across the given ray and output the emissive colour and opacity to be rendered.

4.5.1 Creating a Particle Density Texture

As mentioned previously, this will be relatively simple. I will run through the steps required to create a texture in the custom NDI quickly as this has been covered already.

1. Add a new `FNiagaraPooledRWTexture` variable in the render-thread struct
2. Add new shader parameters for the UAV/SRV of this new texture. It will have the float4 data type.
3. Adjust `ResizeBuffers()` to include initialising this new texture.
4. Adjust `SetShaderParameters()` to assign the texture value to its shader parameter counterpart.
5. I'll define `Get` and `Set` functions for this texture for extensibility.

```
void {FunctionName}(float Charge, float IonD, float Epsilon, float mpw, float precision, float avgParticles)
{
    int3 GridSize = {NumCells};
    float CellVol = {CellSize}.x * {CellSize}.y * {CellSize}.z;

    const int IndexX = GDispatchThreadId.x;
    const int IndexY = GDispatchThreadId.y;
    const int IndexZ = GDispatchThreadId.z;

    // Copy the computed density value to the new texture - CopyTexture
    float NumDens = (float({NumberDensityRead}.Load(uint4(IndexX, IndexY, IndexZ, 0))) / precision);
    {CopyTextureWrite}[uint3(IndexX, IndexY, IndexZ)].x = NumDens / (mpw * avgParticles);

    float toDensity = NumDens / CellVol;
    float ChargeDensity = ((toDensity * Charge) - IonD) / Epsilon;
    {ChargeDensityWrite}[uint3(IndexX, IndexY, IndexZ)] = ChargeDensity;
}
```

Figure 4.49: Adjusted Compute Charge Density Shader

With the texture defined, the next step is compute the new value from the number density and copy it to this new texture. Instead of creating a new shader to do this step, I will instead adjust the current 'Compute Charge Density' shader function. Since this shader is ran after scattering all the particle's mpw, it will include the updated value. Figure 4.49 shows the new code. I have added two parameters for the macro-particle weight and also for the average maximum particles for a node. I will divide the density by the macro-particle weight to get the average number of particles near it; however, I will also divide it further by the average maximum particles per node in order to normalise the number to the [0..1] range. This is so that when I sample the texture, I can be sure that the weightings will be within the same range. Since this 'maximum' isn't an exact number, I will quickly compute the maximum for a few frames and derive the maximum from there. With figure 4.50, I have shown the general steps for computing this. I capture a few frames of the simulation with Renderdoc, then export the results from the `CopyTexture` to a csv file, this is opened in Excel and I use a simple formula to find the maximum. Overall, the values were around 20 with some reaching 30. Although, I could set the `AvgParticles` parameter to 30, I will keep it at 20 in order to preserve density for grid nodes will fewer average particles. Setting the average too high can neglect most of the grid's densities, since only a few grid nodes will get to 20 30 average particles.

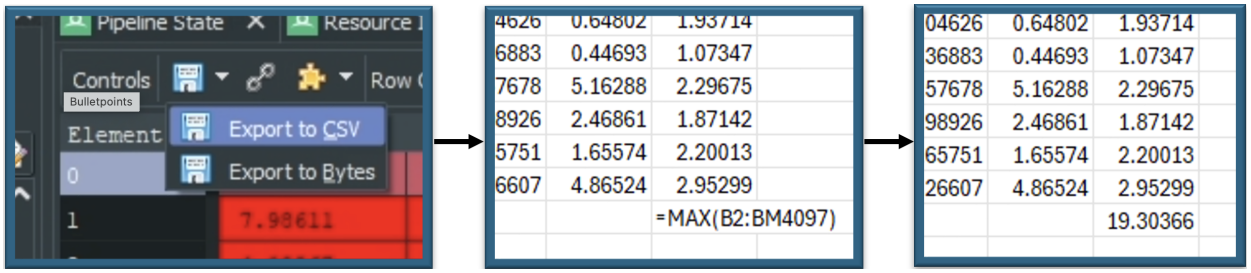


Figure 4.50: Finding the maximum number of particles per node

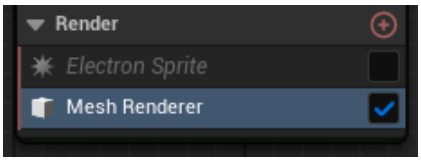
4.5.2 Render Target

Unfortunately, I can't feed the `CopyTexture` into the material shader, therefore I must create a volume render target as a sort of proxy between the custom Niagara Data Interface. I will utilise the render target mechanism I created in section 4.4.3 to pass the reference of the render target into the NDI. I then adjust the `PostStage()` function so that, right after the Compute Charge Density stage, I copy the result to the render target. This is very similar to the code shown in figure 4.39. I just check that the stage just ran was the Compute Charge Density stage, then execute the built-in copy function with the `CopyTexture`.

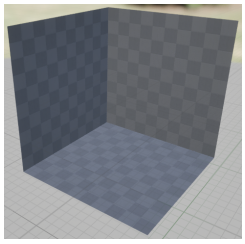
```
if (Context.GetSimStageData().StageMetaData->SimulationStageName == TEXT("Compute Charge Density"))
{
    FNDIAuroraInstanceDataRenderThread* ProxyData = SystemInstancesToProxyData.Find(Context.GetSystemInstanceID());
    if (ProxyData->RenderTargetToCopyTo != nullptr)
    {
        ProxyData->CopyTexture.CopyToTexture(GraphBuilder, ProxyData->RenderTargetToCopyTo, TEXT("NiagaraRenderTarget"));
    }
}
```

Figure 4.51: Adding a copy command to `PostStage()`

Setting up the material: Before moving onto the actual ray-marching, I will run through setting up the material first. A material in Unreal Engine can be attached to a given mesh; when rendering, the material node graph will be executed for each pixel that the mesh covers. Therefore, the material can be thought of as a pixel shader.



(4.52.a) Adding Mesh Renderer



(4.52.b) Inverted Normals Mesh

(4.52.c) Material Settings

The three figures show the setup for creating a new material. For the particles, I have been using the 'Sprite Renderer', so I will add a new renderer for a mesh. With the second figure, I create a static mesh of a cube with inverted normals so that I can easily see into the boundaries of the grid. It will have the same physical size as the grid. Lastly, when you first create material, you have to specify some important settings. The **Material Domain** is defining the purpose of the material. I will set it as a surface, as it is applied to a static mesh. This means that it will be computed as part of the pixel shader, which is what I want. The **Blend Mode** defines how the material interacts with the background. I specify it will be additive as I don't want to lose the background colour, I only want the aurora's volume to add to the colour and opacity. The **Shading Model** defines how the material interacts with light. Although, the aurora does interact with exterior lighting, I keep it as unlit meaning it doesn't interact with other lighting sources. I made this decision on the basis that the Aurora is only visible in the night and that incorporating other lighting factors can be computationally expensive. However, this is something that can be looked into for future work.

4.5.3 RayMarching

For the ray-marching section, I will start by highlighting the work of Ryan Brucks [18], who produced a guide on raymarching for Unreal Engine. Although the method I will use will differ slightly, mainly the sample method and some of the aurora-specific optimisations, the foundations of the solution will be based on his work. Bruck's solution was released quite a long time ago, and therefore some of the methods have been improved upon here. Firstly the sampling method is different. The texture being sampled in his solution is a 'pseudotexture' meaning instead of defining the data in three dimensions, it defines multiple 2D textures to represent a 3D volume. This means his sampling technique has to interpolate between two textures which may not produce results as accurate as a full 3D volume texture. Since his contribution, Unreal Engine has provided the volume render target which I am using, making sampling more accurate. For this implementation, I must first define a few parameters that are fed into the raymarching code. These will be the following:

1. `VolumeTexture`: the volume render target containing the density data. Set as a material parameter.
2. `maxSteps`: the maximum number of steps, set as a material parameter.
3. `cameraVector`: the direction of the ray.
4. `stepSize`: the length between each marching step.
5. `rayStart`: the entry position into the grid.
6. `numSteps`: how many steps I will have to march to get from entry to exit.

