Joshua P. Bradley

Lorentz and Dipoles and Charge, Oh My! Implementing a Software Library for Simulating the Lorentz Force in the Unity Game Engine

MSci Hons Computer Science (with Industrial Experience)

10/06/22

I certify that the material contained in this dissertation is my own work and does not contain unreferenced or unacknowledged material. I also warrant that the above statement applies to the implementation of the project and all associated documentation. Regarding the electronically submitted work, I consent to this being stored electronically and copied for assessment purposes, including the School's use of plagiarism detection systems in order to check the integrity of assessed work.

I agree to my dissertation being placed in the public domain, with my name explicitly included as the author of the work.

Name: Joshua P. Bradley

Date: 10/06/2022

Abstract

Game engines are software frameworks that provide a platform for interactive application development. While they offer a plethora of built-in tools and libraries to facilitate the application development process, it is often necessary to source external tools to handle increasing project complexity. This paper details the motivation, design, and implementation of one such external tool: a novel software library named the Electromagnetism Mechanics Library (EML), built for the Unity game engine. The EML simulates the Lorentz force acting upon particles emitted by Unity's built-in Particle System component, as they travel through a magnetic dipole's field. The solution is capable of producing variable simulations based on arbitrary magnetic dipole strength; particle charges and particle velocities. Initial testing has shown that the system produces simulations that emulate the Lorentz force and is capable of safely handling invalid internal states. Upon being applied to a non-contrived Unity project requiring the EML, named the Particle Pinball Machine (PPM), it has been shown that the EML was capable of achieving the desired effect in the project. Despite the positive results in testing, expert opinion on the simulation correctness is required to verify how successfully it emulates the Lorentz force.

Contents

Glossary

Term	Definition
Air shower	A shower of secondary particles produced when high-energy particles collide with atmospheric nuclei (Engel, Heck and Pierog, 2011).
Application Programming Interface (API)	A well-defined interface in software that provides services to client software (Reddy, 2011).
Bow shock	The curved shock wave formed by the collision of solar wind with an astronomical body's magnetosphere (Kotova et al., 2021).
Component	Functional pieces of every GameObject that are composed of editable properties. Scripts can be attached to GameObjects as components (Unity, 2021c).
Electromagnetism Mechanics Library (EML)	The name given to the sole deliverable of this project - the Unity game engine library that simulates charged particle interaction with magnetic dipoles.
Game engine	A software framework that provides a collection of tools that facilitate the creation of interactive applications (Toftedahl and Engström, 2019).
GameObject	The fundamental objects in Unity that represent characters, props and scenery. They act as containers for components (Unity, 2021a).
Geomagnetic field	The magnetic field that extends from the Earth's interior into space (Mandea and Chambodut, 2020).
Gizmo	Graphics associated with GameObjects that are used for visual debugging or setup aids (Unity, 2021b).
Graphical User Interface (GUI)	A form of UI system composed of interactive visual components for computer software (Oulasvirta et al., 2020).
Magnetic dipole	The limit of a pair of magnetic poles as the size of the source is reduced to zero, while keeping the magnetic moment constant (Corbò and Testa, 2009).

Magnetic dipole moment	A vector that quantifies the magnetic strength and orientation of an object that produces a magnetic field. The vector points from the South pole to the North pole (Nowicki and Szewczyk, 2019).
Magnetic field strength (**H**)	A measure of the strength of a magnetic field at a given point independent of material's own magnetic response: measured in electric current per unit length (A/m) (Tanel, 2008).
Magnetic flux density (**B**)	A measure of the strength of a magnetic field at a given point dependent on the particular material's magnetic response: measured in Teslas (T) (Tanel, 2008).
Magnetosphere	The region of space in which charged particles are meaningfully influenced by an astronomical body's magnetic field (Borovsky and Valdivia, 2018).
Point charge	A point charge is an idealisation of charged particles where the particle is considered dimensionless, and therefore exists in a single point in space (Pshchelko and Vodkailo. 2020).
Polar cusps	Funnel-shaped gaps in the geomagnetic field that direct towards the geomagnetic poles (Pitout and Bogdanova, 2021).
Quantized	A characteristic of certain physical properties, where values can assume only certain discrete magnitudes (Lyre, 1996).
Script	A program (sequence of instructions) that is executed by another program rather than directly by the computer processor.
Solar Energetic Particle (SEP)	High-energy particles emitted from the Sun; typically protons and electrons (Malandraki and Crosby, 2018).
Solar wind	Plasma composed of electrically charged subatomic particles and atomic nuclei that is continuously emitted from the Sun (Suzuki, 2012).
Transform	The component that holds a GameObject's *scale*, *rotation* and *position* properties.
User Interface (UI)	The means by which a human and a computer system interact (Sridevi, 2014).

Vacuum permeability (μ_0)	The measure of magnetisation that a vacuum obtains in response to an applied magnetic field (Goldfarb, 2017).
Vector3	A built-in Unity data structure typically used for storing 3-component vector values. It holds three real numbers, stored in *x*, *y* and *z* properties (Unity, 2021d).
Vector4	A built-in Unity data structure typically used for storing 4-component vector values. It holds four real numbers, stored in *x*, *y*, *z* and *w* properties (Unity, 2021e).
Vector field	A subset of space in which each point has an associated vector (Zhang et al., 2006).

Chapter 1. Introduction

This paper details the motivation, design, implementation and critical evaluation of a novel software library that extends the built-in capabilities of the Unity **game engine** (Unity Technologies, 2005), which is referred to throughout this paper as the Electromagnetism Mechanics Library (**EML**). The EML simulates the motion of charged particles as they travel through a non-uniform magnetic field induced by a **magnetic dipole**.

Game engines provide built-in tools to facilitate the creation of interactive applications, and in addition they generally provide a platform for the application developer to import their own assets. These assets allow the game engine's functionality to be extended beyond its base capabilities to achieve highly customisable behaviour that would otherwise be infeasible.

Unity provides numerous built-in tools and systems to simulate motion, which notably includes the simulation of the fundamental interaction of gravity. However, a similar tool to simulate the Lorentz force is not built in: thus providing the opportunity for work to achieve this.

The EML may be particularly useful in the niche development contexts of physics-based games, and educational aids depicting how charged particles travel in the presence of a magnetic field. In the latter context, it could enable an engaging and active learning experience to be developed (in the form of an interactive demonstration) in contrast to a more passive learning experience (e.g. viewing an educational video or reading from a textbook). This may be a more effective method of learning, that would allow the complexity of electromagnetism to be understood in a more effective and efficient manner.

This project has two aims:

1. To develop a fully documented software library, compatible with the Unity game engine, that is capable of simulating the motion objects of arbitrary charge within a magnetic field of arbitrary strength and direction.
2. To test and verify the correctness of the solution by implementing it within a Unity project that requires the simulation of electromagnetic interaction.

In this paper, the project context and motivation are discussed in depth. Following this, the required background of the electromagnetism concepts used in the system is provided. Next, the solution design is described and motivated, before the implementation is detailed, with key algorithms used in the solution discussed. The solution is then thoroughly tested in a multi-stage process, partly comprised of using the EML in a system that requires the solution, and determining whether the EML performs as expected. Finally, the solution and the development process is critically evaluated.

Chapter 2. Background

In this chapter, the motivation of the work is described, which provides essential context for chapter 6. Particularly, it provides context for the Unity project that the work will be tested with - an uncontrived use-case of the EML. Following this, essential material on the principles that govern the mechanics of electromagnetism is described, which go on to significantly inform the system design, described in chapter 3. To conclude the chapter, a brief description of the solar-Earth electromagnetic interaction is given, which forms a key part of the evaluation process in chapter 7.

2.1. Motivation

2.1.1. Project Background

The work detailed in this paper, whilst designed as a general solution and viewable as an isolated work, is borne from a project assigned from the Lancaster-based company, Hybrid Instruments Ltd. Their work includes developing equipment for space-weather monitoring systems, and most recently working on the Ground Level Enhancement Event Monitoring (GLEEM) project, which involves developing improved neutron detection systems for the UK Met Office. Following on from the work on GLEEM, Hybrid Instruments Ltd wanted to develop an application that would provide context to the company's contribution to the project as a method of public engagement.

From this, they sought an individual with software engineering experience to develop an interactive Particle Pinball Machine (PPM) application. The aim of which is to demonstrate **solar energetic particle** (SEP) emission and the **air shower** phenomena - the origin of the neutrons detected at ground level.

Interactive applications, particularly video games, are typically developed using game engines. This is because they provide a suite of tools that ease the development process for the application developers: decreasing cost and development time. As a result, it was appropriate to select a game engine for the development of the PPM. From the project requirements, the Unity game engine was selected.

The portion of the PPM that is of relevance to this work is the opening scene, as shown in Figure 2.1. It comprises an image of the Sun positioned on the left edge of a landscape display which emits small white circles radially outwards, representing solar wind. Towards the right of this display, the Earth is positioned. The motivation for this work is to simulate the real-world phenomena of solar wind deflecting around the Earth due to the **geomagnetic field**, within this scene. The PPM forms an important role in the EML testing and evaluation processes, as it is this Unity project that will have the solution applied to it.

Figure 2.1: The opening scene of the PPM application.

2.1.2. Unity Built-In Solution Limitations

Figure 2.1 shows the PPM solution in its current state, without any deflection on the particles. The expected result for the deflection of solar wind in the presence of the geomagnetic field can be observed in Figure 2.2: it shows that incoming particles deflect around the Earth in a curved path.

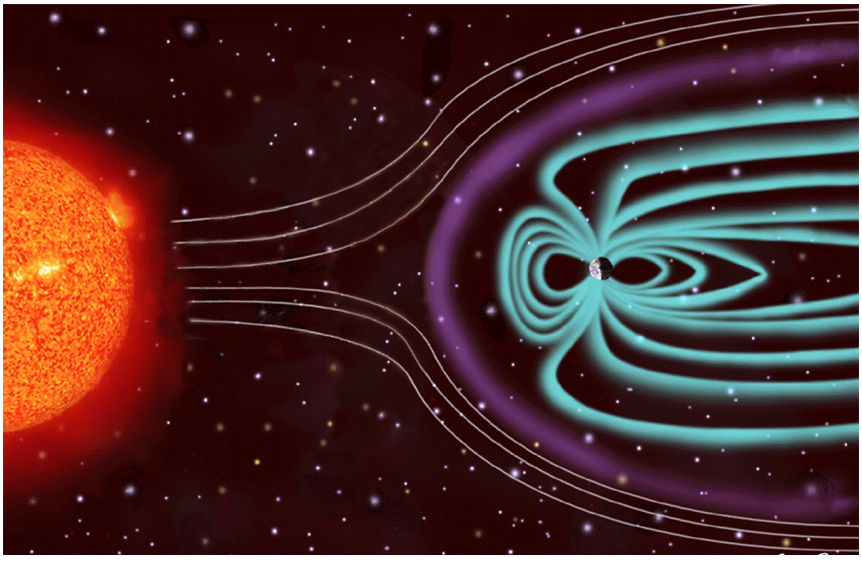
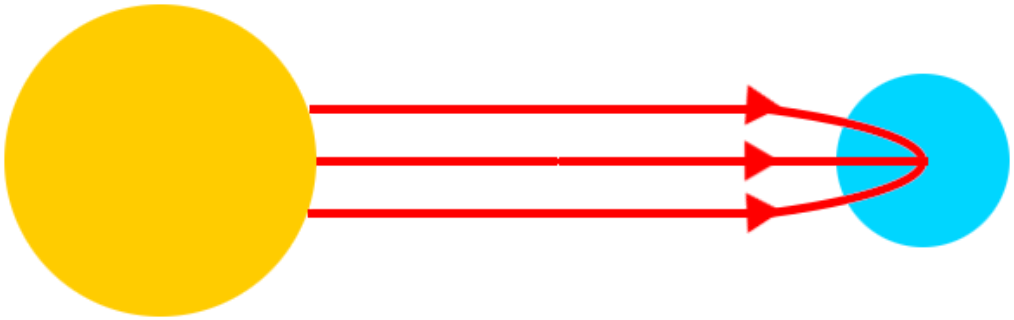
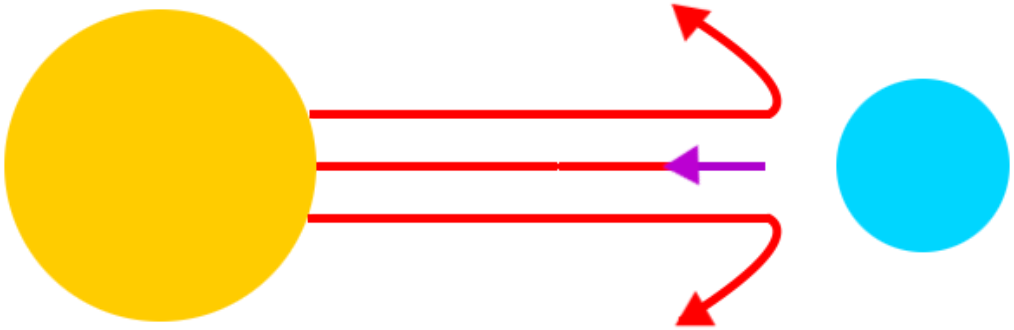


Figure 2.2: An artistic depiction of solar wind-magnetosphere interaction. The beige paths depict the expected deflection paths for solar particles emitted towards Earth on a 2D plane.

There are four approaches using Unity's built-in technologies that could be taken to simulate this behaviour, this includes: simulating particle and Earth collisions; simulating acceleration due to gravity between the Earth and particles (including an additional effect where the attraction is reversed - exhibiting repulsion), and finally simply leaving the deflection unsimulated. The approaches are summarised in Table 2.1.

Deflection Method	**Reference Figure (Above) and Description (Below)**
Collision Simulation	
	Particles bounce away from the surface of the stationary Earth upon collision.
Gravity Simulation (Attraction)	
	Particles accelerating in a path tending towards the centre of the Earth.
Gravity Simulation (Repulsion)	

	Particles are repelled from the Earth, causing particles to be accelerated in a path tending away from the centre of the Earth.
No Simulation	
	Particles do not interact with the Earth; they pass through with their motion unaffected.

Table 2.1: A comparison of the approaches that could be taken to represent the particle-Earth interactions in the PPM.

None of the four solutions described in Table 2.1 achieved an effect that closely resembled the desired outcome shown in Figure 2.2, and so a new Unity system had to be sourced.

2.2. Electromagnetism

The electromagnetic force, or Lorentz force, is one of the four fundamental interactions in nature, together with gravitation; the weak nuclear force, and the strong nuclear force. The field of electromagnetism describes the interactions that take place between electric and magnetic fields.

Electromagnetism is a wide and complex field of study, and so it is necessary to restrict the discussion to that which is relevant to the problem domain. Of particular interest are two key areas: the mechanics that determine how charged particles move in the presence of a magnetic field, and understanding the nature of magnetic fields - particularly how to compute the values relating to the magnetic field that affect charged particle motion.

In this section, the two highlighted domains of electromagnetism are discussed. In addition, electromagnetism in the solar system is briefly discussed to provide context for the desired behaviour in the PPM.

