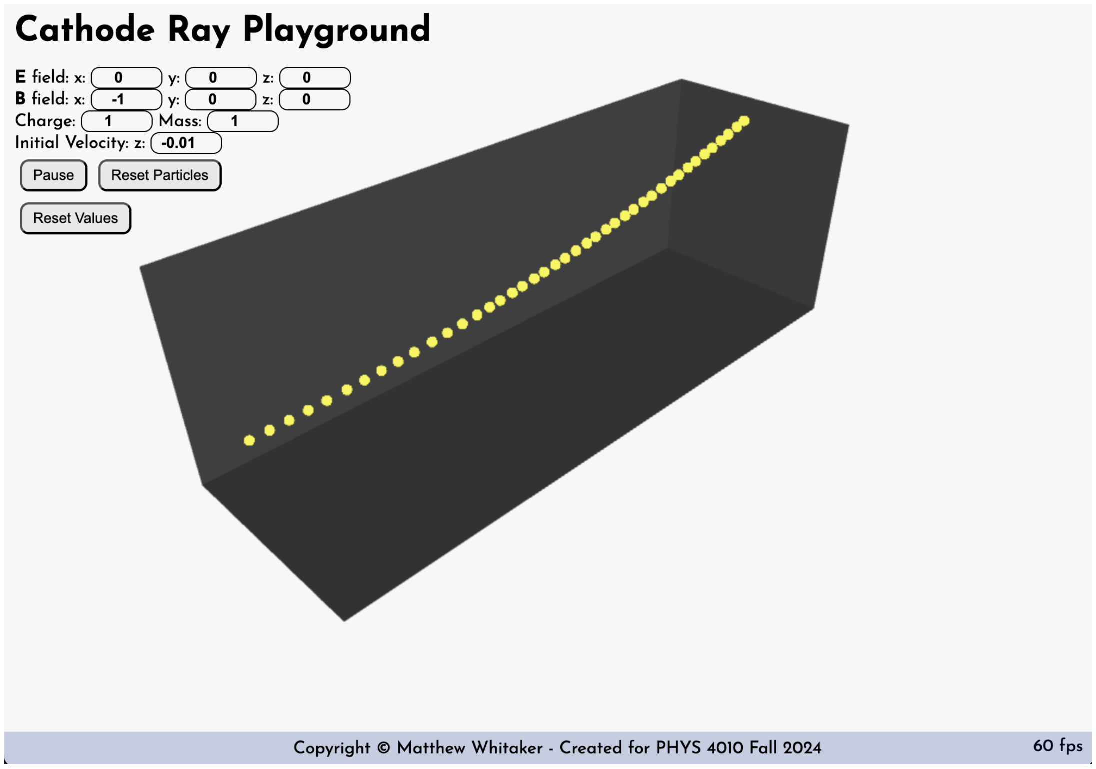
The Cathode Ray Playground

Simulate your very own cathode ray! This project is an interactive simulation of a beam of electric charges moving through uniform electric and/or magnetic fields. It runs in 3d in any modern web browser.

You can see it in action at https://em.matthewwhitaker.me/
(instructions on running from source code are included below)


Cathode Ray Playground

E field: x: 0 y: 0 z: 0
B field: x: -1 y: 0 z: 0
Charge: 1 Mass: 1
Initial Velocity: z: -0.01

Copyright © Matthew Whitaker - Created for PHYS 4010 Fall 2024

About The Project

Available options, physical descriptions, and assumptions

This is my project: the "Cathode Ray Playground". However, the name is a bit misleading, since cathode rays are traditionally streams of electrons, and my project allows you to simulate streams of particles with *whatever charge you want*! There are 42 total particles in the simulation, so if the stream appears to stop, it's probably that your particles are hiding away in some corner of the world. Hit "Reset Particles" at any time to bring particles back to their starting point. "Reset Values" resets the entire simulation, so it's a good starting point if your particles get out of hand.

The Physics: This simulation mainly visualizes the force equation:

$$\mathbf{F} = Q(\mathbf{E} + \mathbf{v} \times \mathbf{B}). \qquad \text{or equivalently} \qquad \mathbf{a} = Q(\mathbf{E} + \mathbf{v} \times \mathbf{B})/m$$

In this case, both the electric and magnetic fields being simulated are uniform across the entire simulated space, and the user is free to modify their magnitude and direction by editing the cartesian components directly using the input fields at the top left of the simulation.

In the same way, the user can also adjust intrinsic properties of the particles — their charge and mass. The initial z-component of velocity can also be configured. Note that the default is in the negative-z direction. More details about the coordinate system can be found below.

