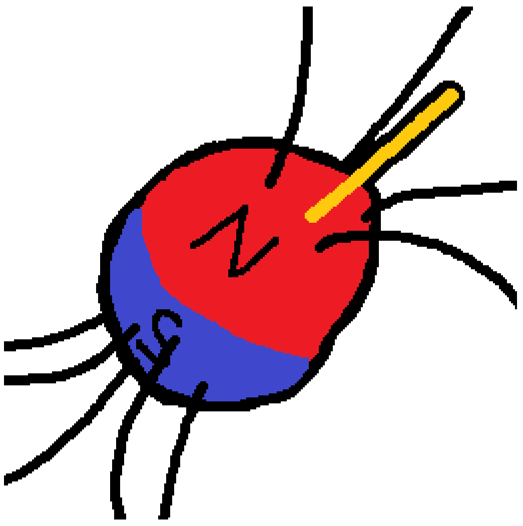


Visualising Magnetic Fields:

A 3D DYNAMIC FIELD SIMULATION

50.017 GRAPHICS AND VISUALISATION

Paul Ong Zesheng 1006025
Seah Ying Xiang 1005966
Loy Pek Yong 1004475

23/04/2025

1. Introduction

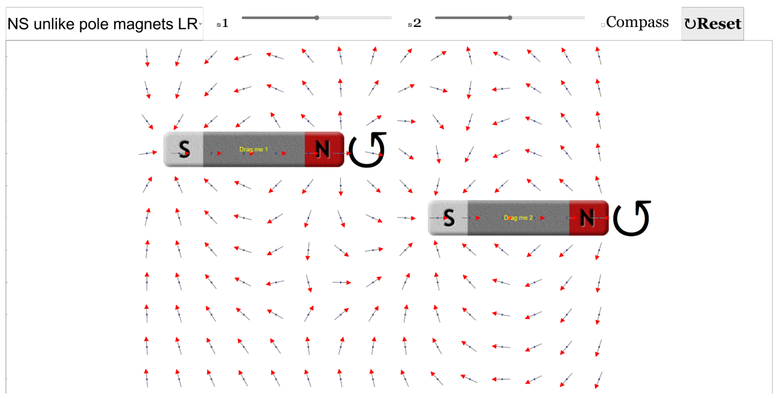
1.1 Problem Scope

Understanding magnetic fields is a fundamental aspect of physics education and plays a crucial role in many areas of engineering design. However, magnetic fields are inherently invisible and abstract, making them challenging to visualize and comprehend—especially for students or individuals without a strong technical background. Our goal is to provide users with a tool that bridges the gap between theoretical understanding and experiential learning through direct manipulation and real-time visual feedback.

1.1. Current implementations

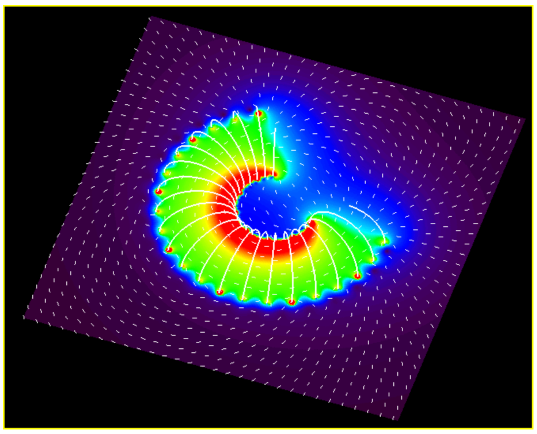
In the early stages of development, we explored a variety of existing field simulation tools to better understand current capabilities and limitations.

JavaScript Magnetic Field Simulation (Wee)



This simulation allows users to interact with magnetic bars by moving and rotating them. While it supports basic interactivity, the UI suffers from poor usability—especially in manipulating the rotation handles—and the visual presentation is relatively dated.

Static Visual Field Map (Boris)



Another reference provided striking visuals with well-defined field lines and a color gradient representing field strength. However, this implementation was entirely static, lacking any form of interactivity or real-time feedback. It serves well for illustration but falls short as a tool for exploratory learning.

1.3 Problem Statement

While several online magnetic field visualizers exist, they often fall short in one of two key areas: interactivity or visual clarity. Many simulations provide either static images with well-defined field lines or interactive elements with limited aesthetic appeal and usability. Moreover, these tools rarely support complex scenarios involving multiple magnets with adjustable orientations, strengths, and spatial configurations. As a result, users are often unable to engage in meaningful, hands-on exploration of magnetic phenomena

To address this gap, we aim to develop a platform that **visualizes magnetic field lines in 2D/3D**, simulates **realistic physical interactions between magnets in real time**, thus offering an engaging, hands-on way to experiment with and better understand magnetic behavior.

2. Approach

2.1. Physics

The physics approach in *magnetic-field-gl* focuses on simulating the magnetic field generated by multiple magnetic dipoles in 3D space. The simulation uses analytical models based on the dipole field equation, along with numerical methods that support real-time visualization and interaction.

Magnetic Field Model

The magnetic field is modeled using the classical magnetic dipole approximation. Each dipole has a position, direction, and magnetic moment. The field at any point in space is computed using the standard dipole field equation:

$$B(r) = \frac{\mu_0}{4\Pi}\frac{1}{r^3}(3(m * \hat{r})\hat{r} - m)$$

This model captures the spatial behavior of the magnetic field and supports accurate visualization across different components of the system.

Multi-Dipole Interaction

The system supports hundreds of dipoles simultaneously. The resulting magnetic field at any point is the superposition (vector sum) of the individual contributions from all dipoles. This enables the simulation of complex field configurations and interactions between dipoles.

Field Line Tracing

Magnetic field lines are traced using a fourth-order Runge-Kutta (RK4) integration method (Chen). Adaptive step sizing is applied to maintain accuracy in regions of rapidly changing field strength. This tracing starts from predefined seed points near each dipole and follows the field in both forward and backward directions.

Simulation Dynamics

Dipole motion is simulated by calculating forces and torques due to magnetic interactions. These are derived from potential energy gradients:

$$U = m_i * B_i \qquad\qquad \tau = m_i * B_i$$

Dipole positions and orientations are updated using Euler integration, while boundary conditions constrain them within a defined simulation volume.

2.2. Programming

The system is developed using modern C++ for simulation logic and OpenGL/GLSL for rendering. It integrates a range of libraries to support interactivity, performance, and modularity.

Core Framework

- **GLFW** is used for window and input management.
- **GLAD** handles OpenGL function loading.
- **Dear ImGui** provides a lightweight and interactive GUI.
- A central main loop handles real-time updates, including input handling, simulation stepping, and rendering.

Design Principles

The application adopts object-oriented design, organizing components like scene objects, dipoles, field line tracers, and cameras into distinct classes. This modularity allows for scalability and maintainability.

Multi-Threading

To improve performance, field line tracing is parallelized across multiple threads. This makes the system capable of handling large numbers of field lines in real time.