2.2.1. Mechanics of Charged Particles in Magnetic Field – The Lorentz Force

To determine the electromagnetic force vector that is applied to a **point charge**, the Lorentz force equation (Lorentz, 1895) is applied. This equation is as follows:

$$\vec{F} = q(\vec{E} + \vec{v} \times \vec{B}) \tag{2.1}$$

where, $\vec{F}$ is the force applied to the particle; q is the charge of the particle; $\vec{E}$ is the electric field vector; $\vec{v}$ is the velocity vector of the particle and $\vec{B}$ is the **magnetic flux density** vector (see 2.2.2). Equation (2.1) can be rearranged to separate the electric field interaction and magnetic field interaction into two distinct terms. This more practical format corresponds to the following equation:

$$\vec{F} = q\vec{E} + q\vec{v} \times \vec{B}. \tag{2.2}$$

The left-hand term of Equation 2.2, $q\vec{E}$, corresponds to electric field interaction, while the right-hand term of Equation 2.2, $q\vec{v} \times \vec{B}$, relates to magnetic field interaction. With this rearrangement, the electric field term can be removed to only include the magnetic field term. This results in the simplified equation:

$$\vec{F}_{magnetic} = q\vec{v} \times \vec{B}. \tag{2.3}$$

In the remainder of this paper, 'Lorentz force' will refer to (Eq. 2.3). In the context of a magnetic field, the Lorentz force states that the force applied to a charged particle in a magnetic field is perpendicular to both the magnetic field vector, $\vec{B}$, and the velocity vector of the particle, $\vec{v}$. The direction of the force is determined using a right-handed coordinate system, and therefore the right-hand vector rule shown in Figure 2.3 can be used to identify the direction of the Lorentz Force. To determine the direction of the force, the cross product of the instantaneous velocity and the magnetic flux density is first calculated. This yields the direction of the force acting on a positive charge: if the charge is negative, then the force direction must be inverted.

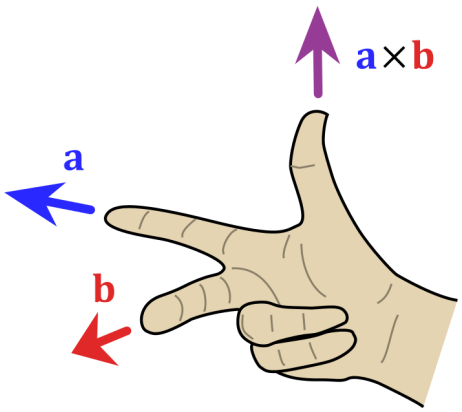


Figure 2.3: The right hand vector rule for determining the direction of the cross product of two vectors.

It is inferable from the Lorentz force equations that a particle must be charged (*charge* ≠ 0) in order to generate a force in a magnetic field, and furthermore that the velocity must have a perpendicular component relative to the magnetic field ($v_{\perp} > 0$) in order to induce a force on the particle. As the Lorentz force always acts perpendicular to the velocity vector, a magnetic field cannot do work on a particle; this means the Lorentz force cannot alter particle speed.

2.2.2. Principles of Magnetic Fields

Magnetic fields are vector fields, whereby the vector can take one of two related (but distinct) forms: magnetic flux density (**B**) or **magnetic field strength** (**H**). In the context of calculating the mechanics of charged particles in a magnetic field, shown in Equation 2.3, it is **B** that is relevant to the motion of charged particles in a magnetic field.

Bar magnets have a magnetic field shape that resembles the geomagnetic field, and so initial research was done to attempt to calculate magnetic flux density relative to a bar magnet. Heuristic formulae for calculating **B** have been derived for the most common magnet shapes, including the bar magnet (Camacho and Sosa, 2013). The formula requires for the vector location to be positioned on the central axis that intersects the North and South poles.

A current-bearing wire induces a circular magnetic field that wraps around the wire, which is dissimilar from the desired shape of the geomagnetic field. However, when the wire is wrapped into a tight coil, the induced field resembles that of the geomagnetic field/bar magnet. Similarly to the physical magnet formulae, a practical formula for calculating **B** at an arbitrary location relative to the coiled wire could not be found.

The discussed approaches for calculating **B** that are derived from applied physics have limitations in their application; making them inappropriate for calculating **B** at arbitrary locations relative to the source. As applied physics formulae proved inappropriate for this work, solutions derived from theoretical physics were sought.

The notion of a magnetic dipole is a theoretical device that enables calculations to be simplified in the domain of electromagnetism. The following formula calculates the magnetic flux density at an arbitrary point in space relative to a magnetic dipole:

$$B(\vec{r}) = \frac{\mu_0}{4\pi}\left[\frac{\vec{m}}{r^3} - \frac{3\vec{r}(\vec{m}\cdot\vec{r})}{r^5}\right]. \tag{2.4}$$

where, $\vec{m}$ is the **magnetic dipole moment**; $\vec{r}$ is the displacement vector between from the magnetic dipole to the selected point in space; and μ_0 is the **vacuum permeability** constant (4π x 10^{-7} N/A^2) (Chow, 2006). A magnetic dipole, like the bar magnet, has a magnetic field shape that

resembles the geomagnetic field, making Equation 2.4 a crucial formula in the implementation of the EML.

2.2.3. Earth-Solar Interaction

The shape of the geomagnetic field resembles that of a magnetic dipole. A comparison of the geomagnetic field in the context of the **magnetosphere** with an ideal magnetic dipole's magnetic field is shown in Figure 2.4 and Figure 2.5.

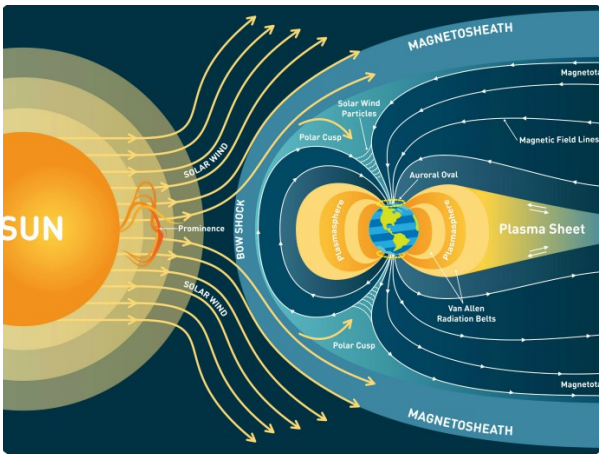


Figure 2.4: An artistic representation of the structure of the Earth's magnetosphere and solar wind interaction.

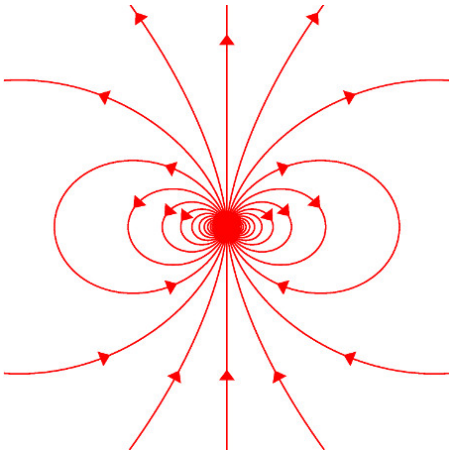


Figure 2.5: A magnetic dipole's magnetic field.

Incoming solar wind is deflected around the curvature of the Earth due to the interaction between the solar wind particles; magnetospheric plasma and the geomagnetic field. The arc that marks the high change in density of solar wind as the solar wind interacts with the magnetosphere is referred to as the **cosmic bow shock**, or simply bow shock. Relatively small quantities of solar wind can get trapped within the **polar cusps**, where particles travel in a helical motion towards the magnetic poles before deflecting back.

2.3. Related Work

Work that directly relates to the EML is limited. A public implementation of the Lorentz force has been found, titled 'lorentz-force-unity' (TheNumber5, 2021), but the simulated magnetic fields are uniform and constrained to the *y*-axis. However, this solution also implements the electric field component of the Lorentz force in the *z*-axis, allowing for visually interesting helical shapes to form when both forces are applied in tandem. Simulations provided by this solution are shown in Figure 2.6 and Figure 2.7.

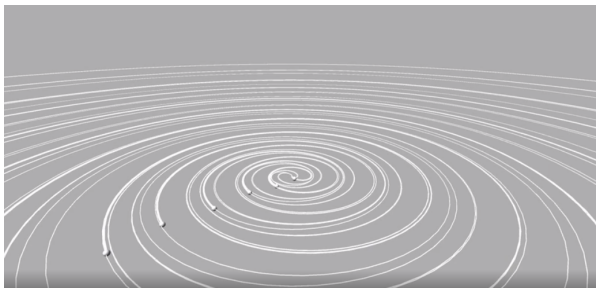


Figure 2.6: An output of the 'lorentz-force-utility' where particles are influenced by the magnetic component of the Lorentz force.

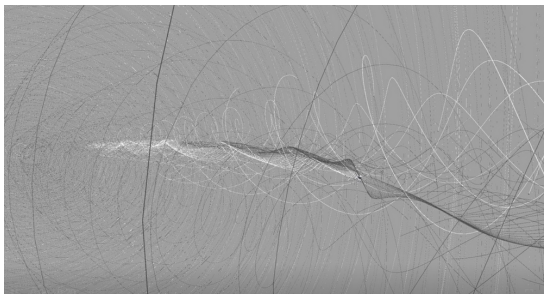


Figure 2.7: An output of the 'lorentz-force-utility' where particles are influenced by both the electric and magnetic components of the Lorentz force.

Despite this difference in approach, the structure of the solution provided insight into a potential approach to structuring the EML.

An extensive Unity library called 'Magnetodynamics' once provided numerous magnetodynamic effects, including a simulation of the Lorentz force due to fields induced by magnetic dipoles. This project was publicly available from Unity's asset store, however it is no longer accessible. Despite the project itself being inaccessible, the documentation still exists (Pivarski, 2012). This documentation provided great insight into how the solution was structured, and how Unity's built-in systems can be utilised to achieve the desired solution.

Chapter 3. Design

In this chapter, the architecture of the system is described and motivated. Following this, a technical look into the classes in terms of their properties and methods is provided. Finally, the design decisions related to the two subsystems that comprise the EML, the 'Magnetic Dipole component' and the 'Charged Particle System component', are analysed in depth.

3.1. System Architecture

The architecture of the EML is largely informed by Unity's built-in systems; particularly in the respect of utilising Unity's built-in systems whenever possible and following Unity's established design patterns when it is not. Utilising the existing systems provides several benefits, including:

- Increasing the learnability of the solution for Unity developers familiar with the built-in systems.
- Reduces the risk of the solution exhibiting incompatibilities with the existing systems.

- Minimising the lines of code required to implement the solution by delegating functionality to built-in systems. This results in:
    - Minimising the file size of the library.
    - Minimising the risk of bugs and errors being present.
    - Reducing the development time of the solution.
    - Minimising the time required to support the solution in the future as the Unity framework is updated.

The architecture diagram for the system is shown in Figure 3.1, with the two subsystems that comprise the EML being shaded in green. The system can perhaps be best understood by considering four input subsystems (*magnetic dipole GameObject* **Transform**; Magnetic Dipole **component**; Charged Particle System component; Particle System component) affecting the behaviour of two output systems (**gizmo** renderer; particle renderer). Please note that '*magnetic dipole GameObject'* refers to the **GameObject** that contains the Magnetic Dipole component, and the term 'Transform' following this is referring to the Transform component belonging to that GameObject. The Unity game engine user interface (**UI**) has two distinct forms for altering the component properties - a scripting interface and the graphical user interface (**GUI**). Input subsystems can be interacted with through either UI paradigm, but this paper typically focuses on the GUI, as it is easier to communicate, particularly through figures, compared to the scripting interface.

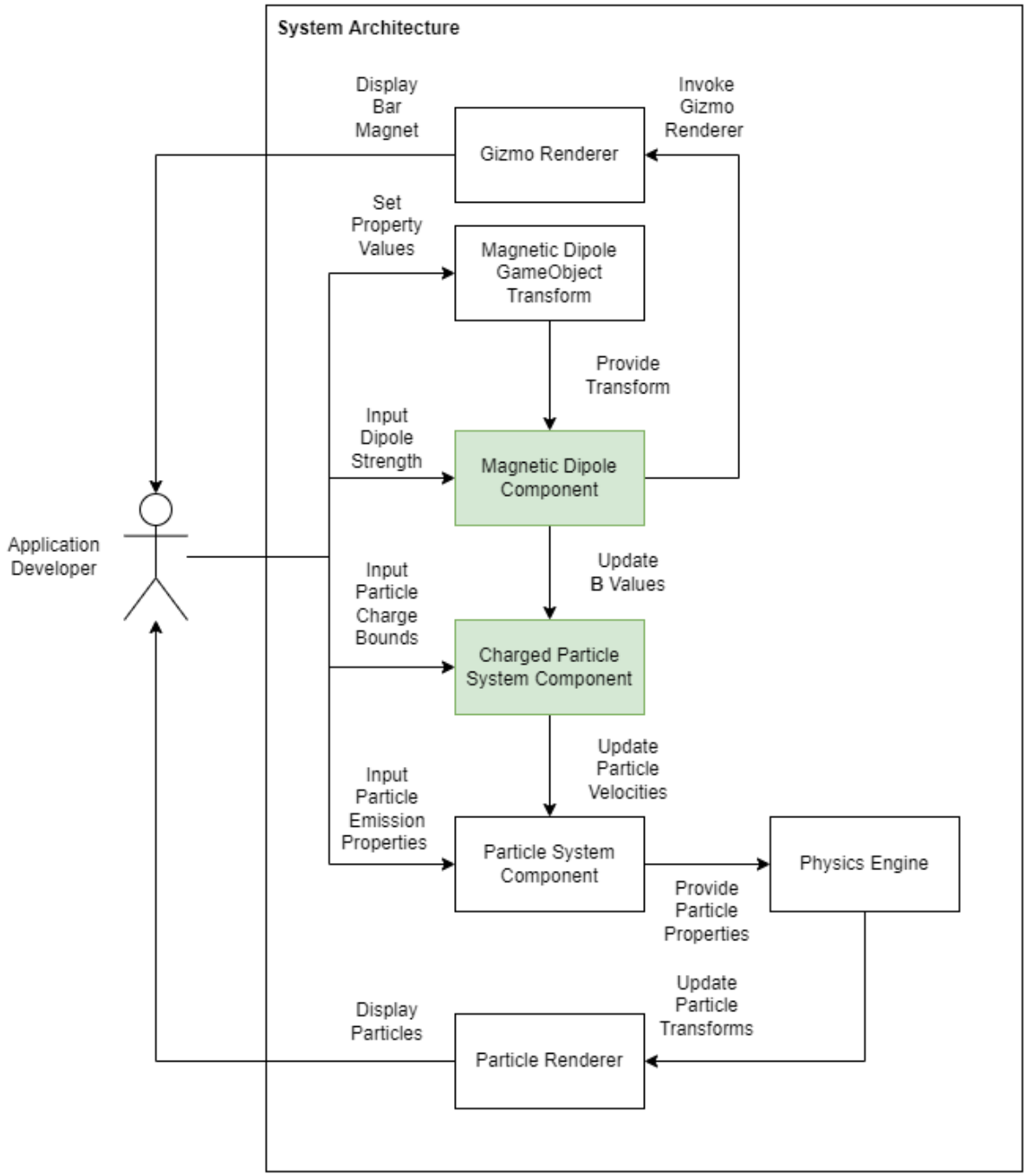


Figure 3.1: The system architecture diagram. The green shaded boxes denote the subsystems that comprise the EML.