I will run through how these parameters are defined using the material graph.

Step Size

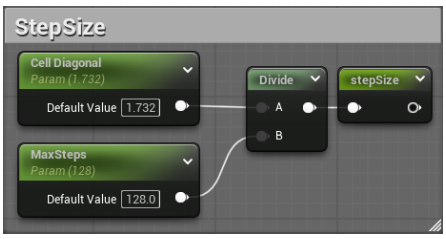


Figure 4.53: Computing the step-size

I compute the step size by dividing the cell diagonal by the max number of steps. The cell diagonal is the corner-to-corner distance of a 1x1x1 cube. By dividing the cell diagonal by the maximum number of steps, it defines the length between each step. The reason this is done is so that the step size ensure adequate coverage of the grid.

Camera Vector (normalised ray direction) - Figure 4.54

The camera vector is essentially the inverse normalised direction to the camera vector. This describes the direction from the camera to the scene. The `invRay` variable will be used later.

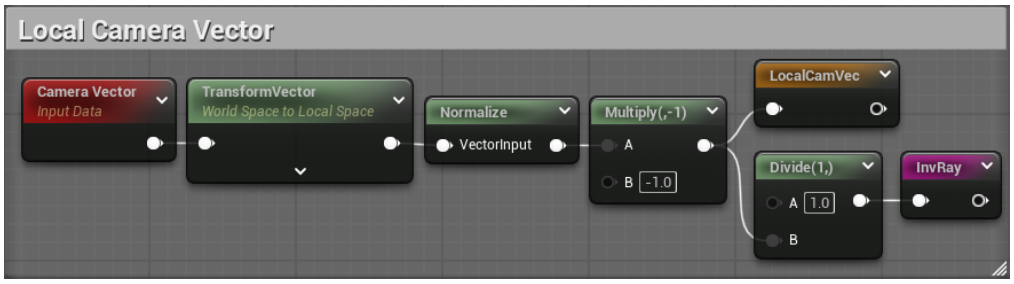


Figure 4.54: Computing the ray direction

Camera Position in Unit Space

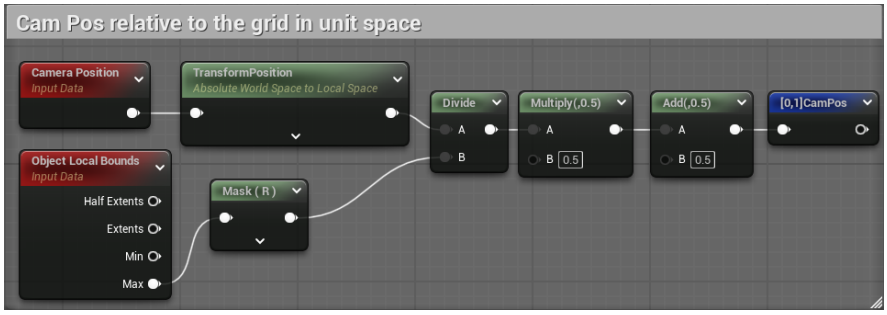


Figure 4.55: Camera Position in Unit Space node graph

In order to compute the unit-space camera position, I divide the camera's local position by the bounds of the grid and adjust the range from [-1..1] to [0..1].

Distances from cam to entry/exit of grid

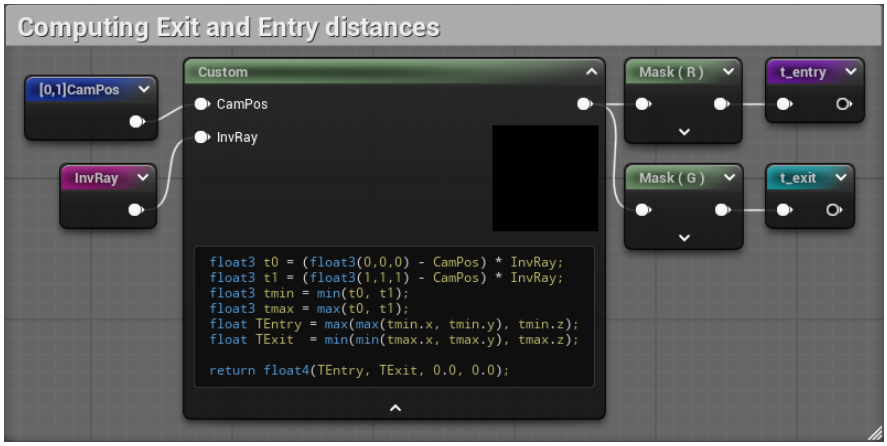


Figure 4.56: Distance from camera to entry/exit of the grid

Using the slab intersection method [68], I compute the distances between the camera and the entry/exit position to the grid. This is a commen intersection technique for an AxisAlignedBoundingBox (AABB).

Number of Steps - Figure 4.57

Using the distances calculated previously, I can figure out the number of steps to march over. I divide the distance from entry to exit by the stepsize and take the floor to return an integer value. This defines the actual number of steps that we march through.

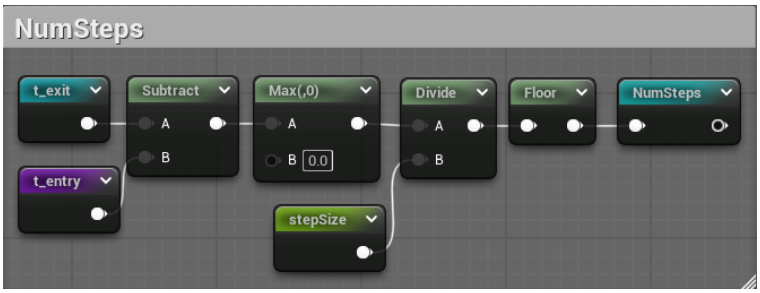


Figure 4.57: Number of steps to march

Ray Start

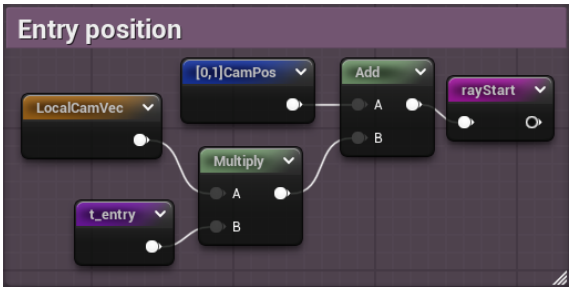


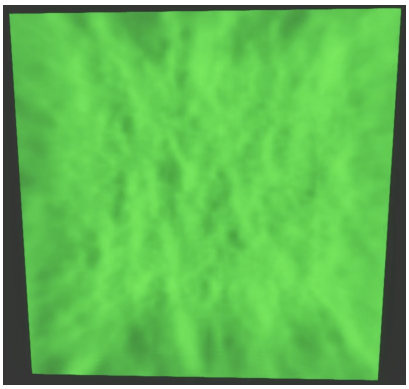
Figure 4.58: The position we enter the volume

To find the position of entry, I add the camera's normalised position with the offset along the camera's direction where the ray hits the volume. This concludes the parameter setup for the raymarcher. Using these defined parameters, I will define a custom HLSL node that defines a basic raymarching solution.