Interactivity

Users can interact with the simulation by:

- Selecting and manipulating dipoles using raycasting techniques.
- Navigating the 3D scene through camera controls (pan, rotate, zoom).
- Adjusting simulation parameters and visualization settings via GUI controls.

2.3. Graphics

Rendering is handled through OpenGL and GLSL shaders, offering real-time interactive visualization of magnetic fields, dipoles, and field lines.

Visualization Pipeline

- A **field plane shader** visualizes field strength as color gradients using HSL-to-RGB mapping.
- **Field lines** are rendered as dynamic polylines that follow magnetic field vectors.
- **Dipoles** are represented visually using color-coded spheres (red for north, blue for south) and arrows indicating direction.
- A wireframe **cuboid** outlines the simulation boundary.

Camera System

The camera supports both perspective and orthographic projections. This enables flexible exploration of the simulation space.

Optimization & Performance

- Efficient use of GPU buffers and shaders minimizes overhead.
- Transparent rendering and depth testing are used to maintain visual clarity.
- Real-time updates to shaders and geometry allow for immediate visual feedback.

Interactive Feedback

- Changes to dipole positions or properties update the visualizations in real time.
- GUI elements let users tweak visualization parameters such as field sensitivity or opacity.
- Labels and tooltips dynamically appear to provide context to users during interactions.

3. Implementation

3.1. Transform Class

3.1.1. Objectives

The Transform class is designed to manage the spatial properties and hierarchical relationships of objects in a 3D environment, commonly used in game engines or 3D graphics applications. Its primary objectives are:

- **Position Management**: Handle local and world positions of objects, supporting both absolute and relative coordinate systems.

- **Rotation Management**: Manage local and world rotations using quaternions and Euler angles, enabling precise orientation control.
- **Hierarchical Structure**: Support a parent-child hierarchy to model complex object relationships, where child transforms inherit transformations from their parents.
- **Transformation Operations**: Provide methods for translating, rotating, and orienting objects in 3D space, including advanced operations like rotating around arbitrary points or axes.
- **Coordinate Transformations**: Transform points and directions between local and world coordinate systems.
- **Matrix Computations**: Compute and maintain transformation matrices for rendering or physics calculations.
- **Directional Queries**: Provide utility methods to query directional vectors (forward, right, up) in world space.

The class leverages the GLM library for vector, quaternion, and matrix operations, ensuring robust and efficient mathematical computations.

3.1.1. Functions

Function Name	Purpose
Position Management	
`setLocalPosition(glm::vec3 localPosition)`	Sets the local position relative to the parent and updates the world transform matrix.
`getLocalPosition()`	Returns the local position as a glm::vec3.
`setWorldPosition(glm::vec3 globalPosition)`	Sets the world position, converting to local coordinates if a parent exists, and updates the world transform matrix.
`getWorldPosition()`	Returns the world position by applying the world transform matrix to the origin.
Rotation Management	
`setLocalRotation(glm::quat localRotation)`	Sets the local rotation quaternion and updates the world transform matrix.
`getLocalRotation()`	Returns the local rotation quaternion.
`setWorldRotation(glm::quat globalRotation)`	Sets the world rotation, adjusting for parent rotation if applicable, and updates the world transform matrix.

`getWorldRotation()`	Returns the world rotation by combining local and parent rotations.
`setLocalRotationEuler(glm::vec3 localRotationEuler)`	Sets the local rotation using Euler angles (in degrees) and updates the world transform matrix.
`setWorldRotationEuler(glm::vec3 globalRotationEuler)`	Sets the world rotation using Euler angles, internally calling setWorldRotation.
Rotation Operations	
`rotateAround(const glm::vec3& worldPoint, const glm::vec3& axis, float angleDegrees)`	Rotates the transform around a specified world point and axis by the given angle, updating both position and rotation.
`rotateAxis(const glm::vec3& axis, float angleDegrees)`	Rotates the transform around a specified axis in local space by the given angle.
`lookAt(const glm::vec3& target, const glm::vec3& worldUp)`	Orients the transform to face a target point, using a specified up vector to resolve ambiguity, and sets the world rotation accordingly.
Translation	
`translate(const glm::vec3& offset)`	Moves the transform by a world-space offset by updating its world position.
`translateLocal(const glm::vec3& localOffset)`	Moves the transform by a local-space offset, transformed to world space using the current rotation.
`Coordinate Transformations`	
`transformDirection(const glm::vec3& direction)`	Transforms a direction vector from local to world space using the world rotation.
`inverseTransformDirection(const glm::vec3& worldDirection)`	Transforms a direction vector from world to local space using the inverse world rotation.
`transformPoint(const glm::vec3& point)`	Transforms a point from local to world space using the world transform matrix.
`inverseTransformPoint(const glm::vec3& worldPoint)`	Transforms a point from world to local space using the inverse world transform matrix.
Hierarchy Management	
`setParent(Transform* newParent)`	Sets a new parent for the transform, preserving world position and rotation, and updates the hierarchy.

`getParent()`	Returns the current parent transform.
`addChild(Transform* child)`	Adds a child transform to the hierarchy if not already present.
`removeChild(Transform* child)`	Removes a child transform from the hierarchy and clears its parent reference.
`getChildCount()`	Returns the number of child transforms.
`getChild(int index)`	Returns the child transform at the specified index, or nullptr if invalid.
Matrix Computations	
`getLocalTransformMatrix()`	Returns the local transformation matrix combining translation and rotation.
`getWorldTransformMatrix()`	Returns the world transformation matrix, which includes parent transformations.
`updateWorldTransformMatrix()`	Updates the world transform matrix based on local transform and parent's world transform, and propagates updates to children.
Directional Queries	
`getForward()`	Returns the world-space forward direction (-Z axis).
`getRight()`	Returns the world-space right direction (X axis).
`getUp()`	Returns the world-space up direction (Y axis).

3.2. Magnetic Dipole Class

3.2.1. Objectives

The **MagneticDipole** class represents the smallest unit of a magnet in the simulation. It is designed to allow precise creation and control of magnetic dipoles in the application, with configurable position, direction, and magnetic moment. The main objectives of the class are:

- Enable the creation of magnetic dipoles at arbitrary positions within the simulation space.
- Allow control over the dipole's orientation and strength (moment) to simulate various magnetic behaviors.

- Provide methods to compute the magnetic field generated by the dipole at any point in space.
- Generate trace points around the dipole to visualize the magnetic field lines dynamically.

The main attributes managed by the class are:

- **Position**: A 3D vector (glm::vec3) representing the spatial location of the dipole.
- **Direction**: A 3D vector indicating the orientation of the dipole's magnetic moment.
- **Moment**: A floating-point value representing the strength of the magnetic dipole.

Key functionalities include calculating the magnetic field at any given point and initializing starting points for tracing magnetic field lines.

3.2.2. Functions

Function Name	**Purpose**

`MagneticDipole(`
`const glm::vec3& position,`
`float moment,`
`Transform* parent,`
`pixelsPerMeter = 100.0f)`
	Constructs a magnetic dipole at a specified position with a given magnetic moment, using a default forward direction.