Of the two system outputs, the drawing of the bar magnet gizmo has reduced complexity. The output is solely achieved through setting/updating the *magnetic dipole GameObject* Transform. The Transform's *position* and *rotation* properties are systematically accessed by the

Magnetic Dipole component, which then informs the orientation and position of the gizmo drawing. The drawing itself consists of a bar magnet, and the magnetic dipole moment points from the centre of the blue half of the magnet (South) through to the centre of the red half (North).

The other output system is the Particle Renderer, responsible for displaying the particles whose motions are modified by the EML, which relies on a combination of all four user inputs combining to achieve particular behaviour. The effect that the four output subsystems have on the simulation is described below:

- The *magnetic dipole GameObject's* Transform has assignable *position* and *rotation* properties. These properties alter **B** calculations, as it affects the magnetic dipole moment vector and varies the displacement/distance between Particles and GameObject *position*.
- The Magnetic Dipole component has an assignable *strength* member that affects **B** calculations by scaling the magnetic dipole moment. This component is responsible for calculating **B** at an arbitrary location relative to the component's Transform.
- The Particle System component is responsible for determining the initial *velocity* and displacement (relative to *subscribed magnetic dipoles*) through a number of interrelated assignable members. This affects Lorentz force calculation both directly via Particle *velocity*, and indirectly through **B** calculations as the Particle System informs on the *position* of Particles.
- The Charged Particle System component has an assignable *charge bounds* property and an application programming interface (**API**) for updating this property via scripting. This affects the *charge* values assigned to Particles, and therefore affects the direction and scale of the Lorentz force. It also has an assignable *subscribed magnetic dipole* member that holds the set of Magnetic Dipole components that should influence the *attached particle system's* emitted Particles. This component is responsible for retrieving **B** values from all *subscribed magnetic dipole* instances and Particle *charge* values, and using them to accelerate the Particle as predicted by the Lorentz force.

A summary of the function/role that the eight elements (including the EML end user) have in the system is provided in Table 3.1.

Element Name	**Element Description**
Application Developer	The end user of the system. Modifies the following four input subsystems (either through scripting, or using the Unity GUI) to produce the desired result in the two output systems (particle renderer and gizmo renderer). The input subsystems are listed below, with the specific inputs for the given

	subsystem supplied in brackets: 1. The *magnetic dipole GameObject* Transform 2. The Magnetic Dipole component 3. The Particle System component 4. The Charged Particle System component
Gizmo Renderer	Responsible for drawing the gizmo shapes provided by the Magnetic Dipole component into the Unity GUI, for visual debugging purposes.
Magnetic Dipole GameObject Transform	The Transform determines the *rotation* and *position* of the Magnetic Dipole magnetic field, and by extension the bar magnet gizmo drawn into the Unity GUI.
Magnetic Dipole Component	One of two subsystems that form the EML. Responsible for retrieving the *magnetic dipole GameObject* Transform. Responsible for invoking the Gizmo Renderer, and providing it with the Transform properties of the *magnetic dipole GameObject*. This allows a conventional bar magnet image to be drawn to the GUI and oriented to the Transform, for the purposes of visual debugging. Responsible for calculating **B** at locations provided by a Charged Particle System component.
Charged Particle System Component	One of two subsystems that form the EML. Responsible for assigning *charge* values to Particles emitted by the Particle System component. Responsible for retrieving **B** values calculated by the Magnetic Dipole component and assigning them to Particles emitted by the Particle System component. Responsible for updating the *attached particle system*'s Particle *velocity* members, as predicted by the Lorentz force.
Particle System Component	Responsible for controlling the values of Particle members; the most relevant being *position* and *velocity* - particularly at the time of emission. *Velocity* values from this component's emitted Particles are updated via the Charged Particle System component.
Physics Engine	Responsible for retrieving the Particle *velocities* from the Particle System component, and using them to update the Particles' Transforms.

Particle Renderer	Responsible for retrieving Particle references emitted by the *attached particle system* member, and drawing the Particles to the GUI.

Table 3.1: The descriptions of the eight elements that compose the system architecture.

3.2. Class Hierarchy

The complete hierarchy of classes relating to the EML is large and complex, owing to the depth of the Unity game engine codebase. In this section, the directly related classes to the EML are detailed, and the relationships between them are explained. They are depicted in the class diagram shown in Figure 3.2. The EML is composed of three classes: ChargedParticleSystem, ChargeBoundPair and MagneticDipole. The MagneticDipole class is considered its own subsystem: the Magnetic Dipole component. The other two classes together form the Charged Particle System component.

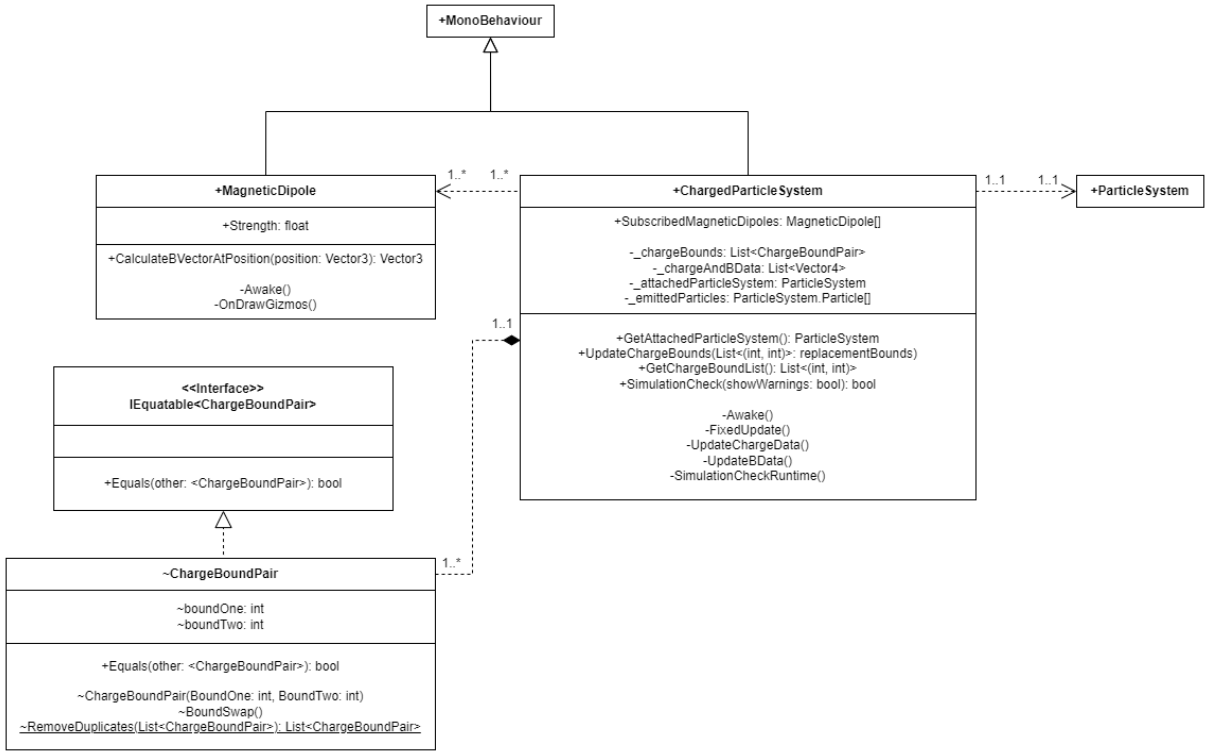


Figure 3.2: The class diagram that describes the relationships between the solution classes and the most relevant built-in classes. Also includes the complete list of properties and methods for the solution classes.

ChargedParticleSystem and MagneticDipole both extend the MonoBehaviour class, which allows them to behave as components that can interact with the Unity GUI. They share a

dependency relationship, whereby a ChargedParticleSystem instance holds references to MagneticDipoles; they use these references to retrieve **B** values for the calculation of the Lorentz force. Both a MagneticDipole instance and a ChargedParticleSystem instance can exist without the presence of the other, however, the system only operates correctly when a ChargedParticleSystem instance exists, and it holds at least one MagneticDipole instance.

Another dependency exists, between ParticleSystems and ChargedParticleSystems, in a strict one-to-one relationship. The ParticleSystem provides Unity's built-in particle emission control, while the ChargedParticleSystem assigns charge values and **B** values to those particles. A ParticleSystem can exist without the presence of a ChargedParticleSystem instance, however attempting to achieve the reverse will result in a compilation error.

The ChargedParticleSystem has a compositional relationship with the aforementioned internal class, ChargeBoundPair. The class provides an API that simplifies the process of modifying bound values in the ChargedParticleSystem class. The class is entirely hidden from the user, with access to the properties of the class only possible through ChargedParticleSystem methods. This restriction was added because the class' only purpose is to provide convenience functionality directly to the ChargedParticleSystem; the structural design abstracts the underlying complexity of how the ChargedParticleSystem component operates.

The ChargeBoundPair class implements the IEquatable interface, which is required for removing duplicate values held in the *charge bound* List. The interface requires implementing classes to implement the *Equals()* method, so that equality between instances can be determined. If two instances are found to be equal, one of these elements is removed

3.3. Charged Particle System Component

A fundamental system to the solution is Unity's built-in Particle System component. This component is responsible for handling the rendering, physics and visual aspects of emitted particles. The Particle System acts as a strong base for simulating charged particle emission - however, a method of assigning a *charge* value to each Particle was required to compute the Lorentz force vector acting on each Particle.

The first attempt to integrate the Particle System into the solution was to extend the ParticleSystem class. This would have involved simply adding the electromagnetic force to the selection of built-in effects, and allowing the developer to add the Charged Particle System component to a GameObject instead of the Particle System component. However, this approach was not possible, as the Particle System class was not made open for extension. A compositional approach was therefore taken; this involved the creation of the novel Charged Particle System component, which requires a Particle System component to also be applied to the same object. This approach guarantees a reference to the Particle System is accessible within the Charged Particle System **script**, which is stored in the *attached particle system* member.

The EML requires Magnetic Dipole components to communicate with associated Charged Particle System components. This can be achieved in three ways: by having a controller component that has access to both Magnetic Dipole and Charged Particle System references; by

giving Magnetic Dipoles access to the Charged Particle System references that they can influence, or by giving Charged Particle Systems access to the Magnetic Dipole references that can influence them. In the solution, Charged Particle Systems have access to Magnetic Dipole references through the *subscribed magnetic dipoles* member; this follows the established pattern set by the Particle System-Particle System Force Field component pair, whereby Particle System Force Field references are passed into Particle Systems that they can affect. This structure means that Charged Particle System components are given complete control of the charged particle deflection behaviour in the solution, with this component only deferring to the Magnetic Dipole component when needing to retrieve updated **B** values for the particles.

The Particle System component contains a module called 'External Forces' which handles how the Particle System interacts with Particle System Force Field components. Although the Particle System Force Field component is unused in the solution, it was desirable for the module to control the particle-magnetic field interaction as much as possible; this ensures that the expectations of developers that are unfamiliar with the solution are met, and additional control of the system is provided to them. Specifically, the interaction only occurs if the module is enabled and the effect strength is scaled by the module's *multiplier* member.

3.4. Magnetic Dipole Component

Unity provides a companion component for the Particle System component, named the Particle System Force Field. This component simulates forces that can be applied to Particles, including a simulation of gravity; object rotation, and drag. This would have been the ideal base for the *Magnetic Dipole* script, which similarly applies a force, the Lorentz force, to charged particles, but it proved unsuitable for the needs of this work. This is because only a limited portion of the Particle System Force Field's functionality was exposed for extension, and developing a workaround would have added complexity to the EML's codebase and architecture without any significant benefit. As a result, the Magnetic Dipole component had to be isolated from Unity's built-in systems, however, due to the Charged Particle System component performing most of the computation in the solution, this did not add complexity to the component's design.

3.4.1. Gizmo Visual Design

The Transform of the *magnetic dipole GameObject* plays a critical role in producing a simulation of the Lorentz force such that the charged particles follow desirable paths. The *rotation* and *position* vectors associated with the Transform affect the path of Particles in the presence of a Magnetic Dipole component, as the **B** values generated will differ. Therefore, the user should be given cues for easing the setup of the *magnetic dipole GameObject*. To this end, a gizmo is drawn by the Magnetic Dipole component to allow the user to more easily visualise the path that *Particles* will take. One approach to this could be to draw magnetic field lines around the *magnetic dipole GameObject*, which would be the most direct and intuitive representation of

the magnetic field. However the built-in gizmo drawing API is not extensive enough to draw the precise curves and symbols that would be required for it to be a useful visual cue. The result is that either one of two approaches could be taken: a solution could be built to handle more complex drawing onto the unity GUI, or a more symbolic method of representing a magnetic field could substitute. The former approach was considered too complex to implement for the purposes of this work, meaning the latter approach was selected.

Magnetic dipoles do not have a directly associated symbol or appearance, with the closest analogue being a bar magnet. Selecting a bar magnet has the benefits of being widely recognised, as well as making the drawing of the gizmo relatively straightforward to implement. A bar magnet is conventionally coloured half red and half blue; split horizontally. They are optionally marked with 'N' and 'S' lettering to denote the North and South magnetic poles respectively, with a pole located at the end of each coloured half. However, the lettering may be omitted as the red and blue colours conventionally denote the North and South magnetic poles respectively. The shape of the magnetic field can be inferred from the orientation of the gizmo (by identifying the vector pointing from the centre of the blue face located on the bottom of a bar magnet, to the centre of the red face located at the top of the bar magnet), and having an appreciation of the shape of a bar magnet's magnetic field (noting that magnetic fields are directed from North to South magnetic poles). The more simplistic design without the lettering was chosen as the final visual design for representing the *magnetic dipole GameObject* Transform. Figure 3.3 shows a diagram of a conventional bar magnet which is compared with the final gizmo design, as it appears in the Unity GUI, shown in Figure 3.4.

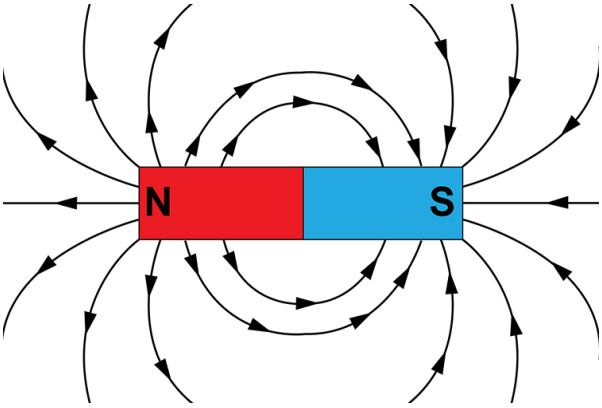


Figure 3.3: A conventional bar magnet and its associated magnetic field.