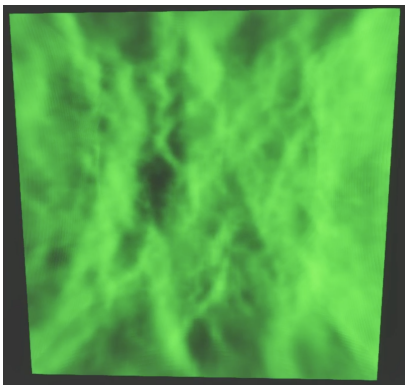
```
cameraVector *= stepSize;
float accum = 0.0;
float transmittance = 1.0f;

for (int i = 0; i < numSteps; i++)
{
    float sampleVal = VolumeTexture.SampleLevel(VolumeTextureSampler, rayStart, 0).r;

    if (sampleVal > 0.001)
    {
        float densityContribution = saturate(sampleVal * stepSize * DensityGain);
        accum += densityContribution;
        transmittance *= (1.0 - densityContribution);
    }
    rayStart += cameraVector;
}

float alpha = saturate(1.0 - transmittance);
return float4(0.0, accum, 0.0, alpha);
```

Figure 4.59: Initial raymarching code

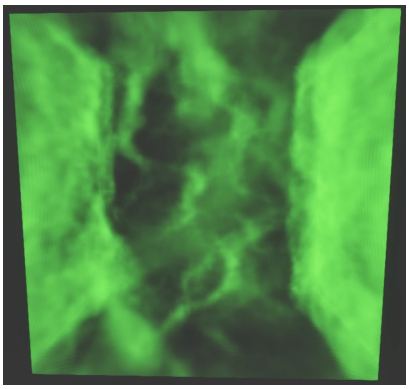
Throughout the raymarching process, I track two variables, `accum` and `transmittance`. The density accumulation will be stored in accum to represent the emissive colour of the pixel, and the transmittance to represent how opaque the pixels value ends up. With each raymarch step, I sample the volume texture, at the `rayStart` position, and obtain the stored density value. Upon checking if the density is worth accumulating, I make sure its within the [0..1] range with `saturate` and accumulate the value. I reduce the transmittance as it moves through the grid which follows the Beer-Lambert law, where light is reduced exponentially based on the density of medium and the distance travelled. The workflow is similar to the raymarcher Brucks proposed [18]. After computing the new accum and transmittance, the current point of the ray is incremented by the stepsize and the entire process repeats until the full number of steps has been reached.



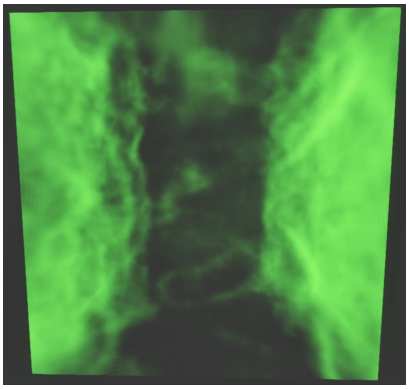
(4.60.a)



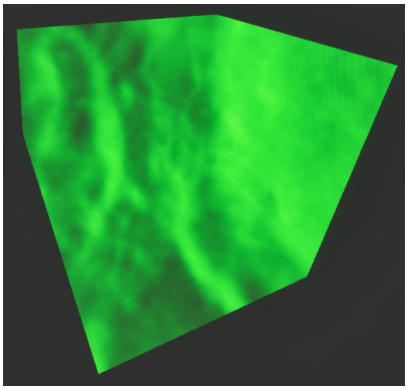
(4.60.b)



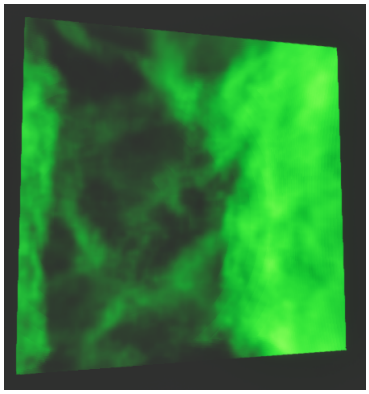
(4.60.c)



(4.61.a)



(4.61.b)



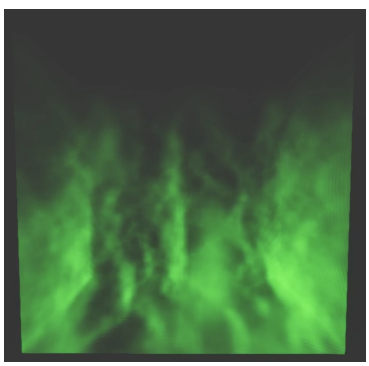
(4.61.c)

I have shown some of the initial results with the current raymarcher. The resulting visualisation is based off a slightly different set of parameters than that of the 'stable solution'. Here, instead of the 'curtain' structure, I attempt to simulate the visual of the 'corona' structure. I also added two extra images that show different angles of the solution. If you zoom into figure 4.61.b, you will notice some faint artifacts in the upper right corner of the grid. This is result of the raymarching process and can appear if the number of steps isn't sufficient.

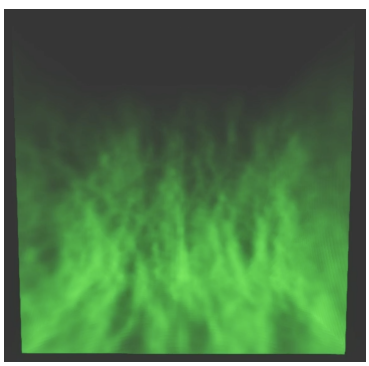
I will add a small improvement to the visualisation, it involves implementing height-based opacity. This is relatively simple change that follows the natural increase in emission the lower the altitude for the aurora [61].

```
float heightfac = saturate(1.0 - rayStart.z);
float densityContribution = saturate(sampleVal * stepSize * DensityGain * heightfac);
```

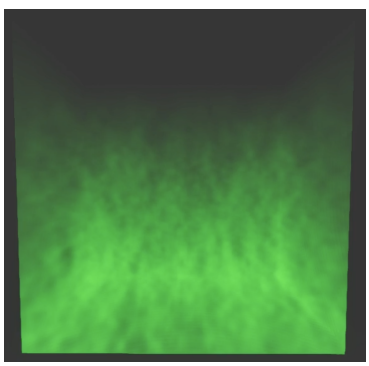
```
float lastheightfac = saturate(1.0 - rayStart.z);
float alpha = saturate((1.0 - transmittance) * lastheightfac);
```



(4.63.a)