`MagneticDipole(`
`const glm::vec3& position,`
`const glm::vec3 &`
`initialDirection,`
`float moment,`
`Transform* parent`
`pixelsPerMeter = 100.0f)`
	Constructs a magnetic dipole at a specified position with a specific initial orientation and magnetic moment.

`void setMoment(float moment)`	Sets the magnitude of the magnetic moment for the dipole.
`float getMoment() const`	Retrieves the current magnetic moment value of the dipole.

`void setDirection(`
`const glm::vec3& direction)`
	Sets the orientation of the magnetic dipole based on a given direction vector. Adjusts the dipole's rotation accordingly.

`glm::vec3 getDirection() const`	Returns the current forward direction of the magnetic dipole.

`glm::vec3`
`calculateMagneticField(`
`const glm::vec3& pos) const`
	Calculates and returns the magnetic field vector at a specified position relative to the dipole, based on the dipole's properties.

`void initializeDipoleTracePoints()`	Initializes a set of points around the dipole that act as starting locations for tracing magnetic field lines. These points are arranged in a small circle perpendicular to the dipole's direction.

3.2.3. Formulas

The magnetic field **B** at a point **r** relative to a magnetic dipole **m** located at **r₀** is given by:

$$B(r) = \frac{\mu_0}{4\Pi}\frac{1}{r^3}(3(m * \hat{r})\hat{r} - m)$$

Where:

- **r** = (pos - r_0) is the relative position vector from the dipole to the field point
- **r̂** is the unit vector pointing from the dipole to the field point.
- **m** is the magnetic moment vector.
- **r** is the distance between the dipole and the point.
- **μ_0** is the permeability of free space (typically constant, omitted in visualization).

In this specific implementation:

- A radial field component B_r and a tangential component B_θ are computed separately:
    - Radial component magnitude:
      $$B_r = \frac{2m\,cos(\theta)}{r^3}$$
    - Tangential component magnitude:
      $$B_\theta = \frac{2m\,sin(\theta)}{r^3}$$

Where:

- **θ** is the angle between the dipole's direction and the radial vector.
- **m** is the magnetic moment magnitude.

3.3. Moving Field Strength Visualiser

3.3.1. Objectives

The objective of the moving field strength visualizer is to provide a 2d plane to showcase the magnetic field strength at that particular plane for the magnetic simulations. Specifically, the visualiser aims to:

Visualize Field Strength:

Display the magnetic field strength across a movable 2D plane within a 3D cuboid, enabling users to analyze field variations at specific z-positions.

Interactive Plane Adjustment:

Allow users to dynamically adjust the z-position of the 2D field plane to explore magnetic field strength at different depths within the simulation space.

Real-Time Dipole Interaction:

Support the addition, removal, and manipulation (position, rotation, and moment) of magnetic dipoles, with immediate updates to the field strength visualization on the 2D plane.

Configurable Visualization Parameters:

Provide controls for adjusting the opacity and sensitivity of the field plane to enhance visibility and highlight field strength variations effectively.

3.3.2. Functions

Function Name	**Purpose**
`FieldPlane::updateZPosition (float z, float cuboidDepth)`	Updates the z-position of the field plane, which visualizes the magnetic field strength at a specific plane, affecting the display of field strength across the plane.
`updateFieldLineGeometry(const std::vector<FieldLine>& fieldLines, unsigned int& field_line_VAO, unsigned int& field_line_VBO, unsigned int& field_line_EBO, std::vector<float>& vertices,`	Updates the vertex and index buffers for rendering magnetic field lines, reflecting changes in dipole positions or orientations in the field strength visualization.

`std::vector<unsigned int>& indices)`	

3.4. Field Line Tracer

3.4.1. Objectives

The FieldLineTracer class is designed to trace magnetic field lines in a 3D space generated by a collection of magnets. Its primary objectives are:

- **Field Line Tracing**: Compute and trace magnetic field lines from specified starting points, following the magnetic field direction using numerical integration.
- **Configurable Tracing**: Support customizable tracing parameters, including step size, maximum steps, adaptive stepping, and rendering options.
- **Bounds Management**: Ensure field lines are traced within a defined 3D cuboid boundary to prevent infinite or out-of-scope tracing.
- **Magnetic Field Computation**: Aggregate magnetic field contributions from multiple magnets at any given point in space.
- **Adaptive Step Control**: Optionally use adaptive step sizes based on field strength to improve accuracy in regions of varying field intensity.
- **Parallel Processing**: Utilize multi-threading to efficiently trace multiple field lines from different starting points.
- **Robust Integration**: Employ 4th-order Runge-Kutta integration for accurate field line tracing, ensuring reliable results even in complex magnetic fields.

The class is intended for applications in physics simulations, magnetic field visualization, or computational electromagnetism, leveraging GLM for vector operations and multi-threading for performance.

Function	Purpose
`FieldLineTracer(const std::vector<BaseMagnet*>& magnets, float bounds_width, float bounds_height, float bounds_depth, float step_size, int max_steps, float adaptive_min_step, float adaptive_max_step, float`	Initializes the tracer with a list of magnets, cuboid bounds, and tracing configuration parameters (step size, max steps, adaptive step settings, and rendering flag).

`adaptive_field_ref, bool use_adaptive_step, bool render_field_lines)`	
`updateBounds(float width, float height, float depth)`	Updates the cuboid bounds for tracing, defining the minimum and maximum coordinates of the tracing volume.
`setTraceConfig(float step_size, int max_steps, float adaptive_min_step, float adaptive_max_step, float adaptive_field_ref, bool use_adaptive_step, bool render_field_lines)`	Sets tracing parameters, including step size, maximum steps, adaptive step range, reference field strength, adaptive stepping flag, and rendering flag.
`calculateTotalField(const glm::vec3& pos)`	Computes the total magnetic field at a given position by summing contributions from all magnets.
`isWithinBounds(const glm::vec3& pos)`	Checks if a position lies within the defined cuboid bounds.
`calculateAdaptiveStepSize(const glm::vec3& field)`	Calculates an adaptive step size based on the magnetic field strength, scaling inversely with field magnitude and clamping between minimum and maximum step sizes.
`traceFieldLineFromPoint(const glm::vec3& startPos, TraceDirection direction, FieldLine& line)`	Traces a field line from a starting position in the specified direction (forward, backward, or both) using 4th-order Runge-Kutta integration, storing points and field values in the provided `FieldLine` object. Stops if the field is too weak, exceeds bounds, or reaches the maximum steps.
`traceFieldLines()`	Traces field lines from all starting points provided by magnets, supporting forward and/or backward tracing based on each point's direction. Uses multi-threading to process start points in parallel and returns a vector of `FieldLine` objects containing the traced lines.

3.5. Shaders

The application employs three shader programs to visualize magnetic fields: **field shader**, **field line shader**, and **cuboid shader**, each handling specific rendering tasks with vertex and fragment shaders. Below is a concise overview of their objectives and functions:

Field Shader

Renders a transparent 2D field plane showing magnetic field strength/direction at a specified z-position.