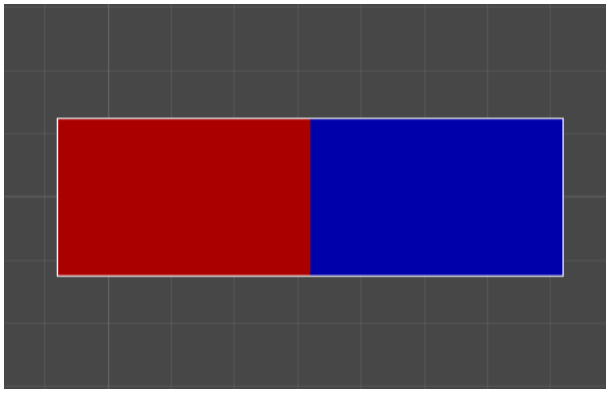


Figure 3.4: The gizmo representation of a bar magnet The gizmo indicates that the magnetic dipole moment is oriented directly to the left.

Chapter 4. Implementation

This chapter contains the details of the EML's implementation. To begin, a description and explanation of the file structure in the EML. Following this, the data structures used in the solution are motivated, particularly where they relate to the UI of the four input subsystems. To conclude, the most significant algorithms used in the EML are discussed.

4.1. Solution Script Structure

The EML is composed of two C# files: the MagneticDipole script (which contains the MagneticDipole class) and the ChargedParticleSystem script (composed of two classes: the ChargedParticleSystem class, and the internal ChargeBoundPair class). A two-file structure for the EML was selected, because at its most fundamental level, the Lorentz force is composed of two elements: charged particles and magnetic dipoles. It was necessary for each of these elements to have at least one associated script, so that the element's behaviour can be applied to GameObjects as components: meaning there should be a minimum of two scripts in the EML. Given three classes were written in the solution, a possible structure could have been for the EML to be composed of three distinct files, each containing its own class. However, users would likely find this structure less intuitive to understand, because it would be unclear what the purpose of each script is, particularly ChargeBoundPair, and how it should be used. Reducing the number of classes to two removes this potential confusion, and therefore strengthens the user's conceptual model of how the system works, based on the files provided in the library.

4.2. Data Structures

This section focuses on the data structures of the EML, and not the system overall, as it is outside of this work to detail the implementation of Unity's built-in systems. Therefore, the Charged Particle System and Magnetic Dipole components will be discussed in this section.

The Charged Particle System component has two UI inputs: an array of Magnetic Dipole components that influence the *attached particle system's* emitted Particles, *subscribed magnetic dipoles*, and an array of ChargeBoundPair instances that provide handling of *charge* values that are assigned to Particles, *charge bounds*. There are two built-in C# data structures for collections that can readily be displayed on the Unity GUI: C#'s generic List, and arrays. The major differences between these data structures is that Lists can be dynamically resized unlike arrays, but arrays can be iterated through more efficiently than Lists. Using dynamic data structures for *subscribed magnetic dipoles* and *charge bounds* was not required in this work; therefore, arrays were selected in both cases as the more performant option.

Charged Particle System components use the *charge bounds* member to calculate Particle *charge* values. This is achieved by retrieving a random ChargeBoundPair element from the *charge bounds* array, and then selecting a random integer (inclusive) within its members, *bound*

one and *bound two*. This data structure was designed to provide a compromise between control of the assignment of *charge* values to Particles, and convenience. It allows for an arbitrarily large range of *charge* values to be assigned with minimal setup. In addition, a specific range of values can easily be omitted within a larger range of values, by simply adding two ChargeBoundPair instances with the range between *bound one* *bound two* set either side of the range to omit. For instance, if the user would like to include the range -5 to 5 but without a neutral charge (*charge* = 0), then they would add the following elements to the array: {-5, -1}; {1, 5}. The decision to use integer values as opposed to real numbers to represent charge may seem counterintuitive at first, as it seems to unnecessarily reduce the control of the developer without any obvious reasoning. However, charge is a **quantized** property and so using integer values most reflects this quality in the most elegant way.

The introduction of this bound system required designing an API for allowing users to edit the property via scripting and validation. At least one ChargeBoundPair instance must be in the *charge bounds* collection for the system to operate, and so *charge bounds* cannot be emptied once populated. Additionally, the ChargeBoundPair class automatically removes duplicate values from the *charge bounds* collection, which if present would reduce the EML's performance.

The interface of the Magnetic Dipole component consists of a single real number input that is used to assign the *strength* member of the associated magnetic field. Using a real number value gives the developer fine-grained control over the output of the simulation, in contrast to the coarse control of integer values. The other notable input for the EML, *charge bounds* in the Charged Particle System component, provides relatively coarse control of the simulation through discrete integer values and randomness in the assignment of *charge* to each Particle.

Throughout section 4.3, Particle *charge* and **B** properties are referred to as distinct properties of a given Particle. However, they are not stored internally as distinct floats (not integers, to account for the ambiguity problem discussed in section 4.3.4) and Vector3 members, as might seem the obvious solution. Rather, these two properties are combined in a single **Vector4** data type, *charge and B field data*, that can hold the three components of **B** in the *x*, *y*, and *z* components, with the charge stored in the *w* component. To assign custom data to Particle objects, the data must be transmitted via one or two Vector4 data streams. It would have been possible to use two data streams to carry the same data, however this was avoided for two reasons: it means users now have an unused data stream available to use, if they require it, and it also halves the amount of data that needs to be processed, theoretically improving system performance.

4.3. Algorithm Breakdown

Four key algorithms form the basis of the implementation of the EML. The algorithms each perform one of the following tasks: calculating **B** for a given position relative to a given Magnetic Dipole*;* updating **B** for each Particle emitted by a Charged Particle System's *attached particle system*; setting the *charge* for each Particle emitted by a Charged Particle System's

attached particle system, and finally changing the *velocity* of each Particle to simulate the Lorentz force.

Each algorithm forms a part of the computation for the Lorentz force simulation, with the algorithm responsible for displaying the gizmo being intentionally omitted, as the pseudocode for this is relatively trivial and does not provide enough scope for meaningful discussion. It should be noted that each algorithm overlooks minor implementation details in the interest of understandability, but important differences between the implementation and pseudocode will be highlighted as appropriate.

4.3.1. Implementation Goals

The EML has been implemented such that performance has been prioritised over precisely simulating the real-world physics of Lorentz forces. Prioritising performance over precise simulation is common practice in the implementation of physics simulations in game engines, including the simulations built into Unity. This is critical as it gives users the freedom to utilise the system without compromising on the user experience of their projects. The aim of the simulation is to allow the user to visually reproduce the desired effect of the physics phenomena; not to have the values used in the computation of the simulation be accurate to real-world values. It is preferable to produce a more performant system that requires some trial and error of values to produce the desired result, than a system where the output is determinable prior to execution, but with worse performance. Where appropriate, instances of this principle being applied will be highlighted in the explanation of pseudocode in the following sections of this paper.

Performance has also been improved through only executing expensive computation when the solution's internal state passes validation checks. For individual Particles, if the *charge* is zero or the *velocity* magnitude is zero, then it is not simulated. Furthermore, the Charged Particle System component is disabled altogether in the event of any of the following criteria being met: the length of the *charge bounds* collection is zero; the length of *subscribed magnetic dipoles* collection is zero; the Particle System's External Forces module is disabled, or the *simulation space* of the *attached particle system* is not set to 'World space'. These validation checks ensure that performance is optimised, the system behaves as expected, and it also reduces the chances of accessing empty collections and attempting to perform computations on an empty reference, typically causing errors.

4.3.2. Magnetic Flux Density Calculation

The algorithm shown in Figure 4.1 applies Equation 2.4; it takes an input vector of a point in space and returns the vector **B** induced at that point by a Magnetic Dipole. The implementation uses Unity's **Vector3** data types for both the input and output, as both position and velocity in a 3D domain have three components. The $\frac{\mu_0}{4\pi}$ constant present in Equation 2.4 has been omitted from the calculation of **B** for the purposes of optimisation. This constant can be omitted, as it simply serves the purpose of scaling **B**, which in turn scales the Lorentz force

applied to the Particle. Both the Particle's *charge* value and the multiplier in the Particle System External Forces module can adjust this scaling in the same manner, making the inclusion of the constant redundant.

A minor difference between the pseudocode and the implementation is how *orientation vector* is retrieved and calculated. In the implementation, this is not directly a property of a Magnetic Dipole and it is instead calculated in the algorithm. The implementation code snippet used to achieve this is:

```
transform.TransformDirection(Vector3.up).
```

This function takes a Vector3 input that is interpreted to be relative to the GameObjects' orientation, and then outputs this vector transformed to the global space. The *transform* member references the GameObject's *Transform* component, which defines the orientation via its *rotation* property. The argument *Vector3.up* is a unit vector that points directly upwards, and this sets the magnetic dipole moment also directly upwards relative to the GameObject's *rotation*.

Algorithm 1: Calculates B values at a given position relative to a Magnetic Dipole GameObject.	
Inputs	**pos**
Outputs	A three-component vector that holds **B** acting on a point in space; resolved into *x*, *y* and *z* components.
Initialisation	Get *s,* d*ipole.**orientationVector***
Assumptions	*s*, **orientationVector** and *dipole* are accessible to the method.
Key	● ***d*** - the displacement between the Magnetic Dipole and a Particle. ● *dipole* - the magnetic dipole that is inducing the magnetic field. ● *dipole.***orientationVector** - the normalised three-component vector that is directed towards the relative 'up' directional vector of the Magnetic Dipole translated to global space. ● *dipole.***pos** - the 3D position vector of a Magnetic Dipole. ● ***l*** - represents the left term found in the square brackets inside Equation 2.4. ● ***m*** - the magnetic dipole moment. ● ***pos*** - the 3D position vector of a Particle. ● ***r*** - represents the right term found in the square brackets inside Equation 2.4. ● *s* - the strength of the magnetic dipole. ● *vector_dot(v1, v2)* returns the dot product between the two supplied vectors, *v1* and *v2*.
function CalculateBVectorAtPosition(**pos**)	

```
        m = s * dipole.orientationVector / || dipole.orientationVector ||
        d = pos - dipole.pos
        l = 3 * vector_dot(m, d) * d / || d ||^5
        r = m / || d ||^3
        return l - r
end function
```

Figure 4.1: Pseudocode for the algorithm that returns the **B** vector for a given point in space, relative to a given Magnetic Dipole. Part of the MagneticDipole class.

4.3.3. Magnetic Flux Density Assignment

The algorithm shown in Figure 4.2 iterates through all emitted Particles from the current Particle System, and assigns a **B** value to them at its current position relative to all *subscribed magnetic dipoles*. This algorithm relies on the algorithm shown in Figure 4.1 to retrieve the appropriate **B** value.

A possible optimisation that could have been implemented is initially determining whether a Particle has a neutral charge, and then bypassing the calculation of **B** for this Particle by immediately iterating to the next one. This is possible because a Particle with a neutral charge cannot have a Lorentz force applied, and so it would waste processing resources to compute it. The decision was made to exclude this check in the interest of reducing the number of operations executed in the general case of this algorithm running, which is that Particles are charged.

Algorithm 2: Assigning B values to emitted charged particles.	
Inputs	-
Outputs	Every p in $p \in P$ has a **B** value assigned to p.**b**
Initialisation	Get P, M
Assumptions	P, M are accessible to the method.
Key	• **b** - temporary variable for storing **B** for each Particle. • M - the collection of *subscribed magnetic dipoles*. • m - a single Magnetic Dipole instance. • P - the collection of currently alive Particles. • p - a single Particle instance. • p.**b** - the magnetic flux property of a Particle. • p.**pos** - the 3D position vector of a Particle.
function UpdateBData() **for each** p ∈ P	

```
            b = <0, 0, 0>
            for each m ∈ M
                b += m.CalculateBVectorAtPosition(p.pos)
            end for
            p.b = b
        end for
end function
```

Figure 4.2: Pseudocode for the algorithm that assigns **B** values to Particles emitted by a Charged Particle System's *attached particle system* member. Part of the ChargedParticleSystem class.

4.3.4. Charge Assignment

The algorithm shown in Figure 4.4 assigns a *charge* value to every Particle emitted. Unlike the algorithm shown in Figure 4.2 for assigning **B** values, once a *charge* value has been assigned to a Particle, it must not be modified. This algorithm must be called routinely over all emitted Particles to ensure the newly emitted also have a *charge* value assigned to them. This presents the problem of *charge* values being overwritten, and so a system needs to be put into place to ensure that the *charge* member cannot be overwritten. The pseudocode implies that an unset *charge* value is equivalent to *null*, and therefore a simple check for whether *charge* is null is sufficient to avoid overwriting, however this is a simplification of the implementation.

The problem derives from the *charge* of newly emitted Particles being initialised to 0 instead of a non-numeric value such as *null*. As a result, a Particle with a *charge* that has intentionally been set to 0 (i.e., a neutrally charged Particle) is indistinguishable from a Particle that has not yet been assigned a value. The solution, therefore, is to ensure that it is possible to differentiate between Particles with an unset *charge* property, and Particles with a neutral *charge* property. The flowchart shown in Figure 4.3, illustrates how this system is implemented.

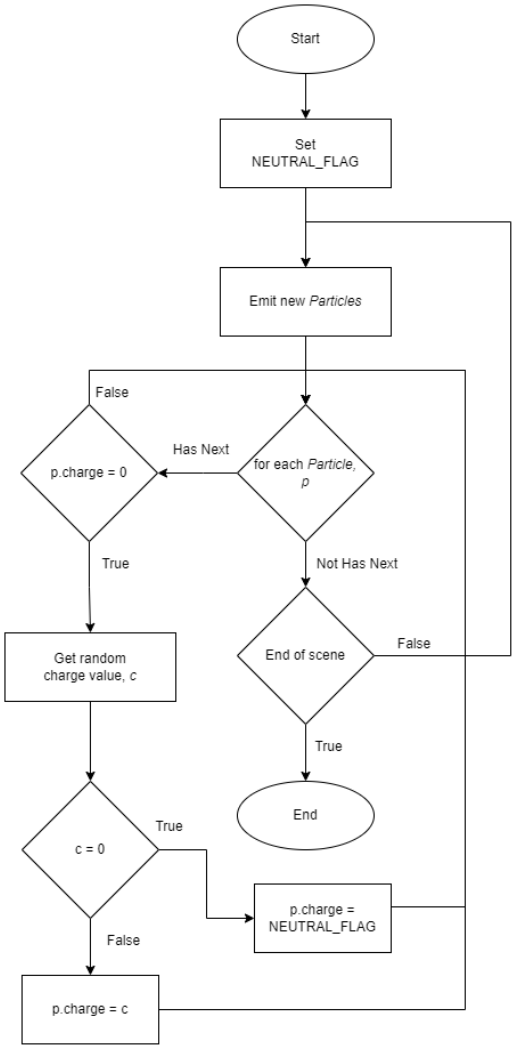


Figure 4.3: A flowchart that visualises how the algorithm that prevents charge value overwriting works.