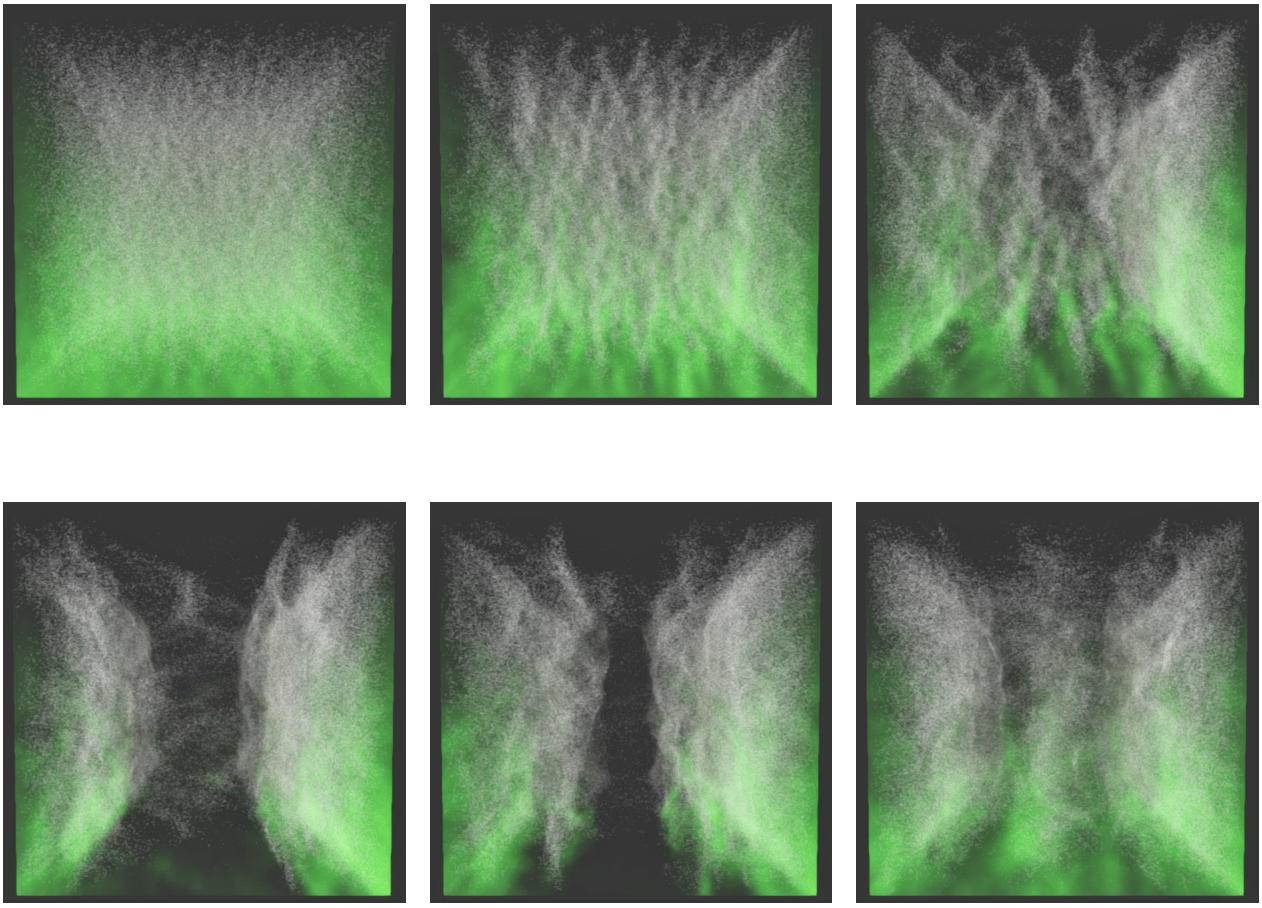


(4.63.b)



(4.63.c)

As the last part to the visualisation, I will show the results when visualising both the particles and volume rendering. This causes the performance to drop quite a lot, down to an average of 20fps. However, you can see how the volume renderer uses the density of the particles to visualise the different parts of the volume. This follows the natural processes of the aurora, where a higher density of charged particles will naturally increase the likelihood of a collision occurring and subsequently releasing the photon we visualise. I reduced the opacity of the particles to help visualise the volumetrics.



4.6 Project Management

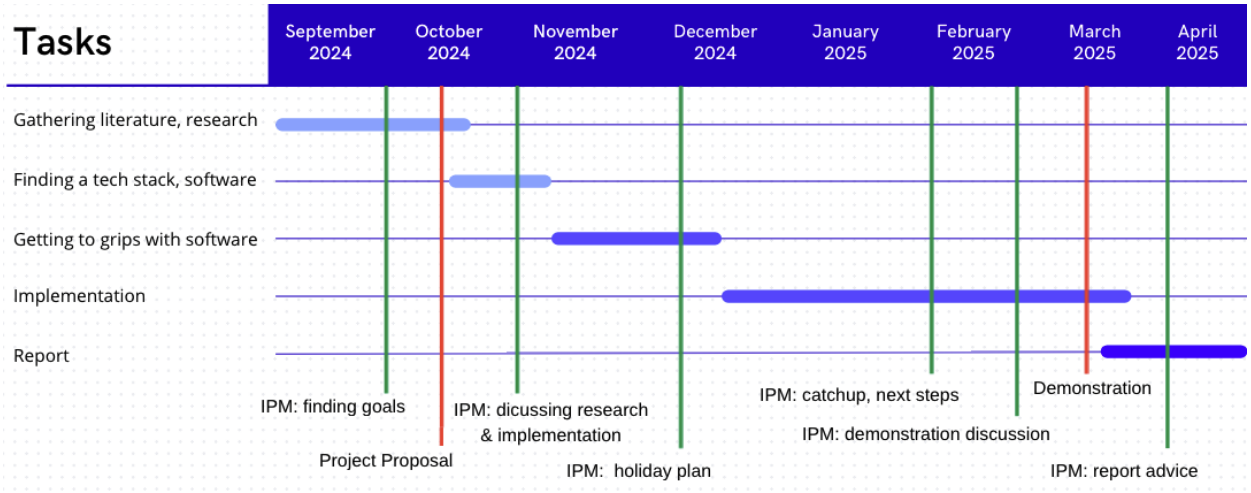


Figure 4.66: Gantt chart for project progress

Figure 4.66 shows how the project was managed over the course of the 2 terms. IPM is for the in-person meetings with Pieter Joubert my supervisor, which mainly consisted of progress updates as well as discussions and advice for the current task. The project had many layers of difficulty, specifically for the implementation that had to be overcome. I had very little understanding of Unreal Engine prior to this project and it therefore required some time to understand its fundamentals and more importantly its pipeline. In terms of what could have been improved with the project management, picking the tech stack alongside gathering the literature would have given me more time for the implementation. I also think some parts of the report could have been consolidated earlier, such as before the implementation, to even out the task of writing the report.

CHAPTER 5

Evaluation

5.1 Performance

Firstly, I will evaluate the performance of the solution. This involves both the computational efficiency of the solution and the effectiveness of the visualisation. Starting with the particle only solution, to re-iterate the results from the stable solution, I got an average of 60fps with hitches that lead to lows of ~20fps. Adding the complexity of raymarching affected performance, reducing the average to around 30fps. This was further reduced when visualising both the particle sprites and the mesh ray-march material to an average of 15fps. To evaluate the performance, I want to compare the solution with past implementations. The only real-time solution that I found was with Lawlor [51], who used a fluid simulation to capture the dynamics of the aurora and rendered it using raytracing to achieve a range of 20-80fps. Given this was produced back in 2010, it is an impressive benchmark. By following his method of using GPU execution to simulate the dynamics of the aurora and visualise the result, I have produced a range similar to that of Lawlor. However, this solution had the added complexity of implementing a kinetic, physically-based simulation for a more physically-based solution. Other plasma based simulations, such as Baranoski [9] and Mills [54] produced similar physically based simulations, however lacked real-time performance. These offline solutions decouples the visual result with the simulation run. This solutions method to enabling real-time rendering, was to making simplifying assumptions to plasmas which we look at further with the 'Realism' section. However, there could be merit to sticking with an offline solution, if it means the simulation could be more physically accurate. Offline rendering is used in fields such as TV/film and architecture visualisation, where the benefit of real-time visualisation is often superseded by the requirement for visual fidelity.

To evaluate the visual effectiveness of the solution, I will make certain evaluations and judgements with respect to observed aurora.