Functions:

- **Vertex Shader**: Transforms the quad’s four vertices using model, view, and projection matrices, passing world-space positions to the fragment shader.
- **Fragment Shader**: Computes magnetic field per fragment using dipole data from the UBO, applies sensitivity_scaled for intensity, and sets color/opacity (plane_opacity) with blending.

Key Uniforms: view, projection, num_dipoles, pixels_per_meter, resolution, plane_opacity, sensitivity_scaled.

Rendering: Draws a quad (6 indices, 2 triangles) via field_VAO, with blending (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA) and depth writes disabled.

Field Line Shader

Visualizes 3D magnetic field lines tracing field direction.

Functions:

- **Vertex Shader**: Transforms field line vertices (position + field vectors) using model, view, and projection matrices, passing field data for coloring.
- **Fragment Shader**: Colors lines based on field vectors if use_field_line_color is true, or uses a uniform color; applies sensitivity_scaled. Uses the same colour calculation as the Field Shader.

Key Uniforms: view, projection, model, sensitivity_scaled, use_color (int).

Rendering: Draw line segments (GL_LINES) via field_line_VAO, opaque, updated dynamically if render_field_lines is true.

Cuboid Shader

Draws the 3D cuboid’s edges to outline the simulation volume.

Functions:

- **Vertex Shader**: Transforms edge vertices (12 edges) using model (identity), view, and projection matrices.
- **Fragment Shader**: Applies a uniform cuboid_color to edges.

Key Uniforms: color, view, projection, model.

Rendering: Renders lines (GL_LINES, 24 indices) via cuboid_VAO, opaque with depth testing.

Dipole Visualizer Shader (Implied, uses cuboid_shader)

Renders dipoles as spheres/arrows showing position/orientation.

Functions:

- **Vertex Shader**: Transforms visualizer geometry using dipole's position/rotation and camera matrices.
- **Fragment Shader**: Applies uniform colors for different parts (e.g., sphere, arrow).

Rendering: Opaque rendering per DipoleVisualizer, ensuring correct scene placement.

3.6. User Interface

3.6.1. Objectives

The UI system in this project provides an intuitive and interactive way for users to control and adjust simulation parameters in real time. It is implemented using the Dear ImGui library (ocornut/imgui: Dear ImGui: Bloat-free Graphical User interface for C++ with minimal dependencies), which offers a fast, minimalistic GUI for OpenGL applications.

The main objectives of the UI are:

- Allow users to view and modify simulation parameters without needing to recompile the program.
- Provide real-time updates to important properties such as the magnetic dipole moment.
- Support future extensibility for adding more interactive controls and debugging tools easily.

The current UI is minimal but functional, designed to not interfere visually with the 3D scene while offering clear, direct control.

3.6.2. Functions and Elements

Instructions Panel

This section displays key interaction controls to the user:

- **Drag Anywhere** – Rotate the scene
- **Scroll** – Zoom in/out
- **CTRL + Drag** – Move objects or pan the camera
- **ALT + Drag** – Rotate selected objects
- **ESC** – Exit the application

Camera Settings

Users can control the camera behavior and reset it when needed:

- **Frames Per Second (FPS)** is shown for performance feedback.
- **Field of View (FOV)** can be adjusted via a slider.
- **Perspective Mode Toggle** allows switching between perspective and orthographic views.
- **Reset Camera Button** returns the camera to its default position, rotation, and view target.

Field Plane Settings

This section configures the visualization of the plane used for field line rendering:

- **Z Position Slider** adjusts the depth placement of the field plane.
- **Opacity Slider** controls plane transparency.
- **Sensitivity Slider** modifies how strongly field strength variations appear visually.

Field Line Settings

Users can customize how magnetic field lines are rendered:

- **Render Field Lines** – Toggle field line visibility.
- **Use Colored Field Lines** – Enable color-coding based on field strength.
- **Adaptive Step Size Toggle** – Enables variable step lengths when tracing field lines.

- **Trace Settings**:
  - **Step Size** – Base step increment for tracing lines.
  - **Max Steps** – Limits how long lines can grow.
  - **Adaptive Step Range** – Minimum and maximum values for adaptive steps.
  - **Reference Field Strength** – Used to normalize adaptive stepping.
- **Apply Trace Settings** – Applies the current settings to the tracer.
- **Retrace Field Lines** – Recomputes all field lines based on updated parameters.

Simulation Settings

This panel manages the dynamic simulation of magnetic dipoles:

- **Simulation Speed Slider** – Controls the time step scale.
- **Start/Stop Simulation Button** – Begins or pauses the simulation.
- **Reverse Time Toggle** – Reverses the direction of simulation time.
- **Step Forward Button** – Manually progresses the simulation by one frame.

Object Management Panel

Users can add, remove, and edit magnetic dipoles:

- **Add Dipole Button** – Adds a new magnetic dipole at the origin with default orientation.
- **Randomize Dipoles Button** – Randomly repositions and reorients all dipoles within a bounding volume.
- **Label Toggles**:
  - **Show Labels** – Display all dipole names.
  - **Show Labels on Hover** – Only show labels when hovered.

Per-Dipole Controls

For each dipole in the scene:

- **Dipole Header** – Identifies the dipole by index (e.g., Dipole 1, Dipole 2...).
- **Position Editor** – Adjusts the dipole's 3D world position.
- **Rotation Editor** – Allows orientation input via Euler angles (in degrees).
- **Moment Editor** – Changes the magnetic moment magnitude.
- **Remove Button** – Deletes the dipole, reordering and updating all relevant lists and buffers.

4. Results

We have produced a fully-functioning OpenGL application that can simulate magnetic field lines in real-time in 3D, achieving the main goal of our project. Images of the results are shown in the appendix.

5. Future Improvements

To enhance the functionality, usability, and scope of the magnetic field simulation, several improvements can be considered for future development:

Bar Magnet Simulation: Extend MagneticDipole to model bar magnets or arbitrary shapes using multiple dipoles for accurate field approximation. Add rendering with distinct north/south pole visualizations for intuitive display.

Improve “Simulation” Mode: The current simulation mode performs physics calculation with a variable time-step in sync with the framerate, which leads to inconsistent accuracy, especially when the framerate is low. Future improvements could involve creating a separate physics rendering loop that runs at a higher speed for accuracy, that has a fixed time-step.

6. Conclusion

The project has been incredibly fun to work with aside from debugging, and we have applied many concepts learnt throughout the term such as object transformations, camera projections, polygon meshing, and hierarchical modelling. Working through the implementation challenges helped reinforce these theoretical concepts in a practical context.