The system starts by setting the *NEUTRAL_FLAG* constant. In the implemented system, this is 0.5, but its value is arbitrary so long as it is not in the set of potential *charges* assigned by the ChargeBoundPair system ($NEUTRAL\_FLAG \notin \mathbb{Z}$). Since *NEUTRAL_FLAG* is non-integer, it is impossible for its value to be assigned as a *charge* value using the process described in Section 4.2. Assigning *NEUTRAL_FLAG* to a *Particle's charge* ($charge \notin \mathbb{Z}$) allows the Particle to be uniquely identified as neutral via its *charge* alone when compared to Particles with unset *charge* ($charge = 0$), and set but charged Particles ($charge \in \mathbb{Z}, charge \neq 0$).

After setting *NEUTRAL_FLAG*, each alive Particle belonging to a Particle System is iterated through. At the start of each iteration, the *charge* is queried: if the value is not zero ($charge \neq 0$), the Particle's *charge* has already been set, and the next Particle is immediately

retrieved. If the charge is zero (*charge* = 0), however, we know the *charge* has not been set, and we progress to assigning a *charge* value to it.

The *charge* value, *temp*, is first retrieved from the *charge bounds* member, as described in Section 4.2. Instead of immediately assigning this value to the Particle's *charge* like the pseudocode suggests, the retrieved value is queried. If *temp* is zero (*temp* = 0), the *Particle* should be considered neutral, and therefore *NEUTRAL_FLAG* is assigned to *charge*. If *temp* is not zero (*temp* ≠ 0), then assign *temp to charge.*

This process then repeats entirely, with all Particles that have had their *charge* value set in the previous iteration being skipped in the current iteration, as all of their *charge* values are non-zero (*charge* ≠ 0) while newly emitted Particles will have a charge of zero (*charge* = 0).

As discussed in section 4.3.1, the implementation prioritises performance over realistic simulation. Particle *charge* values are represented as integers, however, an elementary charge is equivalent to $1.60217663 \times 10^{-19}$ C in SI units. This means that the stored *charge* values would all be scaled by this elementary charge value if accurately simulating real-world physics was prioritised. However, this detail is not introduced in the solution as it would add unnecessary expensive computations to the system, while only providing a scaling of the Lorentz force, which the user is able to control using the *multiplier* member of the Particle System's Emission Module.

Algorithm 3: Assigning charge values to emitted charged particles.	
Inputs	-
Outputs	Every *p* in $p \in P$ has an integer value assigned to p.*charge* unless one has already been applied.
Initialisation	Get *P*, *C*
Assumptions	● $C \neq \varnothing$. ● *c.boundOne*, *c.boundTwo* strictly store integer values. ● *P*, *C* are accessible to the method.
Key	● *C* - the collection of ChargeBoundPair instances. ● *c* - a ChargeBoundPair instance. ● *c.boundOne, c.boundTwo* - one of the two bounds that limit the range of values that can be assigned as *charge* values. ● *P* - the collection of currently alive Particles. ● *p* - a single Particle instance. ● *p.charge* - the charge property of a Particle. ● *random_element(C)* returns a single random element within *C*. ● *random_integer(x,y)* returns a randomly selected integer between *x* and *y*, inclusive. *x <= random_integer(x,y) <= y.*

```
function UpdateChargeData()
        for each p ∈ P
                if  p.charge ≠ null
                        continue
                end if
                c = random_element(C)
                p.charge = random_integer(c.lower, c.upper)
        end for
end function
```

Figure 4.4: Pseudocode for the algorithm that assigns charge values to Particles emitted by a Charged Particle System's attached particle system. Part of the ChargedParticleSystem class.

4.3.5. Lorentz Force Application

The algorithm shown in Figure 4.5 acts as a controller for the other three highlighted algorithms. It is responsible for invoking the algorithms that assign both the *charge* and **B** properties to Particles. Then, the Lorentz force simulation is applied to each individual Particle by accelerating them in the direction where the force acts upon them.

This algorithm is executed in an endless loop while the currently running scene is active. At the start of each iteration, checks are performed to determine whether the Charged Particle System component should be disabled. The component is disabled when the Emission module of the *attached particle system* is disabled, or there are no Magnetic Dipoles subscribed to the Charged Particle System. The former check adds compatibility between the Charged Particle System and Particle System components, while the latter improves system performance, by preventing the simulation from running when the simulation cannot produce an effect.

Each Particle in the *attached particle system* is iterated through, with optimisation checks performed on the Particle *velocity* and *charge* properties as described in section 4.3.1. The Lorentz force is calculated using Equation 2.3, but with the *multiplier* property of the External Forces module acting as a final multiplier for the effect.

A small, but important detail for the implementation was to switch the operands in the cross product during the calculation. This has the effect of reversing the cross product vector, which is necessary because it accounts for the difference in handedness between Unity, which uses a left-handed coordinate system, and the domain of Physics, which uses a right-handed coordinate system. The result is that the Lorentz force in the simulation will act in the same direction as real-world charged particles.

Forces are applied to Particles by simply adding the Lorentz force vector to the current velocity. This creates the effect of accelerating the project in the direction of the Lorentz force vector, without requiring the computation for accurately predicting how the Lorentz force translates to a an updated *velocity* vector,

Algorithm 4: Applying the Lorentz force to Particles emitted by the Charged Particle System.	
Inputs	-
Outputs	Every p in $p \in P$ has an updated velocity value assigned to p.**velocity**, calculated by applying the Lorentz force to the Particle.
Initialisation	Get P, M
Assumptions	• *aps.externalForces.multiplier* is automatically set to 1 unless specified by the user. • *aps*, *M*, *P* are accessible to the algorithm.
Key	• *aps.externalForcesModule.enabled* - the state of the *attached particle system's* External Forces module. If *False,* then the module is disabled. • *aps.externalForces.multiplier* - the strength of the Lorentz force effect, determined by the *attached particle system's* External Forces module. • *aps.simulationSpace* - determines what the Transform properties of Particles are relative to. The EML requires the *simulation space* to be in World space. • *M* - the collection of *subscribed magnetic dipoles*. • *P* - the collection of currently alive Particles. • *p.**b*** - the magnetic flux density property of a Particle. • *p.charge* - the charge property of a Particle. • *p.**velocity*** - the velocity property of a Particle. • *vector_cross(v1, v2)* - returns the cross product of *v1* and *v2*.

```
function FixedUpdate()
    if aps.externalForcesModule.enabled = False or
       M = ∅ or
       aps.simulationSpace ≠ World
            disable the Charged Particle System component
            return
    end if
    UpdateBData()
    UpdateChargeData()
    for each p ∈ P
        if p.charge = 0 or || p.velocity || = 0
                continue
        end if
```

```
            p.velocity += p.charge * aps.externalForces.multiplier *
            vector_cross(p.velocity, p.b)
        end for
end function
```

Figure 4.5: Pseudocode for the algorithm that applies a simulation of the Lorentz force to Particles emitted by attached particle systems. Part of the ChargedParticleSystem class.

Chapter 5. System In Operation & Process Description

In this chapter, a walkthrough of a typical session of using the EML is provided, including essential information on how to set up a Unity project so that the EML operates correctly.

5.1. EML Installation

To install the EML into a Unity project, a Unity project must first be created. This can either be a newly created project, or an existing one. Inside the root directory that contains the entire Unity project, there will be an *Assets* directory; this should contain all files that are used or manipulated in the project, including the EML.

While the EML files can also be manually copied into the *Assets* folder, the solution can be distributed as a *Unity package* file (*.unitypackage*) as a more convenient method of importing a library and its dependencies. A *Unity package* compresses the contents of the folder, and maintains the file structure of how the package was exported.

The *Import Unity Package* window, shown in Figure 5.1 allows the user to import the files into the project, and additionally have control over the directory structure of the files upon being imported.

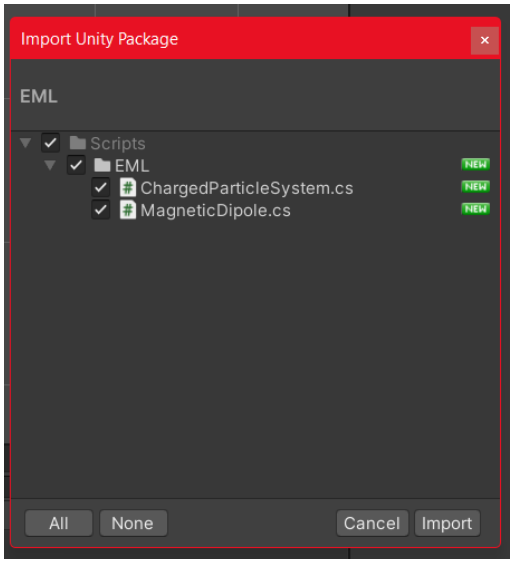


Figure 5.1: The 'Import Unity Package' tab in the Unity GUI for an opened Unity project. It shows the file structure of the EML to be imported into the project.

Once the files have been imported, either by manually adding them to the project or using the *Import Unity Package* system, the EML is now ready to be used inside the current Unity project.

5.2. System Walkthrough

5.2.1. GameObject and Component Setup

Upon the installation of the EML, the first action to perform is to create empty GameObjects that will represent either a magnetic dipole or a charged particle stream. Figure 5.2 shows how an empty GameObject can be added to the project through the *Hierarchy* tab, as well as the current GameObject structure of the Unity scene. Note, Figure 5.2 shows that three GameObjects are in the Unity scene: the two previously mentioned GameObjects, as well as an object called *Main Camera* that is automatically added into new Unity scenes. Main Camera is responsible for providing a perspective through which the application can be viewed, and this is how different angles of the system are captured in chapter 6.

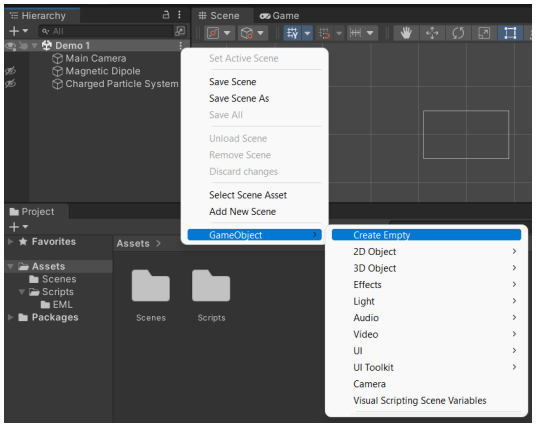


Figure 5.2: The *Hierarchy* tab on the left side of the image shows the GameObject structure of the Unity scene. There are three GameObjects present, called: 'Main Camera', 'Magnetic Dipole' and 'Charged Particle System.

Components are now able to be added to the empty GameObjects. In this demonstration, the 'Magnetic Dipole' GameObject has the 'Magnetic Dipole' script applied to it, to apply a Magnetic Dipole component. Similarly, the 'Charged Particle System' has the Particle System component applied, in addition to the 'Charged Particle System' script, which applies a Charged Particle System component. The Particle System component must be added to the *charged particle system GameObject* before attempting to add the Charged Particle System, as the Charged Particle System component depends on it.

Components are added via the *Inspector* tab. From here, components are added to the selected GameObject by interacting with the "Add Component" button in the *Inspector*. You can then search for the desired component/script, and select the desired component from a list that appears based on the text input. The process of adding components to GameObjects is shown in Figure 5.3.

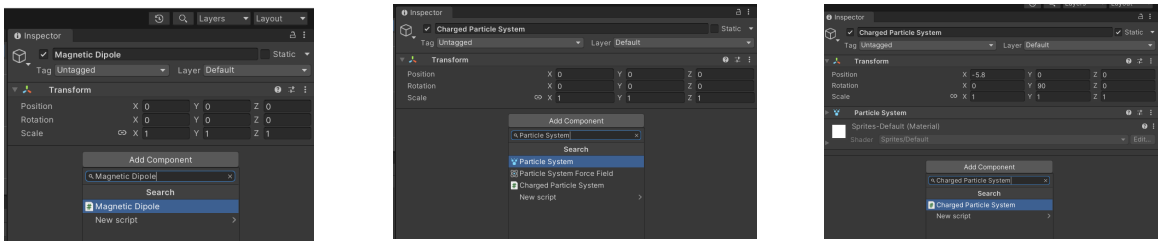


Figure 5.3: The Magnetic Dipole, Particle System, and Charged Particle System component being added to their associated GameObject. Note, the Particle System has already been added to the *charged particle system* GameObject prior to the Charged Particle System component.

5.2.2. Simulation Setup & Execution

This section will be structured by first introducing the mandatory steps to run the simulation, followed by the non-mandatory steps. This section focuses on the setup of the EML derived components. This walkthrough demonstrates a simplified example for the purposes of communicating the fundamentals of the system, and is not representative of all effects that can be achieved with the EML.

Before the application can begin, setup is required in the Charged Particle System component. This involves subscribing Magnetic Dipole components to the *subscribed magnetic dipoles* array, and then adding at least one ChargeBoundPair instance to the *charge bounds* array.

For the demonstration, two elements have been added to the *charge bounds* array. Figure 5.4 shows that the bounds {1, 1} have been assigned to 'Element 0', with 'Element 1' being assigned {-1, -1} as shown in Figure 5.5. This means Particles with a +1 and -1 *charge* will be emitted, with emission probability being the same for both charges.

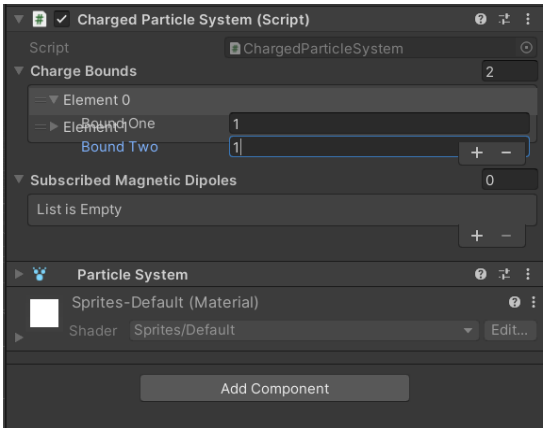


Figure 5.4: The {1, 1} bounds being added to the first element, 'Element 0'.

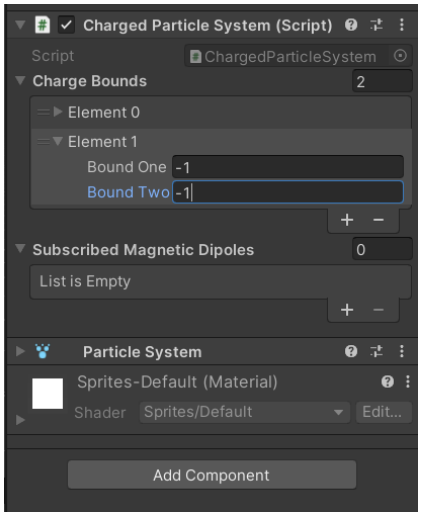


Figure 5.5: The {-1, -1} bounds being added to the second element, 'Element 1'.

Next, we must add Magnetic Dipole references to the Charged Particle Systems' 'Subscribed Magnetic Dipoles' arrays. The demonstration only contains a single Charged Particle System and one Magnetic Dipole component, and so the matter of selection is trivial. Figure 5.6 shows the dialog that appears upon interacting with the 'Subscribed Magnetic Dipoles' array in the *Inspector*. Notice, only a single object is available for selection as there is only one Magnetic Dipole component present in the scene. The result of adding the Magnetic Dipole component into the 'Subscribed Magnetic Dipoles' array is shown in Figure 5.7.