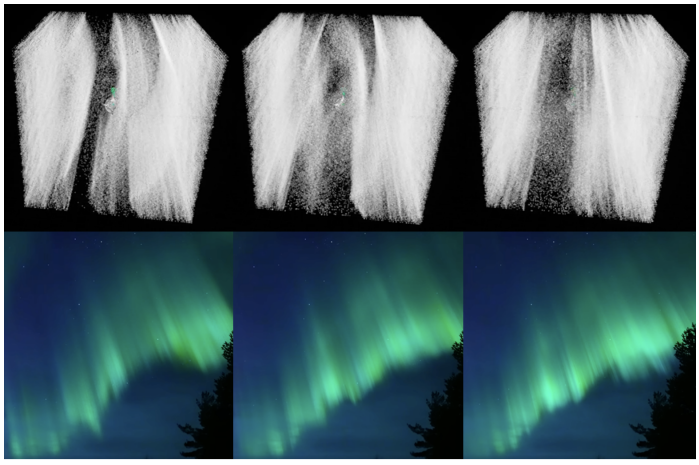


Figure 5.1: A comparison of the dynamic 'curtain' structure seen with discrete aurora

Figure 5.1 shows a comparison between the dynamic movement of commonly seen 'curtain' structures being reproduced. I think the major motions and structure are well captured. I should, however, keep in mind that the particles themselves are not what we refer to as the aurora. However, as a simplification, the general structure of the precipitating electrons coincide with the visual outcome, as a higher density of charged particles increase the chance for collisions with neutral particles, producing the visual aurora.

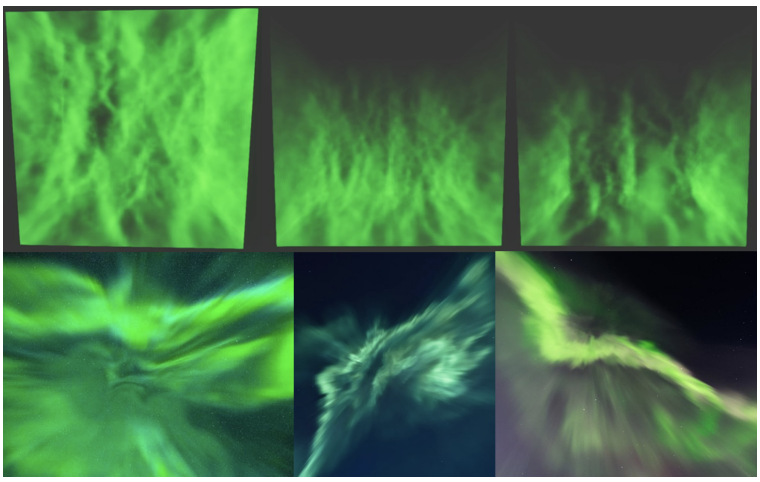


Figure 5.2: A comparison of the visual makeup of the 'corona' structure seen with discrete aurora

Figure 5.2 shows a comparison of the more rare 'corona' auroral event. A problem I encountered with visualisation was dealing with the problem of 'noise' commonly found with PiC simulations. This lead to difficulty with capturing the natural diffuse look of the aurora that appears alongside the more dynamic structures. Overall, I believe I managed to capture the auroral look, however, there is definitely room for improvement with respect to the visualisation. Perhaps, modelling the plasma as a fluid, such as with MHD, would provide greater visual clarity as opposed to a purely kinetic, particle based method.

5.2 Realism

This project attempted the advance the simulation of the aurora with the added constraint of real-time rendering. However, in order to enable this, it required a lot of simplifying assumptions. In order to formalise the assumptions I will devise a list to evaluate how relevant the assumption is towards a more realistic simulation:

Table 5.1: Niagara Simulation Properties

Assumption	**Explanation**
The electrostatic assumption	Plasma in the ionosphere are effected by varying magnetic and electric fields. This simulation simplified the model by assuming a time-invariant magnetic field. This means that we only needed to solve the Poisson equation, as opposed to the full Maxwell's equations. This means that we lose capturing the full electromagnetic wave dynamics or the time-varying magnetic fields. However, this assumption can still be justified, as it is still capable of capturing the electric potential formation and particle drifts. It can also be said that at time scales that this model covers, the magnetic field is largely time invariant [25]. Therefore, this assumption can partially be justified, especially when simulating a basic plasma and it allowed the ability to operate in real-time
Static ion background	This assumption means that the ions formed a uniform charge density background. They still contributed to the electrostatic potential, however they were never explicitly modelled. This can cause a loss in realism as omit phenomena caused by ion motion such as ion acoustic waves or ion-driven sheaths. This assumption is hard to fully justify as electrons and ions go hand in hand when simulating plasma; however, as a simplified simulation, I afforded to neglect the dynamics of ions and any ion-driven motion.

Continued on next page

Table 5.1 – *Continued*

Assumption	**Explanation**
Simulation parameters	I will run through the assumptions I made when devising the parameters. PiC simulations are not meant to model macroscopic phenomena. This makes it an unusual target for simulating the aurora. When devising the scope of the solution, I had to assume a more limited size, in order to maintain some physical properties. This can be justified by adjusting our focus from large-scale aurora to the dynamics of small-scale aurora [46]. This had to be done to enable a solution that didn't become too computationally expensive, however when evaluating the parameters, it can be considered unphysical. PiC simulations are kinetic models and they have to make the assumption that each simulated particle represents a significantly much larger number of real particles. This can affect the accuracy of your solution, as accumulated field values could be under/over-estimating the actual value. This assumption is a limitation to kinetic models as there is computational limit to how many particles you can simulate before the solution becomes unfeasible. With this solution, the number of particles simulated is considered small compared to proper scientific models. Unreal Engine itself limited the number of particles to simulate in order to maintain stable speeds. As a natural consequence to using kinetic models, this assumption can be largely justified. Various assumptions were made with the particles themselves. One example was to increase their mass to artificially slow their movements. This was one of the ways I prevented the simulation from 'exploding'; however, it can question the level of realism of the solution. With computational limits, a large constraint I had to work around was to use a time step that is considered large in the context of scientific applications. A real-time solution, like I was aiming for, required various methods to manage the potentially explosive behaviour of particles. This meant I was forced to change some of the parameters from their physically accurate values. To conclude, many assumptions were made regarding the certain parameters I chose for this solution. Some can be justified with real-world observations, however some were definitely made in order to keep the model stable.

5.3 Suitability

Finally, to evaluate the suitability of this project, I will start with the main aims of the project. Physics simulations for applications other than scientific computing is something I believe is important and incredible relevant. Looking at the advancements of the Computational Fluid Dynamics field and seeing how it has been adopted into a wide range of domains were the grounds for the motivation of this project. When thinking about the trajectory for plasma simulations, I believe this project can have some relevance. Although there are recognisably more challenging aspects to modelling plasmas, there are an infinite amount of future prospects with this domain. Plasmas are not only theorised to make up 99% of the universe but they are still becoming ever more relevant with applications such as fusion [49] [74] [72]. I believe the continued improvement with plasma simulations will inevitably give rise to its use in applications that aren't specifically scientifically research related.

This project touches on one of the domains that plasma simulations can be incorporated with, namely computer graphics. This relationship can be extended with simulating other plasma phenomena such as Transient Luminous Events, which involves sprite lightning as a possible use-case with its red emissions, similar to those seen at high altitudes with the aurora; plasma phenomena for other planetary ionospheres can also be a future prospect here, which even involve auroras as well for planets like Jupiter, Saturn, and Mars.