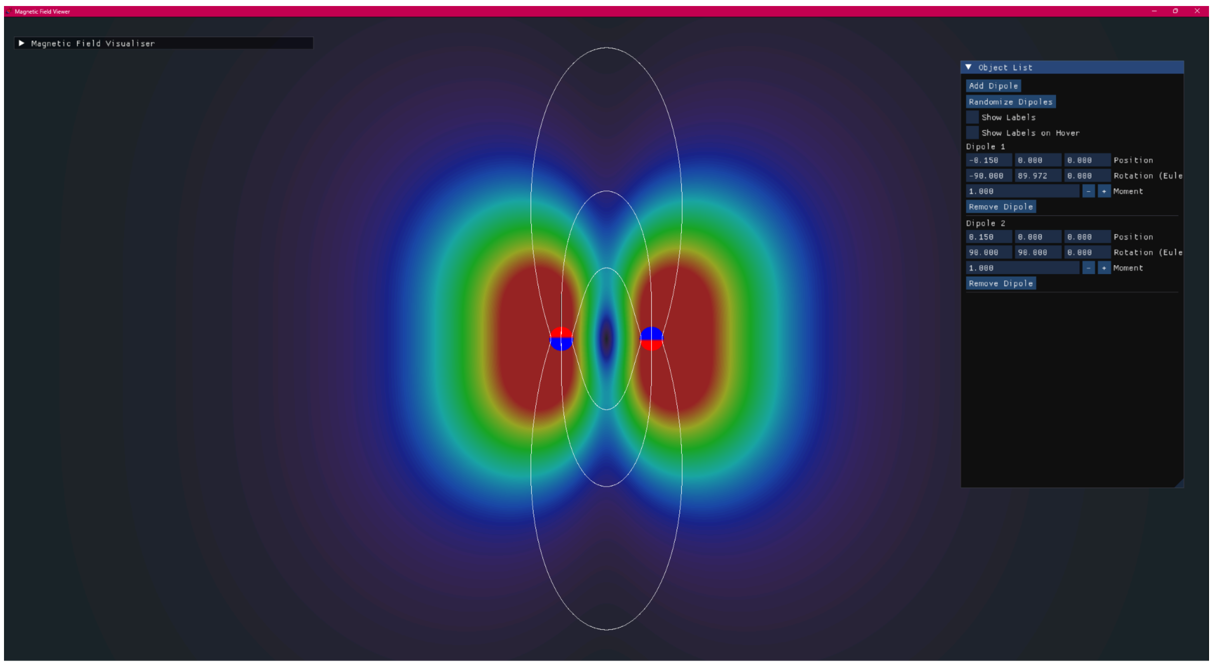
This project has taught us how to combine the concepts we have learnt to create a fully functioning, interactive magnetic field simulator in 3D, in addition to learning how OpenGL rendering works and how to interface with the GPU via the array and buffer objects. The process of setting up vertex array objects (VAOs), vertex buffer objects (VBOs), and element buffer objects (EBOs) gave us valuable hands-on experience with graphics programming fundamentals.

We've learnt how to design and build a 3D scene from scratch and render the scene, write custom shaders, and pass in complex attributes such as dipole data into the shaders for visualisation of field strength. This required us to think carefully about efficient data structures and memory management to achieve smooth performance.

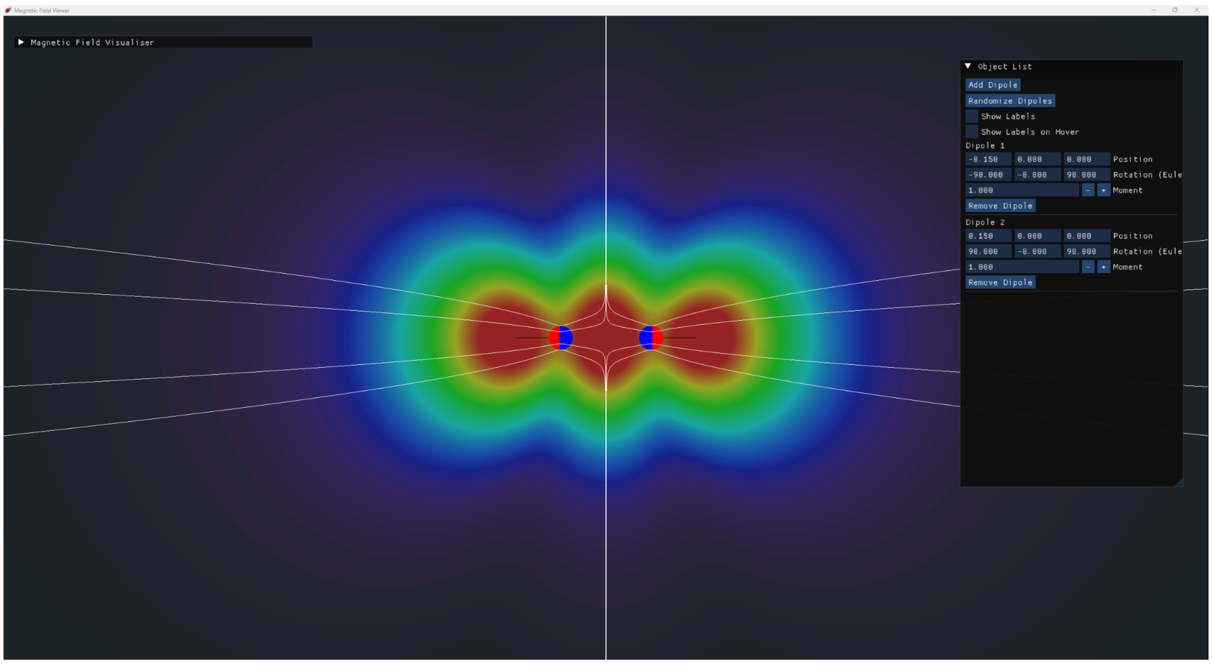
The interactive nature of our simulator presented additional challenges, as we needed to implement event handling systems that allowed users to manipulate dipoles while maintaining accurate physics calculations. Finding the right balance between computational efficiency and physical accuracy required several iterations of our code.

Overall, this project has demonstrated how the theoretical knowledge from our coursework translates into practical applications. The experience of building a complete visualization system has developed our technical problem-solving skills and given us confidence in our ability to tackle complex programming challenges in the future.

Appendix



```
Dipole 1
-0.150    0.000    0.000    Position
-90.000   89.972   0.000    Rotation (Eule
1.000                 - +   Moment

Dipole 2
0.150     0.000    0.000    Position
90.000    90.000   0.000    Rotation (Eule
1.000                 - +   Moment
```