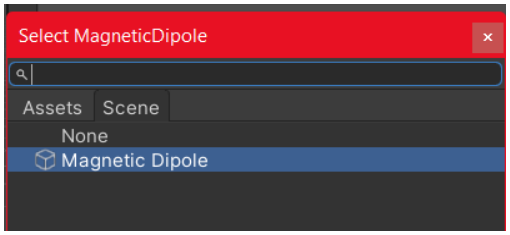


Figure 5.6: The dialog that appears upon interaction with the empty element.

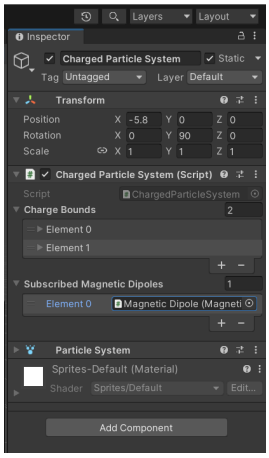


Figure 5.7: The *Inspector* tab showing the 'Charged Particle System' GameObject after adding the Magnetic Dipole component to the 'Subscribed Magnetic Dipoles' array.

The compulsory elements of the system have now been set up. Now, the non-compulsory inputs will be discussed. The final UI field that directly relates to the EML is the ‘Strength’ field from the Magnetic Dipole component, which has been set to 0.5, as shown in Figure 5.8.

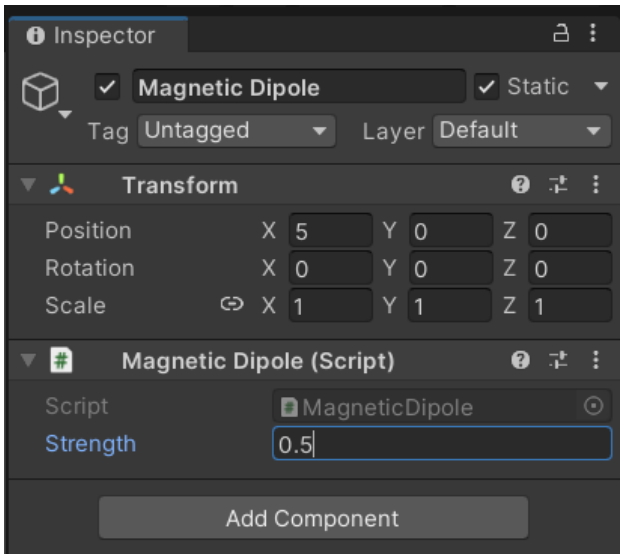


Figure 5.8: Setting the ‘Strength’ field in the Magnetic Dipole component. The value has been set to 0.5.

The final components directly related to the EML that influence the simulation are the Transform components of both the *magnetic dipole GameObject* and the *charged particle system GameObject.* In the demonstration, the Transforms have been set up such that the *charged particle system GameObject* is positioned on the left, and particles are emitting from the left side of the scene towards the right. Conversely, the *magnetic dipole GameObject* is positioned towards the right of the scene and it is directly centred in the emission cross-section. The orientation of the *magnetic dipole GameObject* is directly vertical relative to the view of the scene, signified by the orientation of the bar magnet gizmo. The positioning and orientation of GameObjects in the scene is shown in Figure 5.9.

The initial Particle trajectories are determined by the Particle System component. Notably, the Particles are emitted at random locations on a ring shaped emission surface at a *velocity* of 1.2.

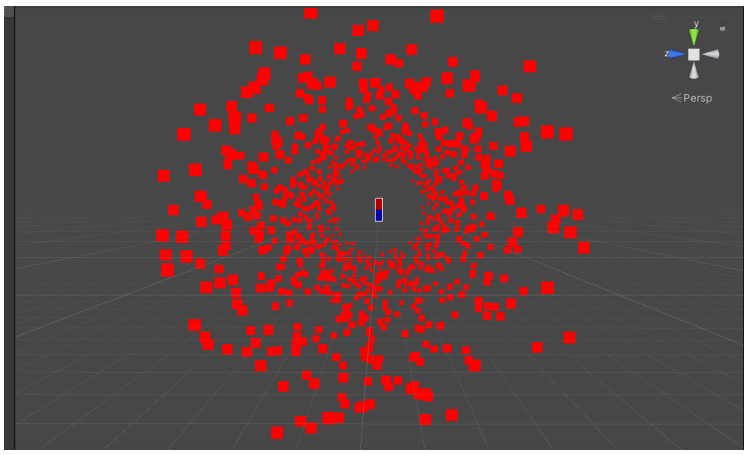


Figure 5.9: Particle emission viewed from the perspective of the source of the emission. The ring emission shape is shown, with the magnetic dipole GameObject placed centrally in the cross-section of particle emission. The particles are not being deflected.

Before the simulation is performed, a compilation process must take place, using the parameters supplied from the *Inspector*. If this process is successful, the program will be emulated.

A still image taken from the simulation using the values indicated throughout the chapter is shown in Figure 5.10. The incoming red particles can be seen forming a curved bow shock as they approach the bar magnet gizmo. The bar magnet gizmo marks the location of the *magnetic dipole GameObject*.

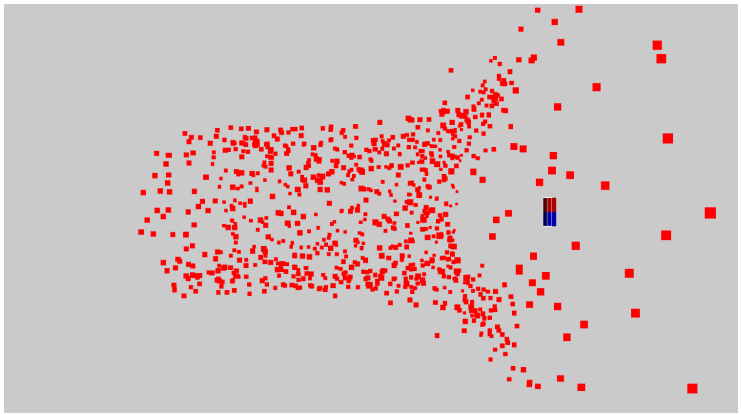


Figure 5.10: The final result of the EML Demonstration, with particles emitted horizontally to the right until a strong deflection is induced upon them, forming a bow shock.

During simulation runtime, the variables set in the *Inspector* can be modified, with the state of the simulation immediately being updated to reflect the newly inserted values. Crucially, the values that are updated during runtime will revert to what they were prior to emulation. This

provides a convenient way to experiment with new values to see how the simulation is altered, without fear of losing the values that were initially set if the new values are not producing a desirable effect, and is a helpful interaction framework to achieve a particular simulation effect.

Chapter 6. Testing

In this chapter, the EML system is tested, particularly the interactions that a user can have with the system through the Unity GUI. Three different testing scenarios are applied. First, the *charge*, *strength*, number of magnetic dipoles subscribed and the magnetic dipole *rotation* is tested. This first set of tests establishes how the properties affect the output simulation for valid system states. Following this, it is established whether the properties affect the simulation output for invalid states of the system - i.e., that the system behaves as expected when entering an invalid state. Finally, the system is applied to the PPM, to determine whether the desired particle deflection effect can be achieved.

To begin, a baseline simulation needs to be established so that comparisons can be drawn between the baseline and a simulation with a change in its properties. The simulation created in Chapter 5 will be used for this purpose, and using the same baseline properties. Figure 6.1 shows the baseline simulation in use. With each test (unless explicitly stated), all variables will remain identical between the baseline and altered simulations, apart from the variable that is to be tested. This ensures that any change observed in the system can be attributed to the change in the property that is being tested.

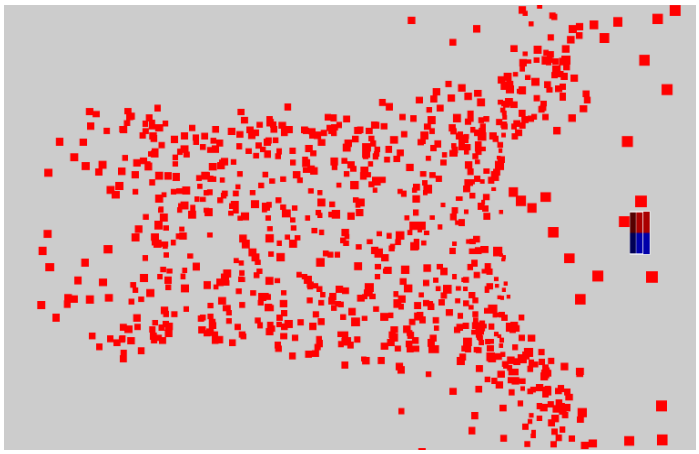
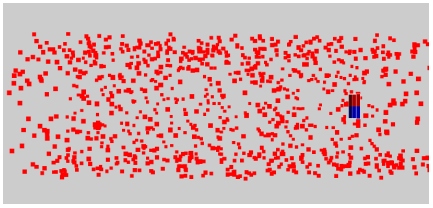
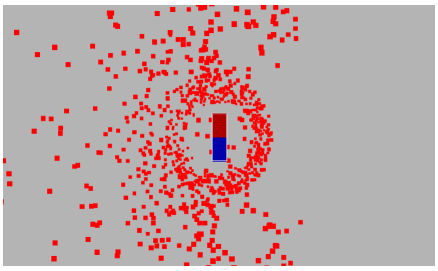
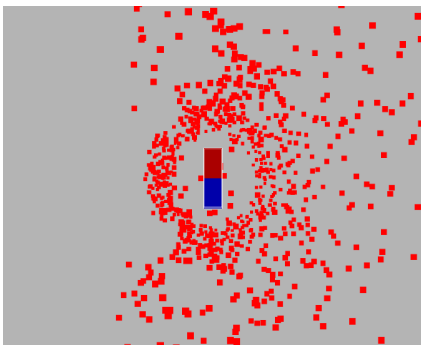
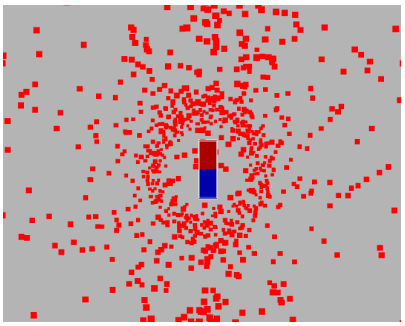
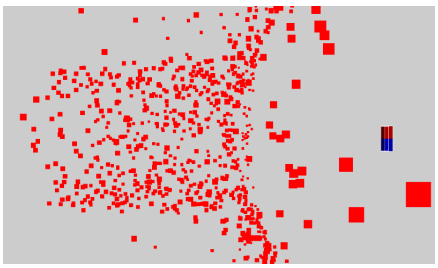
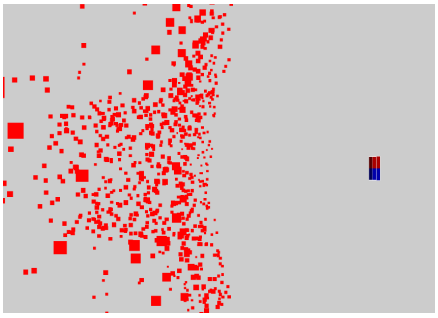
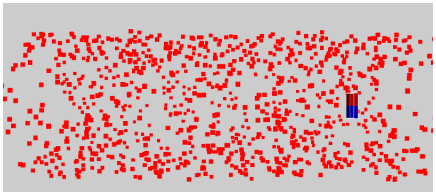


Figure 6.1: The baseline simulation.

6.1. Valid State Testing

Table 6.1 shows the results of this valid state testing, with the EML passing each test. The test criteria are based on a visual assessment of the pattern made by the deflected particles; often directly comparing a test's *Observation Figure* to Figure 6.1. Each test aims to isolate a different assumption about how the system operates, and verify whether that property works as expected.

Test #	Test Description and Purpose	Expected Output and Reasoning	Observation Figure	Pass/ Fail
1	*Charge* is set to {0, 0}. Verifies that neutral charge (0) produces no deflection.	Particles continue on their horizontal trajectory. Particles have no charge, so the Lorentz force should be scaled to 0.		Pass
2	*Charg*e is set to {-1, -1} and then {1, 1}. Verifies that flipping *charge* values results in the particle trajectory also being flipped. Also verifies that the particles move in the direction predicted by the Lorentz force. Note: the camera angle has been moved to be opposite the dipole relative to the particle system, so the result can be more easily visualised. The baseline system (with *charges* {1, 1}; {-1, -1}) from the same perspective has also been provided for comparison.	Particles follow symmetrical trajectories relative to each other when viewed along the cross-section of the plane of emission. Particles are either +1 charge or -1 charge. A change in charge flips the Lorentz force direction, resulting in an equal but opposite trajectory pattern between groups of charges. If Lorentz's Force law is applied correctly, negatively charged particles should deflect towards the left, and positively charged particles towards the right.	 Charge: {-1, -1}  *Charge:* {1, 1}  *Charge:* {1, 1}; {-1, -1}	Pass

3	*Charge* is set to {3, 3}, {-3, -3} Verifies that the magnitude of *charge* is proportional to the strength of deflection.	Particles will deflect at more extreme angles and the bow shock will form further away from the magnetic dipole relative to the baseline. Scaling charge also scales the magnitude of the Lorentz force.		Pass
4	*Strength* is set to 3. Verifies that *strength* has a positive correlation with deflection.	Particles will deflect at more extreme angles and the bow shock will form further away from the magnetic dipole relative to the baseline. Increasing strength increases the magnetic moment, and therefore the magnitude of the Lorentz force.		Pass
5	*Strength* is set to 0. Verifies that *strength* must be >0 for particles to deflect.	Particles will not deflect, and will remain travelling on their horizontal trajectory. A strength of 0 means no magnetic moment, and so no Lorentz force.		Pass