CHAPTER 6

Conclusion and Future work

This project aimed to produce both a real-time simulation of the aurora based on physically-based methods, and to incorporate visualisation based on modern graphics techniques. To achieve the simulation, I implemented a GPU-based Particle-in-Cell simulation in order to reproduce the natural dynamics of plasma which make up the aurora. The visualisation was based on a volumetric rendering technique, raymarching, which sampled from the simulation's particles density to reproduce the natural emissive effects of visual aurora. The project achieved acceptable real-time performance for the simulation; although the incorporation of the visualisation elements reduced performance but kept it at reasonable levels. The visualisation confirmed commonly seen dynamics seen within aurora, such as 'curtains', whilst, with adjustments made to the simulation, the volumetric renderer managed to capture some of the visual effects seen from auroral 'corona'.

This project serves as an advancement to coupling the simulation of physical processes to its visual phenomena. Whilst it managed to achieve this specification, it did so under many assumptions which limited the overall accuracy of the solution. Therefore, many further iterations can be made such as incorporating more physically-based assumptions, possibly by using the MHD method [44] or a combined MHD-PIC technique [48] [73]. There has been a number of scientific simulations/research for capturing the effects of Alfvén waves on producing the discrete aurora [**?**] [30] [52] [28], which is something that can be further looked at. Simulation and visualisation of a number of plasma instabilities can further validate the cause for auroral dynamics. Finally, this project could benefit from a friendly user interface, possibly for use in a browser, to allow accessible, educational content.

Bibliography

[1] Hlsl reference guide. https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/interlockedadd.

[2] Phet: Free online physics simulations. URL: https://phet.colorado.edu/.

[3] Jos Starn and. A simple fluid solver based on the fft. *Journal of Graphics Tools*, 6(2):43–52, 2001. doi:10.1080/10867651.2001.10487540.

[4] Baldurk. Renderdoc. URL: https://renderdoc.org/.

[5] Herbert James Banda and Joseph Nzabahimana. Effect of integrating physics education technology simulations on students' conceptual understanding in physics: A review of literature. *Phys. Rev. Phys. Educ. Res.*, 17:023108, Dec 2021. URL: https://link.aps.org/doi/10.1103/PhysRevPhysEducRes.17.023108, doi:10.1103/PhysRevPhysEducRes.17.023108.

[6] J Banda, H.J Nzabahimana. he impact of physics education technology (phet) interactive simulation-based learning on motivation and academic achievement among malawian physics students. *J Sci Educ Technol*, 2023. doi:doi.org/10.1007/s10956-022-10010-3.

[7] Gladimir Baranoski and Jon Rokne. Rendering plasma phenomena: Applications and challenges. *Computer Graphics Forum*, 26:743 – 768, 12 2007. doi:10.1111/j.1467-8659.2007.01041.x.

[8] Gladimir Baranoski, Jon Rokne, Peter Shirley, Trond Trondsen, and Rui Bastos. Simulating the aurora. *The Journal of Visualization and Computer Animation*, 14:43 – 59, 02 2003. doi:10.1002/vis.304.

[9] Gladimir V. G. Baranoski, Justin Wan, Jon G. Rokne, and Ian Bell. Simulating the dynamics of auroral phenomena. *ACM Trans. Graph.*, 24(1):37–59, 2005. doi:10.1145/1037957.1037960.

[10] G.V.G. Baranoski, J.G. Rokne, P. Shirley, T. Trondsen, and R. Bastos. Simulating the aurora borealis. In *Proceedings the Eighth Pacific Conference on Computer Graphics and Applications*, pages 2–432, 2000. doi:10.1109/PCCGA.2000.883852.

[11] Matthias Berger and Verina Cristie. Cfd post-processing in unity3d. *Procedia Computer Science*, 51:2913–2922, 2015. International Conference On Computational Science, ICCS 2015. URL: https://www.sciencedirect.com/science/article/pii/S1877050915012843, doi:10.1016/j.procs.2015.05.476.

[12] Charles K. Birdsall and Dieter Fuss. Clouds-in-clouds, clouds-in-cells physics for many-body plasma simulation. *Journal of Computational Physics*, 135(2):141–148, 1997. URL: https://www.sciencedirect.com/science/article/pii/S0021999197957235, doi:10.1006/jcph.1997.5723.

[13] Joseph E. Borovsky and Gian Luca Delzanno. Active experiments in space: The future. *Frontiers in Astronomy and Space Sciences*, Volume 6 - 2019, 2019. URL: https://www.frontiersin.org/journals/astronomy-and-space-sciences/articles/10.3389/fspas.2019.00031, doi:10.3389/fspas.2019.00031.

[14] Robert Bridson and Christopher Batty. Computational physics in film. *Science*, 330(6012):1756–1757, 2010. URL: https://www.science.org/doi/abs/10.1126/science.1198769, doi:10.1126/science.1198769.

[15] L Brieda. *Plasma Simulations by Example*. CRC Press, 2019. doi:10.1201/9780429439780.

[16] N. Brindley, D. Whiter, and I. Gingell. Statistical survey of fine-scale auroral structure at high latitude: Evidence consistent with acceleration by dissipative non-linear inertial alfvén waves. *Journal of Geophysical Research: Space Physics*, 130(4):e2025JA033715, 2025. e2025JA033715 2025JA033715. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2025JA033715, arXiv:https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/2025JA033715, doi:10.1029/2025JA033715.

[17] Erica K. Brockmeier. Lights, camera, physics simulations, 2025. URL: https://www.aps.org/apsnews/2025/03/lights-camera-physics-simulations.

[18] Ryan Brucks. Creating a volumetric ray marcher, 2016. URL: https://shaderbits.com/blog/creating-volumetric-ray-marcher.

[19] D. A. Bryant. *Rocket Studies of Particle Structure Associated with Auroral Arcs*, pages 103–111. American Geophysical Union (AGU), 1981. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/GM025p0103, arXiv:https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/GM025p0103, doi:10.1029/GM025p0103.

[20] J. Büchner. *Space and Astrophysical Plasma Simulation: Methods, Algorithms, and Applications*. Springer International Publishing, 2024. URL: https://books.google.co.uk/books?id=QSQs0AEACAAJ.

[21] Austin C. Niagara data interfaces. URL: https://dev.epicgames.com/community/learning/knowledge-base/jPMm/unreal-engine-niagara-data-interfaces.

[22] C. C. Chaston and K. Seki. Small-scale auroral current sheet structuring. *Journal of Geophysical Research: Space Physics*, 115(A11), 2010. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2010JA015536, doi:10.1029/2010JA015536.

[23] A.B Langdon C.K. Birdsall. *Plasma Physics via Computer Simulation*. CRC Press, 1991. doi:Birdsall,C.K.,\&Langdon,A.B.(1991).PlasmaPhysicsviaComputerSimulation(1sted.).CRCPress.https://doi.org/10.1201/9781315275048.

[24] Christopher A. Colpitts. *Investigations of the Many Distinct Types of Auroras*, chapter 1, pages 1–18. American Geophysical Union (AGU), 2015. doi:10.1002/9781118978719.ch1.

[25] Vincent E. Courtillot and Jean-Louis Le Mouël. Time variations of the earth's magnetic field with a period longer than two months. *Physics of the Earth and Planetary Interiors*, 12(2):237–240, 1976. URL: https://www.sciencedirect.com/science/article/pii/0031920176900522, doi:10.1016/0031-9201(76)90052-2.

[26] H. Dahlgren, N. Ivchenko, J. Sullivan, B. S. Lanchester, G. Marklund, and D. Whiter. Morphology and dynamics of aurora at fine scale: first results from the ask instrument. *Annales Geophysicae*, 26(5):1041–1048, 2008. URL: https://angeo.copernicus.org/articles/26/1041/2008/, doi:10.5194/angeo-26-1041-2008.