Dipole 1

-0.150 0.000 0.000 Position

-90.000 -0.000 90.000 Rotation (Eule

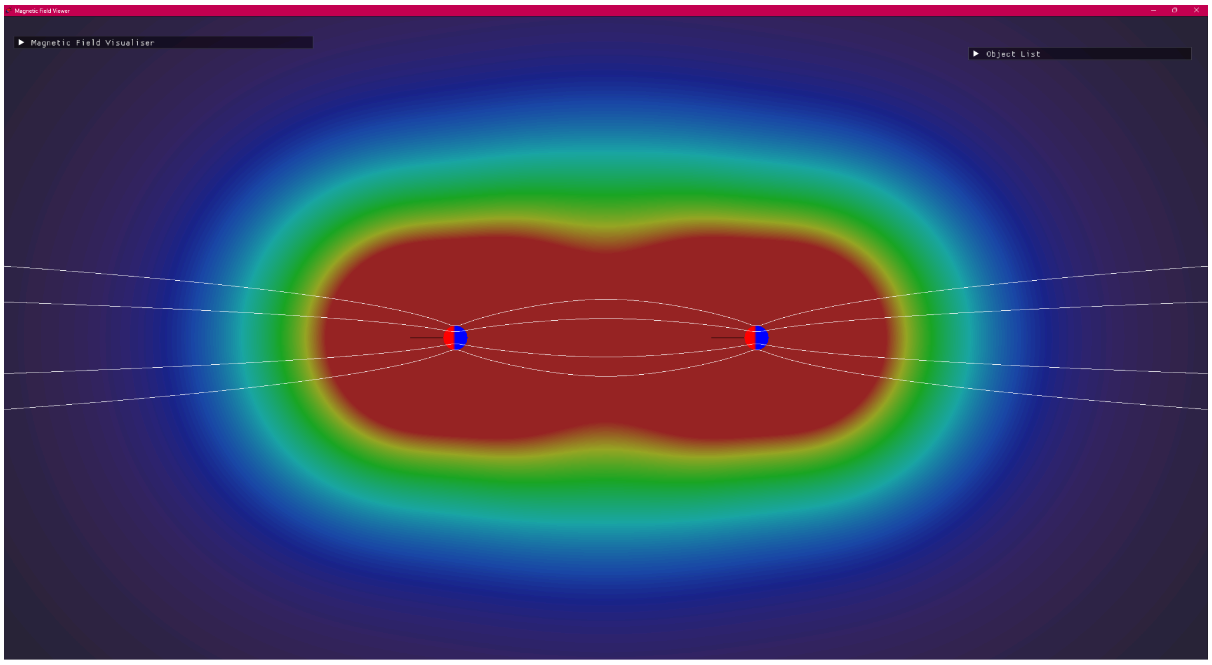
1.000 - + Moment

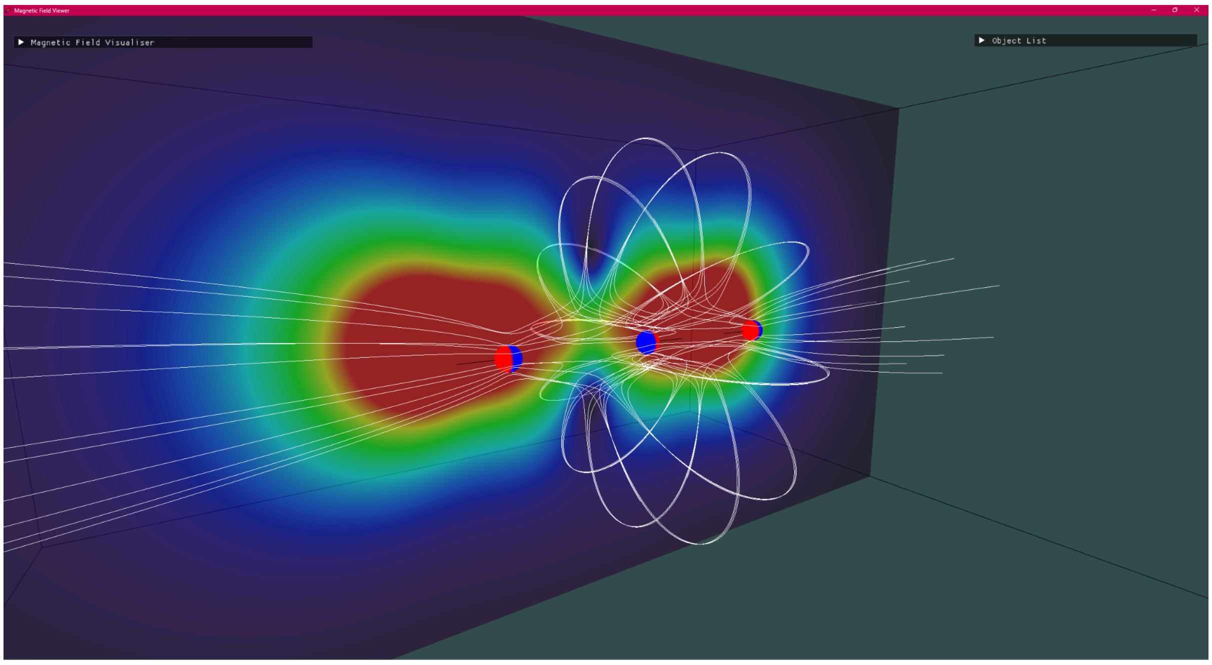
Dipole 2

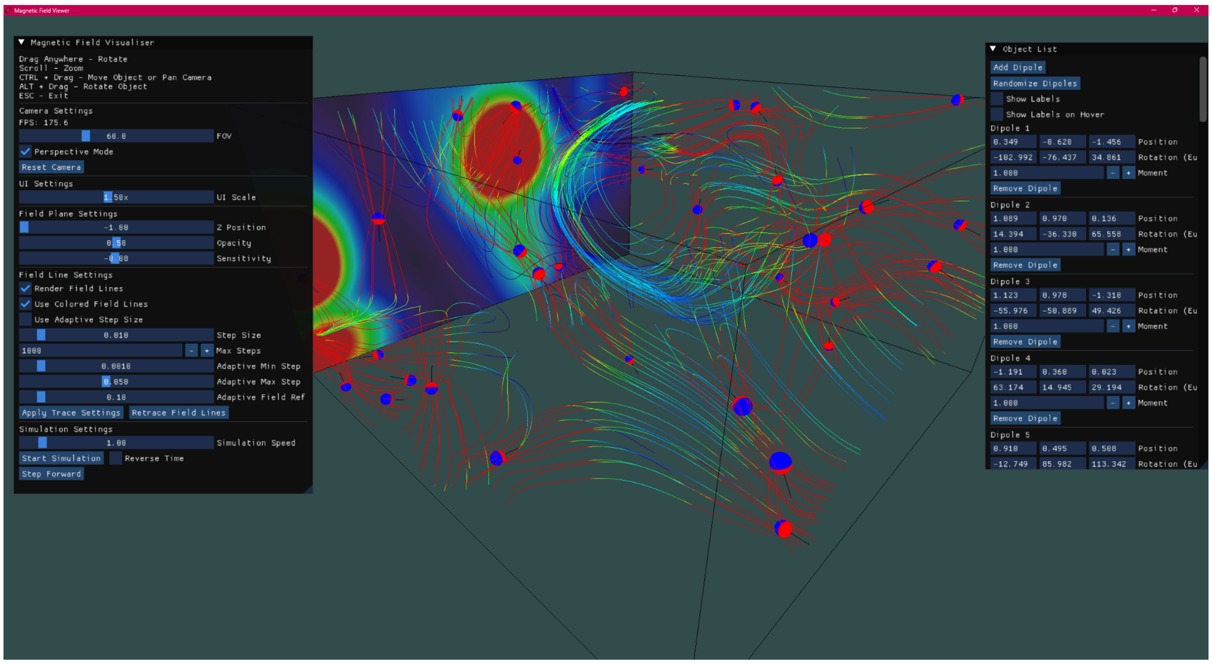
0.150 0.000 0.000 Position

90.000 -0.000 90.000 Rotation (Eule

1.000 - + Moment

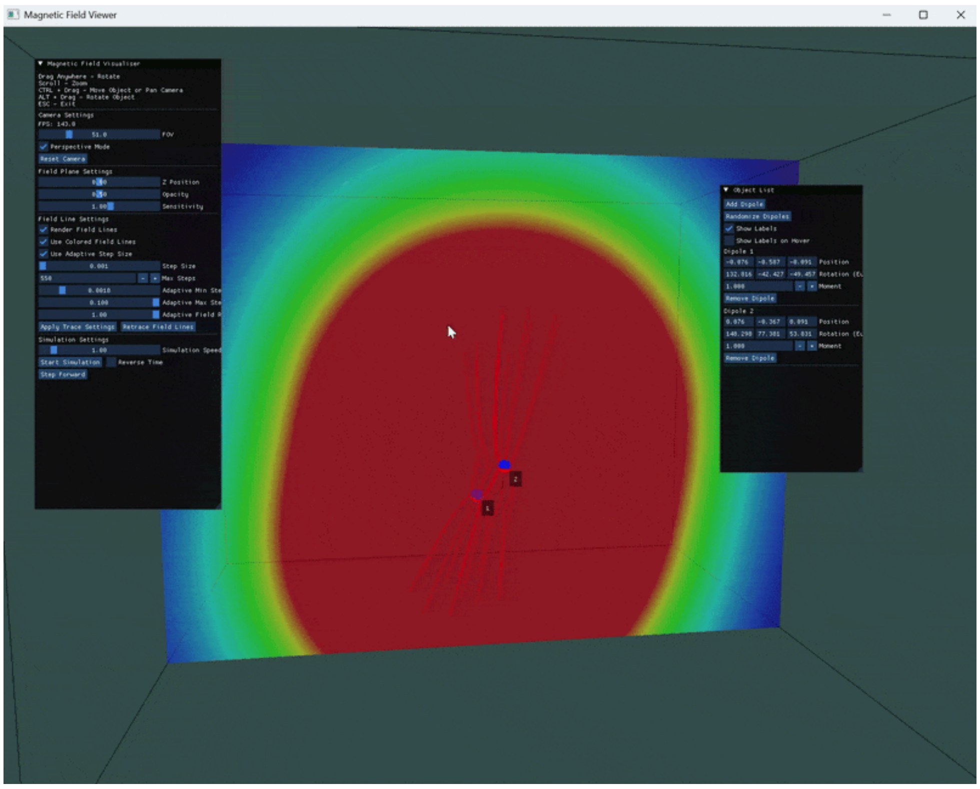




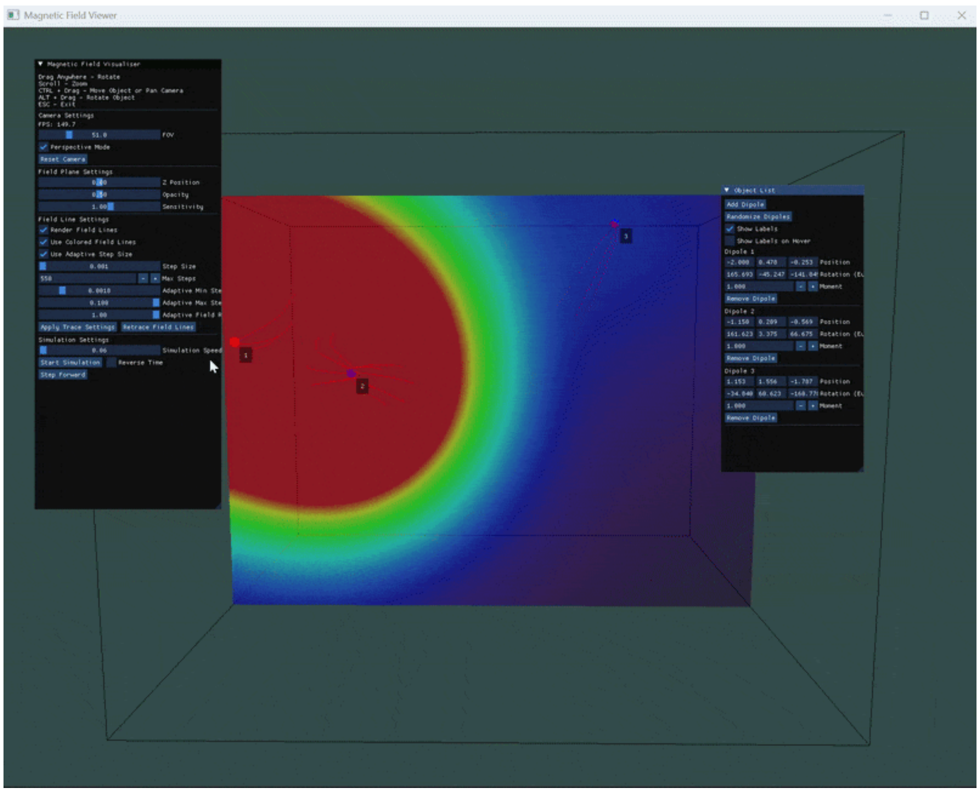


```
▼ Magnetic Field Visualiser
Drag Anywhere - Rotate
Scroll - Zoom
CTRL + Drag - Move Object or Pan Camera
ALT + Drag - Rotate Object
ESC - Exit
Camera Settings
FPS: 175.6
60.0                    FOV
[✓] Perspective Mode
Reset Camera
UI Settings
1.50x                   UI Scale
Field Plane Settings
-1.00                   Z Position
0.50                    Opacity
-0.08                   Sensitivity
Field Line Settings
[✓] Render Field Lines
[✓] Use Colored Field Lines
[ ] Use Adaptive Step Size
0.010                   Step Size
1000           - +      Max Steps
0.0010                  Adaptive Min Step
0.050                   Adaptive Max Step
0.10                    Adaptive Field Ref
Apply Trace Settings   Retrace Field Lines
Simulation Settings
1.00                    Simulation Speed
Start Simulation  [ ] Reverse Time
Step Forward
```