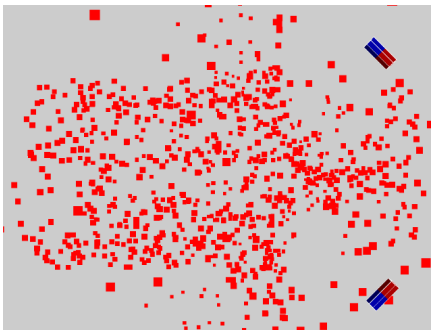
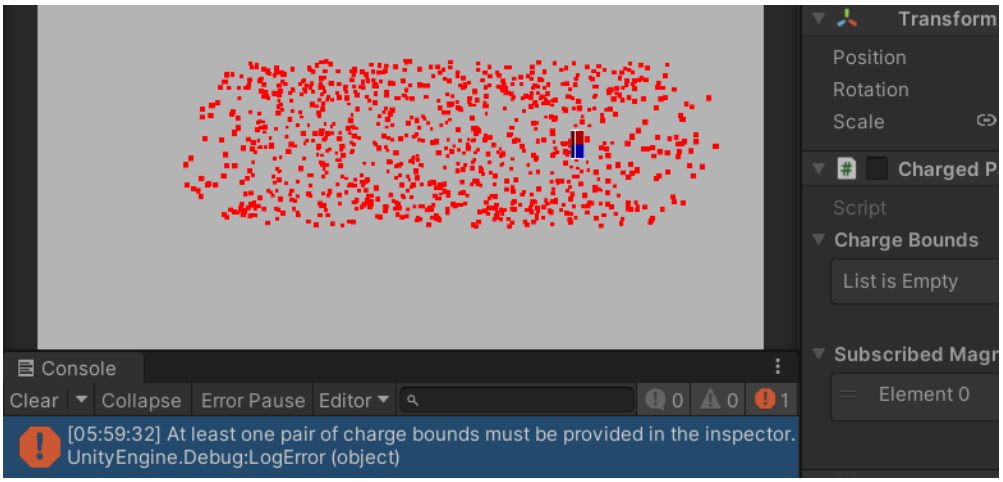
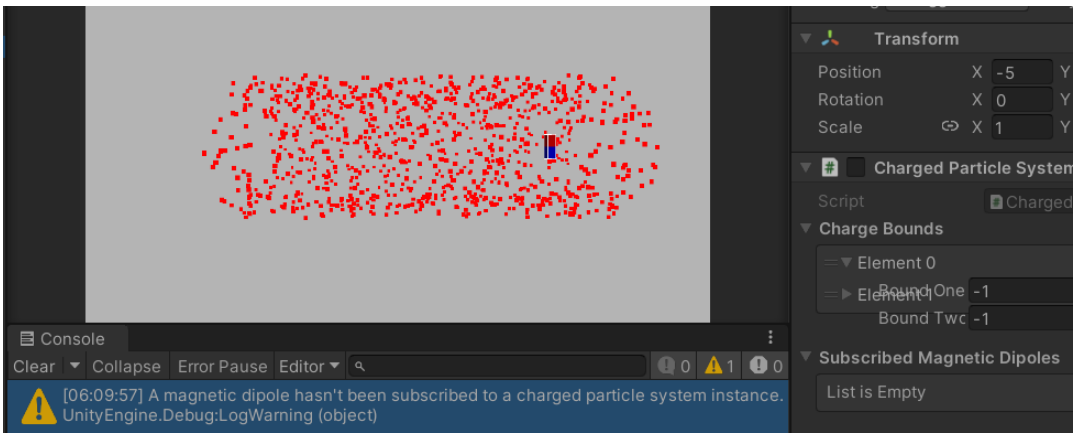
6	A second dipole of equivalent *strength* to the first is registered to the same Charged Particle System. Its position and orientation is symmetrical relative to the normal vector through the centre of the emission plane. Verifies that multiple dipoles can affect the same particle system, and in a predictable manner. Note: the transform of the initial magnetic dipole has been changed from the baseline system to achieve the symmetrical property with space between the magnetic dipoles.	Particles should form a bow shock around the second magnetic dipole, as well as the first. The pattern that the deflected particles make should be symmetrical. The symmetry of the dipole positioning and orientation should cause the deflections to be symmetrical about the GameObject plane of symmetry.		Pass

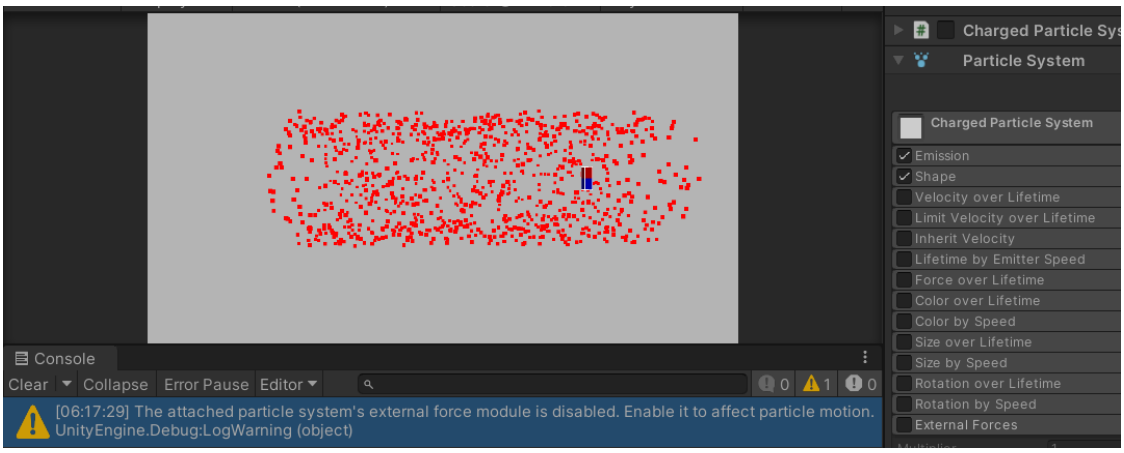
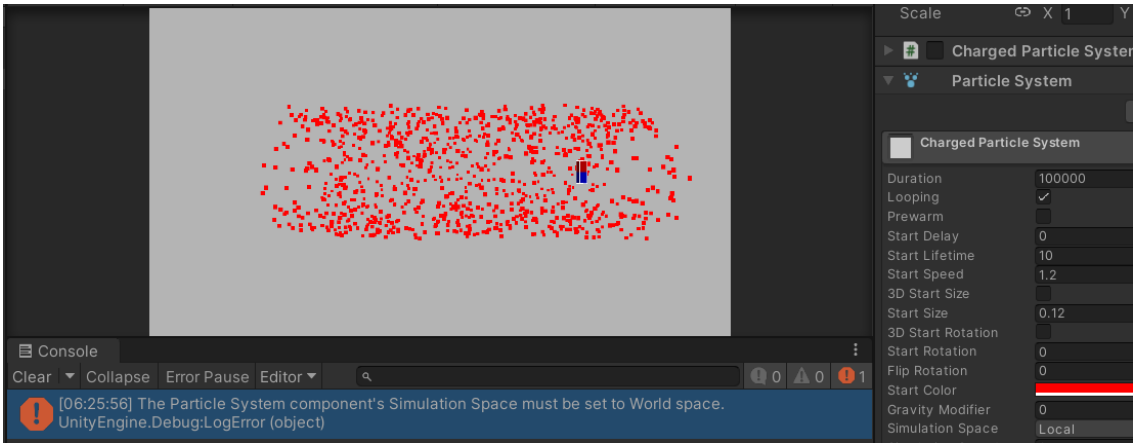
Table 6.1: Component property tests for valid system states.

6.2. Invalid State Testing

With the valid system states tested, the system is intentionally put into invalid states, and the EML is expected to handle the invalidity. Generally, the system provides feedback to the user about how the invalid state was entered, and what they can do to resolve the issue. A warning denotes an invalid state that may have been intended by the user and is recoverable, whereas an error denotes a state that is likely unwanted by the user. Table 6.2 shows that every test has been passed, meaning every invalid state is reproducible and guidance is provided to the user in the *Console* tab to improve the user experience with the EML.

Test #	**Test Description**	**Expected Outcome**	**Pass/ Fail**
1	Zero	The Charged Particle System component is disabled;	Pass

	ChargeBoundPair instances have been added to the *charge bounds* array.	particles are not deflected. A custom error message appears in the Unity *Console*.	
			
2	Zero Magnetic Dipole component instances have been added to the *subscribed magnetic dipole* array.	The Charged Particle System component is disabled; particles are not deflected. A custom warning message appears in the Unity *Console*.	Pass
			
3	The External Forces module of the *attached particle system* is disabled.	The Charged Particle System component is disabled; particles are not deflected. A custom warning message appears in the Unity *Console*.	Pass

			
4	The *attached particle system's simulation space* is set to 'Local', not 'World'.	The Charged Particle System component is disabled; particles are not deflected. A custom error message appears in the Unity *Console*.	Pass
			
5	The *strength* of the Magnetic Dipole is set to less than 0, to -1.	The *strength* member should be inverted (to 1) and the simulation should run. A custom warning message should appear in the Unity *Console*.	Pass

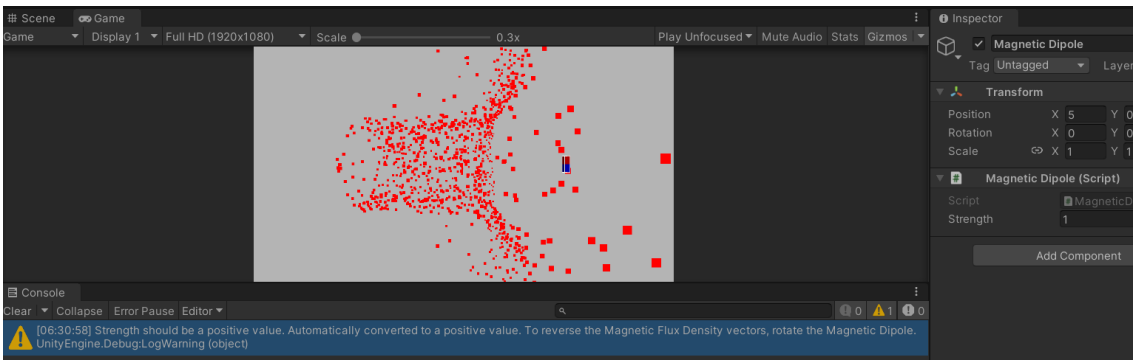


Table 6.2: Component property tests for invalid system states.

6.3. EML Applied to the Particle-Pinball Machine Application

In this section, the EML is applied to the PPM application introduced in section 2.1.1. Figure 2.1 shows the PPM scene without any deflection on the particles; the particles simply pass through the Earth object. The aim of this section is simply to show the setup of the EML components, and the resulting effect on the PPM: the evaluation of this section will be performed in Chapter 7.

A single Magnetic Dipole component has been added to the scene, attached to the 'Earth' GameObject. Similarly, a single Charged Particle System component has been added to the scene and attached to the 'Solar Ejection Particle System' component.

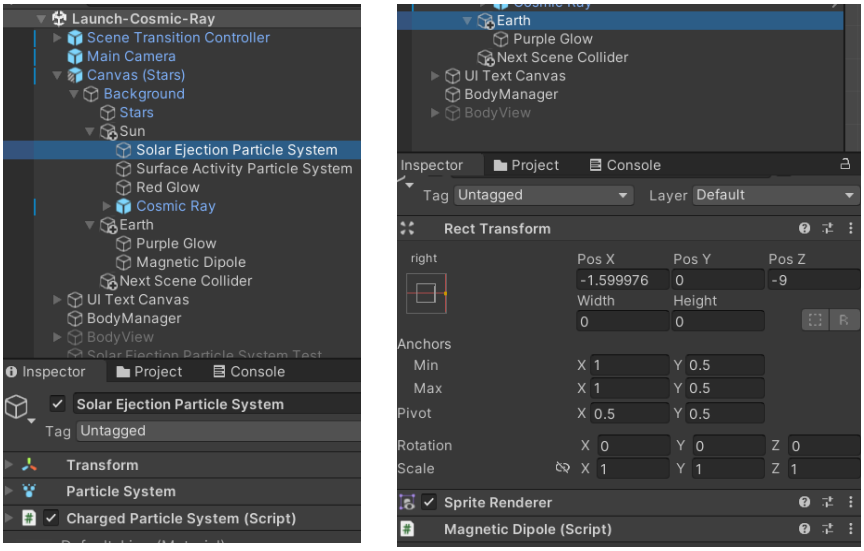


Figure 6.2: The *Hierarchy* tab of the PPM's first scene. The Charged Particle System component has been added to the 'Solar Ejection Particle System' GameObject and the Magnetic Dipole component has been added to the 'Earth' GameObject.

Next, the Magnetic Dipole and Charged Particle System component properties are set up. This first step involved subscribing the Magnetic Dipole component instance to the Charged Particle System instance.

Following on from this, the *charge* values were assigned to the Particles through the *charge bounds* property. A ChargeBoundPair instance with the values {1, 1} was added. Solar wind is composed of positively (+1) and negatively (-1) charged particles, and so another instance with {-1, -1} values could have been added. This was omitted, however, as it should allow for a slight performance increase, and adding a negative charge would not yield a different visual effect from the perspective of the Camera component.

The final adjustment to consider was the Magnetic Dipole *strength,* which was adjusted until the desired effect was achieved. The final *strength* value that best approximates the desired deflection being 0.17.

These adjustments as seen in the *Inspector* are shown in Figure 6.3. Adjustments could have been made inside the Particle System component to also alter the effect of the simulation, but found it not to be necessary to alter these values to achieve the desired effect.

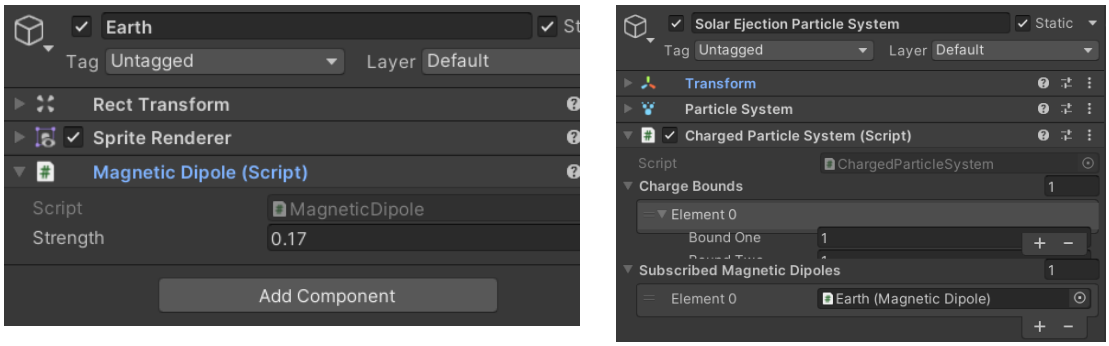


Figure 6.3: The values the EML components were initialised to in the *Inspector*.

These values result in the simulation shown in Figure 6.4, which shows the particles deflecting around the Earth forming a bow shock, instead of passing through it.

Figure 6.4: The first scene of the PPM after the EML has been applied. Particles deflect around the Earth forming a bow shock.

Chapter 7. Evaluation

In this chapter, the EML is evaluated in terms of its usability within the Unity GUI and accuracy at emulating the Lorentz force on charged particles. The evaluation of the latter, unfortunately, is limited due to a lack of expertise in the effects of the Lorentz force, but where possible the correctness of the simulation is assessed.

This lack of expertise is reflected in the rudimentary approach of testing the system in its valid states, whereby visual assessments were performed as opposed to quantitative methods. The testing approach showed that the component properties change the simulation behaviour as predicted, however it did not address to what extent the baseline simulation, and by extension every simulation made using the EML, resembles the Lorentz force.

Table 6.1 shows how component property values influence the simulation effect, and we can infer that the Lorentz force equation (Eq. 2.3) and the **B** equation (Eq. 2.4) are likely implemented correctly, as well as the process of assigning both *charge* and **B** values to Particles. This is most notably demonstrated in Test Number 2, whereby particles accelerate towards the direction predicted by the right-hand rule.