[27] H. Dahlgren, B. S. Lanchester, N. Ivchenko, and D. K. Whiter. Electrodynamics and energy characteristics of aurora at high resolution by optical methods. *Journal of Geophysical Research: Space Physics*, 121(6):5966–5974, 2016. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1002/2016JA022446, arXiv:https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1002/2016JA022446, doi:10.1002/2016JA022446.

[28] P A Damiano, J R Johnson, and C. C. Chaston. Ion gyroradius effects on particle trapping in kinetic alfvén waves along auroral field lines. *Journal of Geophysical Research: Space Physics*, 121(11):10,831–10,844, 2016. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1002/2016JA022566, arXiv:https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1002/2016JA022566, doi:10.1002/2016JA022566.

[29] Joey de Vries. learnopengl.com. URL: https://learnopengl.com/.

[30] P. A. Delamere, K. Lynch, M. Lessard, R. Pfaff, M. Larsen, D. L. Hampton, M. Conde, N. P. Barnes, P. A. Damiano, A. Otto, M. Moses, and C. Moser-Gauthier. Alfvén wave generation and electron energization in the kinet-x sounding rocket mission. *Physics of Plasmas*, 31(11):112108, 11 2024. arXiv:https://pubs.aip.org/aip/pop/article-pdf/doi/10.1063/5.0228435/20259970/112108_1_5.0228435.pdf, doi:10.1063/5.0228435.

[31] Y. Ebihara and T. Tanaka. Substorm simulation: Quiet and n-s arcs preceding auroral breakup. *Journal of Geophysical Research: Space Physics*, 121(2):1201–1218, 2016. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1002/2015JA021831, doi:10.1002/2015JA021831.

[32] Y. Ebihara and T. Tanaka. Energy flow exciting field-aligned current at substorm expansion onset. *Journal of Geophysical Research: Space Physics*, 122(12):12,288–12,309, 2017. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1002/2017JA024294, doi:10.1002/2017JA024294.

[33] Y. Ebihara and T. Tanaka. volution of auroral substorm as viewed from mhd simulations: dynamics, energy transfer and energy conversion. *JRev. Mod. Plasma Phys*, 2020. doi:10.1007/s41614-019-0037-x.

[34] Unreal Engine. Coding standard for unreal engine. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/epic-cplusplus-coding-standard-for-unreal-engine.

[35] Unreal Engine. Fluid simulation in unreal engine overview. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/fluid-simulation-in-unreal-engine---overview.

[36] Unreal Engine. Fluid simulation overview. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/fluid-simulation-in-unreal-engine---overview.

[37] Unreal Engine. Niagara editor ui. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/editor-ui-reference-for-niagara-effects-in-unreal-engine.

[38] Unreal Engine. Uniagaradatainterface. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/API/Plugins/Niagara/UNiagaraDataInterface.

[39] B. et al Gombosi T.I. van der Holst. Extended mhd modeling of the steady solar corona and the solar wind. *Living Rev Sol Phys 15, 4*, 2018. doi:10.1007/s41116-018-0014-4.

[40] Rudolf Treumann Götz Paschmann, Stein Haaland. *Auroral Plasma Physics*. Springer Dordrecht, 2003.

[41] Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben van Werkhoven, and Henri E. Bal. Optimization techniques for gpu programming. *ACM Comput. Surv.*, 55(11), March 2023. doi:10.1145/3570638.

[42] Masahiro Hoshino. *Fully Kinetic (Particle-in-Cell) Simulation of Astrophysical Plasmas*, pages 337–357. Springer International Publishing, Cham, 2023. doi:10.1007/978-3-031-11870-8_11.

[43] Yongyu Hu, Yunlong Peng, Zhi Gao, and Fusuo Xu. Application of cfd plug-ins integrated into urban and building design platforms for performance simulations: A literature review. *Frontiers of Architectural Research*, 12(1):148–174, 2023. URL: https://www.sciencedirect.com/science/article/pii/S2095263522000644, doi:10.1016/j.foar.2022.06.005.

[44] Tomokazu Ishikawa. Visual simulation of magnetohydrodynamics: —modeling and visualization for cg rendering—. *Journal of the Visualization Society of Japan*, 36:19–23, 01 2016. doi:10.3154/jvs.36.140_19.

[45] Soo Jeong Jo, James Jones, and Elizabeth Grant. Trends in the application of cfd for architectural design. 05 2018. doi:10.5281/zenodo.11078892.

[46] C.C. Knudsen Kataoka, R. Chaston. Small-scale dynamic aurora. *Space Sci Rev 217*, 2021. doi:10.1007/s11214-021-00796-w.

[47] Shinsuke Kato. Review of airflow and transport analysis in building using cfd and network model. *JAPAN ARCHITECTURAL REVIEW*, 1(3):299–309, 2018. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/2475-8876.12051, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/2475-8876.12051, doi:10.1002/2475-8876.12051.

[48] T. B. Keebler, G. Tóth, Y. Chen, and X. Wang. Simulating extreme space weather with kinetic magnetotail reconnection. *Space Weather*, 23(3):e2024SW004057, 2025. e2024SW004057 2024SW004057. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2024SW004057, arXiv:https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/2024SW004057, doi:10.1029/2024SW004057.

[49] Rachel Kremen. Using artificial intelligence to speed up and improve the most computationally intensive aspects of plasma physics in fusion. URL: https://www.pppl.gov/news/2024/using-artificial-intelligence-speed-and-improve-most-computationally-intensive-aspects.

[50] Aleksander Król, Małgorzata Król, and Wojciech Węgrzyński. A study on airflows induced by vehicle movement in road tunnels by the analysis of bulk data from tunnel sensors. *Tunnelling and Underground Space Technology*, 132:104888, 02 2023. doi:10.1016/j.tust.2022.104888.

[51] Orion Lawlor and Jon Genetti. Interactive volume rendering aurora on the gpu. *Journal of WSCG*, 19:25–32, 01 2011.

[52] R.L Lysak. Kinetic alfvén waves and auroral particle acceleration: a review. *Reviews of Modern Plasma Physics*, 2023. doi:10.1007/s41614-022-00111-2.

[53] D. M. Miles, I. R. Mann, I. P. Pakhotin, J. K. Burchill, A. D. Howarth, D. J. Knudsen, R. L. Lysak, D. D. Wallis, L. L. Cogger, and A. W. Yau. Alfvénic dynamics and fine structuring of discrete auroral arcs: Swarm and e-pop observations. *Geophysical Research Letters*, 45(2):545–555, 2018. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1002/2017GL076051, arXiv:https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1002/2017GL076051, doi:10.1002/2017GL076051.

[54] Kyle Mills. Agent-based simulation of the aurora. 2015. URL: https://github.com/millskyle/AuroraSim/tree/master.

[55] Evgeny V. Mishin. Artificial aurora experiments and application to natural aurora. *Frontiers in Astronomy and Space Sciences*, Volume 6 - 2019, 2019. URL: https://www.frontiersin.org/journals/astronomy-and-space-sciences/articles/10.3389/fspas.2019.00014, doi:10.3389/fspas.2019.00014.

[56] Mark Moldwin. *An Introduction to Space Weather*. Cambridge University Press, 2 edition, 2022.

[57] Suvvari Naidu, Madhavan V M, Sandeep Chinta, R. Manikandan, A. Premkumar, and R. Girimurugan. Analysis of aerodynamic characteristics of car diffuser for dissimilar diffuser angles on sedan's using cfd. *Materials Today: Proceedings*, 92, 05 2023. doi:10.1016/j.matpr.2023.04.379.