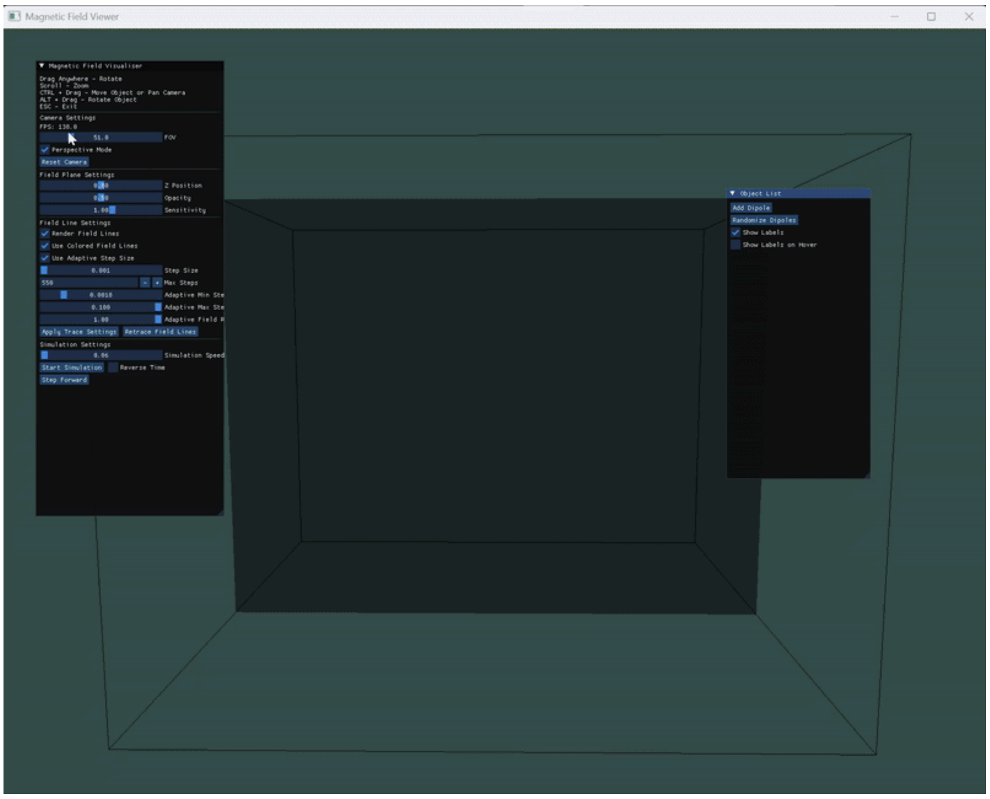
```
▼ Object List
Add Dipole
Randomize Dipoles
[ ] Show Labels
[ ] Show Labels on Hover
Dipole 1
0.349      -0.620     -1.456     Position
-102.992   -76.437    34.861     Rotation (Eu
1.000               - +          Moment
Remove Dipole
Dipole 2
1.089      0.970      0.136      Position
14.394     -36.330    65.558     Rotation (Eu
1.000               - +          Moment
Remove Dipole
Dipole 3
1.123      0.978      -1.310     Position
-55.976    -50.889    49.426     Rotation (Eu
1.000               - +          Moment
Remove Dipole
Dipole 4
-1.191     0.360      0.023      Position
63.174     14.945     29.194     Rotation (Eu
1.000               - +          Moment
Remove Dipole
Dipole 5
0.910      0.495      0.508      Position
-12.749    85.982     113.342    Rotation (Eu
```



```
Magnetic Field Viewer

▼ Magnetic Field Visualiser
Drag Anywhere - Rotate
Scroll - Zoom
CTRL + Drag - Move Object or Pan Camera
ALT + Drag - Rotate Object
ESC - Exit
Camera Settings
FPS: 143.0
51.0                 FOV
✓ Perspective Mode
Reset Camera
Field Plane Settings
0.00                 Z Position
0.50                 Opacity
1.00                 Sensitivity
Field Line Settings
✓ Render Field Lines
✓ Use Colored Field Lines
✓ Use Adaptive Step Size
0.001                Step Size
550          - +     Max Steps
0.0010               Adaptive Min Ste
0.100                Adaptive Max Ste
1.00                 Adaptive Field R
Apply Trace Settings   Retrace Field Lines
Simulation Settings
1.00                 Simulation Speed
Start Simulation   Reverse Time
Step Forward

▼ Object List
Add Dipole
Randomize Dipoles
✓ Show Labels
Show Labels on Hover
Dipole 1
-0.076   -0.587   -0.091   Position
132.816  -42.427  -49.457  Rotation (Eu
1.000           - +        Moment
Remove Dipole
Dipole 2
0.076    -0.367   0.091    Position
148.298  77.381   53.831   Rotation (Eu
1.000           - +        Moment
Remove Dipole
```



```
Magnetic Field Viewer

▼ Magnetic Field Visualiser
Drag Anywhere - Rotate
Scroll - Zoom
CTRL + Drag - Move Object or Pan Camera
ALT + Drag - Rotate Object
ESC - Exit
Camera Settings
FPS: 149.7
51.0                 FOV
✓ Perspective Mode
Reset Camera
Field Plane Settings
0.00                 Z Position
0.50                 Opacity
1.00                 Sensitivity
Field Line Settings
✓ Render Field Lines
✓ Use Colored Field Lines
✓ Use Adaptive Step Size
0.001                Step Size
550          - +     Max Steps
0.0010               Adaptive Min Ste
0.100                Adaptive Max Ste
1.00                 Adaptive Field R
Apply Trace Settings   Retrace Field Lines
Simulation Settings
0.06                 Simulation Speed
Start Simulation   Reverse Time
Step Forward

▼ Object List
Add Dipole
Randomize Dipoles
✓ Show Labels
Show Labels on Hover
Dipole 1
-2.000   0.478    -0.253   Position
165.693  -45.247  -141.84  Rotation (Eu
1.000           - +        Moment
Remove Dipole
Dipole 2
-1.150   0.209    -0.569   Position
161.623  3.375    66.675   Rotation (Eu
1.000           - +        Moment
Remove Dipole
Dipole 3
1.153    1.556    -1.787   Position
-34.040  60.623   -160.77  Rotation (Eu
1.000           - +        Moment
Remove Dipole
```



```
▼ Magnetic Field Visualiser
Drag Anywhere - Rotate
Scroll - Zoom
CTRL + Drag - Move Object or Pan Camera
ALT + Drag - Rotate Object
ESC - Exit
Camera Settings
FPS: 130.0
51.0                FOV
✓ Perspective Mode
Field Plane Settings
0.00                Z Position
0.50                Opacity
1.00                Sensitivity
Field Line Settings
✓ Render Field Lines
✓ Use Colored Field Lines
✓ Use Adaptive Step Size
0.001               Step Size
550          - +    Max Steps
0.0018              Adaptive Min Ste
0.100               Adaptive Max Ste
1.00                Adaptive Field R
Simulation Settings
0.06                Simulation Speed
☐ Reverse Time

✓ Show Labels
☐ Show Labels on Hover
```

Bibliography

Wee, Loo Kang. “🧲2 Magnetic Bars Field JavaScript HTML5 Applet Simulation Model.” Open Educational Resources / Open Source Physics at Singapore , 11 Nov. 2015, sg.iwant2study.org/ospsg/index.php/interactive-resources/physics/05-electricity-and-magnetism/07-magnetism/268-magneticbarfieldsecondmagnet06. Accessed 23 Apr. 2025.

Boris Spokoinyi. “Static Magnetic Field Simulator”. 2011, http://em_field.tripod.com. Accessed 23 Apr. 2025.

Chen Jun. “Numerical solution of tracing magnetic field lines.” University of Science and Technology of China.11 August 2020, https://space.ustc.edu.cn/users/1367579391JDEkeVpzVnBZT3QkL1FxL01GR3g3bkJBL3NsNUV2MTEwLw/blog/20131102030701.711/20200811193052.167/at/trace_bline.pdf. Accessed 23 Apr. 2025.