Through testing the simulation, notable observations of Particle behaviour have been identified, and go some way towards addressing how accurately the EML emulates the Lorentz force. One of the more obvious characteristics of the Particles is the bow shock that forms as

Particles approach the *magnetic dipole GameObject*, shown in Figure 6.1. Additionally, the formation of polar cusps may occur when particle velocities and magnetic dipole strength are appropriately balanced, as shown in Figure 7.1. When this occurs, Particles approach the North and the South of the dipole in a spiralling motion, before spiralling back and away from the magnetic dipole. The occurrence of these real-world observable phenomena indicates that the EML is likely a reasonably accurate emulation of the Lorentz force.

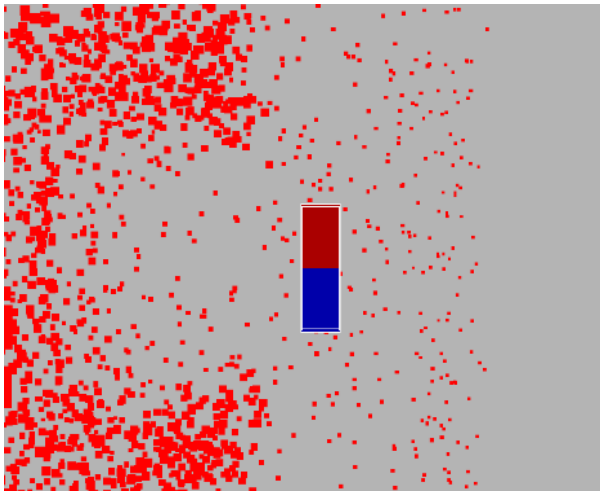


Figure 7.1: The Particles forming polar cusps at the North and South of the *magnetic dipole GameObject*.

The Particles that form the polar cusps are prone to increasing in speed and launching away from the magnetic dipole at high speed, which violates the principles of the Lorentz force, as discussed in section 2.2.1. Particles that do not get trapped in a polar cusp exhibit negligible changes in speed during runtime; so this issue is not detrimental.

From the perspective of a Unity developer, the performance and component design meet expectations. The design of the input systems feels appropriate for the properties they represent and of an equally high standard to the built-in systems. The testing process has made it apparent that often the lower bounds and upper bounds for a particular *charge bounds* element will be the same value, which can become tedious to update if done often enough. However, the benefit of having a system that can flexibly spawn particles with arbitrarily large charge ranges outweighs this drawback, and a typical user will likely interact far more infrequently with the system.

The simulation created in section 6.3 was a success. The deflection pattern shown in Figure 6.4 strongly resembles the effect shown in Figure 2.2, and certainly produces a more accurate effect than any alternative approach highlighted in Table 2.1 could achieve. The particles can clearly be shown forming a bow shock in front of the Earth, with particles deflecting past the Earth above and below, which were key parts in achieving the desired effect.

Chapter 8. Conclusions

8.1. Review of Aims

The project has been defined by two simple aims that were set out at the beginning of this project in chapter 1. These aims were:

1. To develop a fully documented software library, compatible with the Unity game engine, that is capable of simulating the motion objects of arbitrary charge within a magnetic field of arbitrary strength and direction.
2. To test and verify the correctness of the solution by implementing it within a Unity project that requires the simulation of electromagnetic interaction.

Aim 1 has been met on all counts. The software library, consisting of two files, is fully completed for the purposes of this work, and documented throughout. It is compatible with the Unity game engine. The library has been written with robustness and usability in mind. Three particular qualities of the library show this: duplicates of the Magnetic Dipole and the Charged Particle System components cannot be applied to the same GameObject; 'tooltips' appear in the Unity Editor upon hovering over each component property in theInspector tab, and errors/warnings appear as appropriate so that users are made aware when the system's internal state is invalid.

The second part of aim 1 has also been met. Particle *charge*, Magnetic Dipole *Strength* and the *magnetic dipole GameObject* Transform, particularly the *position* and *rotation* properties, all influence the simulation. Testing has shown that each of these three properties can be modified and the simulation exhibits an expected response.

Aim 2 is also successfully met, but in a more nuanced way than aim 1. The 'Unity project that requires the simulation' is referring to the PPM. Testing has shown that the EML alone has successfully introduced Particle deflection, resembling Figure 2.2, and therefore this application of the EML is considered a success. The nuance comes from the aim implying that the test performed in section 6.3 evaluates the correctness of the system, whereas it only partially shows correctness.Actually, even with all the additional testing that was performed in section 6.1, it is still not knowable how accurately the system resembles the Lorentz force, and therefore the level of correctness that the system has. Despite this, the testing shows that the EML can strongly emulate the Lorentz force and the user inputs behave in a predictable manner.

8.2. Project Overview & Suggested Revisions

On the whole, this project has been very successful, with few problems arising throughout the process that threatened the success or integrity of the work. Both project aims were satisfied, with the PPM exhibiting desirable behaviour in terms of solar wind deflection after the EML solution has been applied. The most notable issue that arose was during

implementation, where Particles were discovered to be moving in the direction opposite to the one predicted by the Lorentz force. However, this was soon resolved after learning that the Unity coordinate system was left-handed, and therefore produces the opposite vector to the one that the Lorentz cross product produces. Another notable issue that still persists is the Particles being liable to change speed during the simulation, a flaw in the current implementation of the system. Before formally beginning this work, the goal was to create a solution specifically for 2D applications; the concept being that Particles would move only along the *xy* plane throughout the system's runtime. Soon after research began, however, I learned that this isn't possible without having an internal 3D representation of the world, as the Lorentz force (and magnetism in general) is intrinsically a 3D phenomenon.

The testing procedure was sufficient to verify that the input systems of the EML work as expected, and that the system achieves similar behaviour to the Lorentz Force, however a lack of expertise in this domain has resulted in the extent of the solution's correctness being unclear.

If the project could be repeated, there are three elements of the work in particular that would likely be done differently. Firstly, an expert in the domain of electromagnetism would be sought that could verify the correctness of the solution. Having the system behaviour being evaluated by an expert would allow the success of the project itself to be well-established, and it may reveal insights into how the solution could be implemented to improve the EML.

In addition, the current implementation of mapping the Lorentz force vector to a *velocity* would be scrutinised, and likely reworked. Currently, Particles have variable speed: which is particularly apparent when they approach too close to the Magnetic Dipole's *position*, or the Particles become caught in the polar cusp region of the magnetic field. A naive approach to this problem could be to simply retrieve the Particle's speed, and then apply that magnitude in the direction of the Lorentz force. The problem, however, is that the magnitude of *charge* and the External Forces multiplier would not affect the simulation. Therefore, a method of calculating an updated velocity direction from the Lorentz force would be ideal.

Finally, testing would have been performed to ensure the ChargeBoundPair API works as expected. Currently it is assumed that the functions perform as desired as the code complexity for its implementation is relatively straightforward, but for completeness, test cases should have been written for the API alongside the other test cases.

8.3. Future Work

Here, three suggestions are offered for future work that can be achieved to improve the EML. The first of these is work to rigorously benchmark the performance of the EML, to determine the best contexts for the system's use; to understand the limits of the system's capabilities, and to identify potential bottlenecks in the performance that can be mitigated or removed in future releases. In this work, using the EML has not caused noticeable performance degradation, even as the number of particles alive simultaneously has reached into the thousands. However, the system, particularly the system used in the testing process of this work, had low resolution Particles, which may not make that simulation representative of all simulations.

Although the EML could produce the desired effect for the purposes of the PPM, it is clear that the simulations are tightly bound to rigid equations, and thus the simulations that can be produced could be too limited for a user's needs. Future work could experiment with 'breaking' the physics of the simulations by allowing users to insert their own custom properties into the equations to change the simulation behaviour beyond how the force naturally works. In this way, it may be possible to produce a more flexible, customisable and perhaps even user-friendly solution, whereby the shape of particle deflection is more predictable and controllable by the user.

The interaction of magnetic fields causes an exertion of force between the sources of the magnetic field. The current work could be extended by attempting to model the magnetic interaction between Magnetic Dipoles. If work could achieve this, it would further justify the EML as potentially being useful as an educational tool. Further, it could allow for complex and interesting reactions to occur, if both Charged Particle Systems and multiple Magnetic Dipole instances are simultaneously placed within a Unity scene.

8.4. Reflection

This section includes a brief summary of what has been learned through the project implementation process, in terms of the information learned in order to complete the work, as well as what has been learned through completing the work. To finish, a brief sentiment on the overall project.

With a background in computer science, completing a project that has strong interdisciplinary links with another subject, in this case physics, was a significant challenge. All of the electromagnetism content in this work had to be researched specifically for this work. This also includes content regarding the interaction between the Earth and solar weather. Skills with programming in C# had to be developed. This included learning C# documentation standards; continuing to develop programming fluency in C#, and writing software specifically for the Unity game engine.

The process of completing this work has provided great insight into the researching process - particularly as it applies to implementing a novel solution. As discussed in section 8.2, the original assumptions about the work heading into the project (regarding wanting to represent the Lorentz force in a 2D plane) were wrong, which led to a period of uncertainty. At this early period, it felt like the prospects of the project being successful were low. However, this work has reaffirmed that by showing perseverance and remaining level-headed, a solution to the particular problem at hand will more than likely make itself apparent.

To conclude, the work completed in this project has shown, despite uncertainty over the correctness of the system, that the project is an overall success. While this work is currently limited in its application, there is potential for future work to improve the system and potentially establish the EML as a fundamental tool in the development of physics-based Unity applications.

References

Borovsky, J. E. & Valdivia, J. A. (2018) The Earth's Magnetosphere: A Systems Science Overview and Assessment. Surveys in geophysics, 39(5), 817-859. 10.1007/s10712-018-9487-x.

Camacho, J. M. & Sosa, V. (2013) Alternative method to calculate the magnetic field of permanent magnets with azimuthal symmetry. Revista mexicana de física E, 59, 8-17.

Chow, T. (2006) Introduction to electromagnetic theory. Sudbury, Massachusetts: Jones and Bartlett Publishers.

Corbò, G. & Testa, M. (2009) Magnetic dipoles and electric currents. American Journal of Physics, 77(9), 818-820. 10.1119/1.3139186.

Engel, R., Heck, D. & Pierog, T. (2011) Extensive Air Showers and Hadronic Interactions at High Energy. Annual Review of Nuclear and Particle Science, 61(1), 467-489. 10.1146/annurev.nucl.012809.104544.

Kotova, G., Verigin, M., Gombosi, T., Kabin, K., Slavin, J. & Bezrukikh, V. (2021) Physics-Based Analytical Model of the Planetary Bow Shock Position and Shape. Journal of Geophysical Research: Space Physics, 126(6), e2021JA029104. https://doi.org/10.1029/2021JA029104.

Lorentz, H. (1895) Versuch einer Theorie der electrischen und optischen Erscheinungen in bewegten Körpern. Leiden: Brill.

Lyre, H. (1996) Multiple quantization and the concept of information. International Journal of Theoretical Physics, 35(11), 2263-2269. 10.1007/BF02302445.

Malandraki, O. E. & Crosby, N. B. (2018) Solar Energetic Particles and Space Weather: Science and Applications. In: Malandraki, O. E. & Crosby, N. B. (eds.) Solar Particle Radiation Storms Forecasting and Analysis: The HESPERIA HORIZON 2020 Project and Beyond. Cham: Springer International Publishing.

Mandea, M. & Chambodut, A. (2020) Geomagnetic Field Processes and Their Implications for Space Weather. Surveys in Geophysics, 41(6), 1611-1627. 10.1007/s10712-020-09598-1.

Nowicki, M. & Szewczyk, R. (2019) Determination of the Location and Magnetic Moment of Ferromagnetic Objects Based on the Analysis of Magnetovision Measurements. Sensors (Basel, Switzerland), 19(2), 337. 10.3390/s19020337.

Oulasvirta, A., Dayama, N. R., Shiripour, M., John, M. & Karrenbauer, A. (2020) Combinatorial Optimization of Graphical User Interface Designs. Proceedings of the IEEE, 108(3), 434-464. 10.1109/JPROC.2020.2969687.

Pitout, F. & Bogdanova, Y. V. (2021) The Polar Cusp Seen by Cluster. Journal of Geophysical Research: Space Physics, 126(9), e2021JA029582. https://doi.org/10.1029/2021JA029582.

Pivarski, J. (2012) Magnetodynamics. [ebook] Available at: http://coffeeshopphysics.com/magnetodynamics/. [Accessed 07/05/22].

Pshchelko, N. & Vodkailo, E. (2020) Features of Electrostatic Fields and Their Force Action When Using Micro- and Nanosized Inter-Electrode Gaps. Materials (Basel, Switzerland), 13(24), 5669. 10.3390/ma13245669.

Reddy, M. (2011) API Design for C++. Elsevier Science.

Sridevi, S. (2014) User interface design. International Journal of Computer Science and Information Technology Research, 2(2), 415-426.

Suzuki, T. K. (2012) Solar wind and its evolution. Earth, Planets and Space, 64(2), 14. 10.5047/eps.2011.04.012.

Tanel, Z. (2008) Students' difficulties in understanding the concepts of magnetic field strength, magnetic flux density and magnetization. Latin-American Journal of Physics Education, 2(3), 4.

TheNumber5. (2021) the-lorentz-force. GitHub repository. [Online] Available at: https://github.com/TheNumber5/lorentz-force-unity [Accessed 08/06/22].

Toftedahl, M. & Engström, H. (2019). A Taxonomy of Game Engines and the Tools that Drive the Industry. Unpublished paper presented at the DiGRA 2019, The 12th Digital Games Research Association Conference, Kyoto, Japan, August, 6-10, 2019, 2019.

Unity (2021a) Unity Manual - Scripting - Important Classes - GameObject Available at: https://docs.unity3d.com/Manual/class-GameObject.html [Accessed 08/06/22].

Unity (2021b) Unity Manual - Scripting - Important Classes - Gizmos. Available at: https://docs.unity3d.com/Manual/GizmosAndHandles.html [Accessed 08/06/22].

Unity (2021c) Unity Manual - Working in Unity - Create Gameplay - GameObjects - Introduction to Components. Available at: https://docs.unity3d.com/Manual/Components.html [Accessed 08/06/22].

Unity (2021d) Unity - Scripting API - Classes - Vector3. Available at: https://docs.unity3d.com/ScriptReference/Vector3.html [Accessed 08/06/22].

Unity (2021e) Unity - Scripting API - Classes - Vector4. Available at: https://docs.unity3d.com/ScriptReference/Vector4.html [Accessed 08/06/22].

Unity Technologies (2005) Unity (Version 2021.3.3f1) [Computer program]. Available at: https://unity3d.com/get-unity/download [Accessed 08/06/22].

Zhang, E., Mischaikow, K. & Turk, G. (2006) Vector field design on surfaces. ACM Trans. Graph., 25(4), 1294–1326. 10.1145/1183287.1183290.

Appendix A

The following link directs individuals from Lancaster University to a repository, containing:

- EML source code
    - MagneticDipole.cs
    - ChargedParticleSystem.cs
- The Unity Package file (.unitypackage) for a simplified importing process of the EML into a Unity project.
    - EML.unitypackage

https://livelancsac-my.sharepoint.com/:f:/g/personal/bradlejp_lancaster_ac_uk/Et0M7uK95gxGm-CENISjmCAB7tp6gDU0fPdUAtCw6X-sZA?e=Z4wMKM