[58] C. et al Nishikawa K. Duţan, I. Köhn. Pic methods in astrophysics: simulations of relativistic jets and kinetic physics in astrophysical systems. *Living Rev Comput Astrophys*, 2021. doi:10.1007/s41115-021-00012-0.

[59] T Obidi. *Theory and Applications of Aerodynamics for Ground Vehicles*. SAE International, 03 2014. doi:10.4271/r-392.

[60] Overvoorde. vulkan-tutorial.com. URL: https://vulkan-tutorial.com/Introduction.

[61] Götz Paschmann, Stein Haaland, and Rudolf Treumann. *Remote Sensing of Auroral Arcs*, pages 21–40. Springer Netherlands, Dordrecht, 2003. doi:10.1007/978-94-007-1086-3_2.

[62] M. Pohl, M. Hoshino, and J. Niemiec. Pic simulation methods for cosmic radiation and plasma instabilities. *Progress in Particle and Nuclear Physics*, 111:103751, 2020. URL: https://www.sciencedirect.com/science/article/pii/S0146641019300869, doi:10.1016/j.ppnp.2019.103751.

[63] Alessio Regalbuto. How we wrote a gpu-based guassian splats viewer in unreal with niagara, 2024. URL: https://www.magnopus.com/blog/how-we-wrote-a-gpu-based-gaussian-splats-viewer-in-unreal-with-niagara.

[64] T. Sakaki, T.-H. Watanabe, and S. Maeyama. Convective growth of auroral arcs through the feedback instability in a dipole geometry. *Journal of Geophysical Research: Space Physics*, 129(12):e2023JA032407, 2024. e2023JA032407 2023JA032407. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2023JA032407, arXiv:https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/2023JA032407, doi:10.1029/2023JA032407.

[65] Andrew Schneider. Andrew schneider publications. URL: https://www.schneidervfx.com/.

[66] Andrew Schneider. Nubis: Methods (and madness) to model and render immersive real-time voxel-based clouds, 2023. URL: https://advances.realtimerendering.com/s2023/index.html#Nubis3.

[67] H. Shimazu, K. Kitamura, T. Tanaka, S. Fujita, M.S. Nakamura, and T. Obara. Real-time global mhd simulation of the solar wind interaction with the earth's magnetosphere. *Advances in Space Research*, 42(9):1504–1509, 2008. URL: https://www.sciencedirect.com/science/article/pii/S027311770700779X, doi:10.1016/j.asr.2007.07.014.

[68] Peter Shirley, Ingo Wald, and Adam Marrs. *Ray Axis-Aligned Bounding Box Intersection*, pages 37–39. Apress, 2021. doi:10.1007/978-1-4842-7185-8_2.

[69] Yousuf Soliman, Marcel Padilla, Oliver Gross, Felix Knöppel, Ulrich Pinkall, and Peter Schröder. Going with the flow. *ACM Trans. Graph.*, 43(4), July 2024. doi:10.1145/3658164.

[70] Jos Stam. Stable fluids. SIGGRAPH '99, page 121–128, USA, 1999. ACM Press/Addison-Wesley Publishing Co. doi:10.1145/311535.311548.

[71] Svetlana Valger and N.N. Fedorova. Cfd methods in architecture and city planning. *Journal of Physics: Conference Series*, 1425:012124, 12 2019. doi:10.1088/1742-6596/1425/1/012124.

[72] Jean-Luc Vay, Justin Ray Angus, Olga Shapoval, Rémi Lehe, David Grote, and Axel Huebl. Energy-preserving coupling of explicit particle-in-cell with monte carlo collisions. *Phys. Rev. E*, 111:025306, Feb 2025. URL: https://link.aps.org/doi/10.1103/PhysRevE.111.025306, doi:10.1103/PhysRevE.111.025306.

[73] Xiantong Wang, Yuxi Chen, and Gábor Tóth. Global magnetohydrodynamic magnetosphere simulation with an adaptively embedded particle-in-cell model. *Journal of Geophysical Research: Space Physics*, 127(8):e2021JA030091, 2022. e2021JA030091 2021JA030091. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2021JA030091, arXiv:https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/2021JA030091, doi:10.1029/2021JA030091.

[74] Yifei Yang. The role of quantum computing in advancing plasma physics simulations for fusion energy and high-energy. *Frontiers in Physics*, Volume 13 - 2025, 2025. URL: https://www.frontiersin.org/journals/physics/articles/10.3389/fphy.2025.1551209, doi:10.3389/fphy.2025.1551209.

[75] B. & Mitchell K Zadick, J. Kenwright. Integrating real-time fluid simulation with a voxel engine. *Comput Game J*, 2016. doi:10.1007/s40869-016-0020-5.

[76] Felipe Miguel Álvarez Siordia, César Merino-Soto, Samuel Antonio Rosas-Meléndez, Martín Pérez-Díaz, and Guillermo M. Chans. Simulators as an innovative strategy in the teaching of physics in higher education. *Education Sciences*, 15(2), 2025. URL: https://www.mdpi.com/2227-7102/15/2/131, doi:10.3390/educsci15020131.

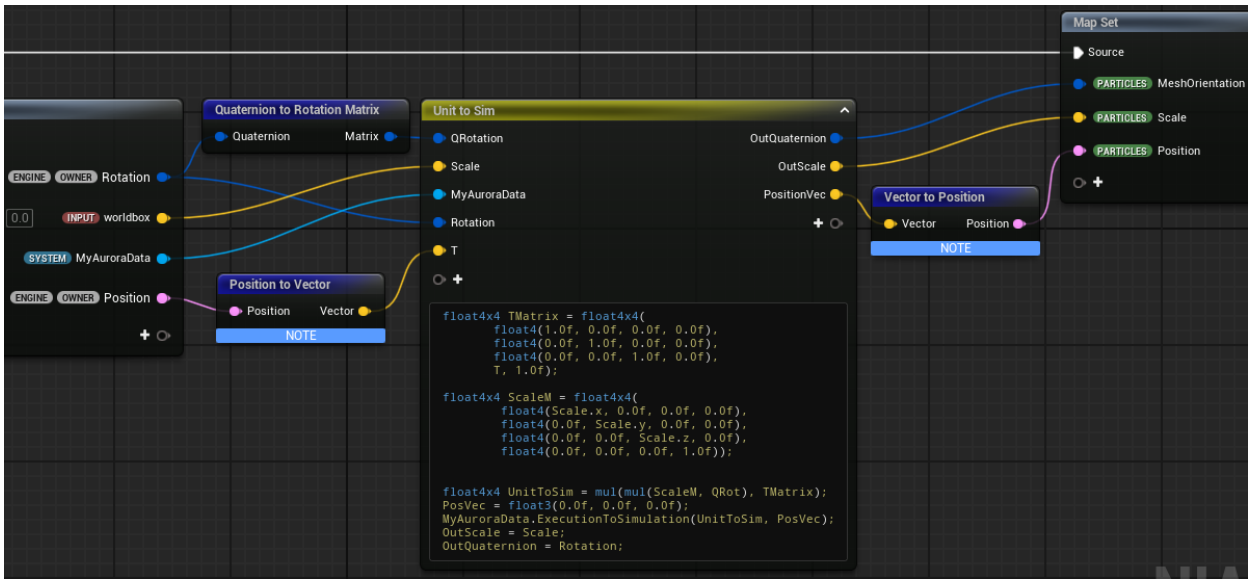


Figure 6.1: Grid Visual for Debugging Module Script