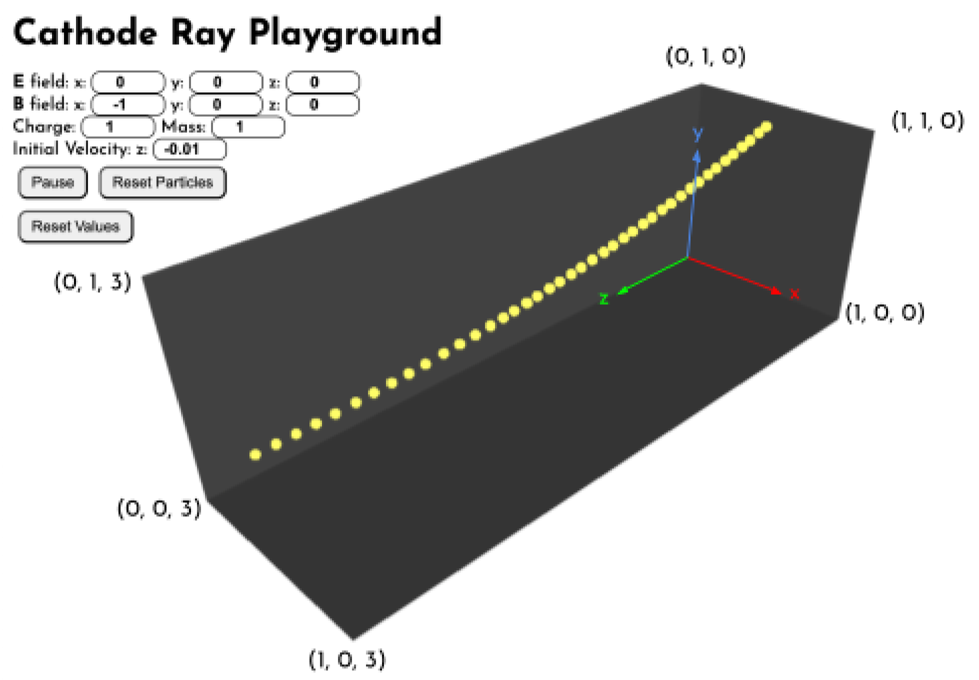
The entire simulation is in 3d, so you can fly around the space using the arrow keys on your keyboard (if you are on a computer). You can also rotate the camera using the QWEASD keys (the rotation matrices are set up a little weird, adding to the sense of adventure you may feel while navigating this electromagnetic world).

Physical Description: The units in this simulation are somewhat arbitrary (although the relationships between them preserve the original physics).

Time is measured in *frames*, with approximately 60 frames per second (fps). An indicator of the average fps over the last 10 frames is shown on the bottom right of the simulation. If your computer's graphics are running slow (or *lagging* as the cool kids say), less frames will pass in the same amount of time, so the simulation will appear to run slower. (Modern game engines take the fps into account when computing graphics such that even when the graphics are lagging, the simulated time still runs at the same rate. I've skipped that for this project to keep things simple).

Charge and **Mass** are measured in arbitrary units. 1 unit of charge probably corresponds most closely to the magnitude of the charge of one electron, and 1 unit of mass the mass of an electron.

Position is measured in world-distance. The coordinate system aligns with the visible grey faces: The square panel towards the back is the x-y plane, with the origin at the farthest point from the camera. The lower rectangular panel is the x-z plane, and the upper rectangular panel is the y-z plane. The simulation begins with the camera at the position (x, y, z) = (3, 2.5, 4), and the particles first enter the world at the position (x, y, z) = (0.5, 0.5, 3.0)


Cathode Ray Playground

E field: x: 0 y: 0 z: 0
B field: x: -1 y: 0 z: 0
Charge: 1 Mass: 1
Initial Velocity: z: -0.01

As this image shows, the space has a size of 1 x 1 x 3 world-distance units.

Velocity is measured in world-units per frame. As such, 1 is relatively fast. The default velocity, -0.01 seems to be a good velocity for the visualization.

The **Electric Field** has units of force per charge, which in our world ends up being mass-units * world-distance * frames^{-2} * charge-units^{-1}

Similarly, the **Magnetic Field** has units of mass per time squared per current, which ends up being mass-units * frames^{-1} * charge-units^{-1}

Assumptions/Caveats: There are a few caveats I should point out:

First: in this simulation the charges do not interact with the other charges. They move solely under the influence of the electric and magnetic fields.

Second: There are some numerical artifacts from floating point precision issues. More details about this are in the technical details below.

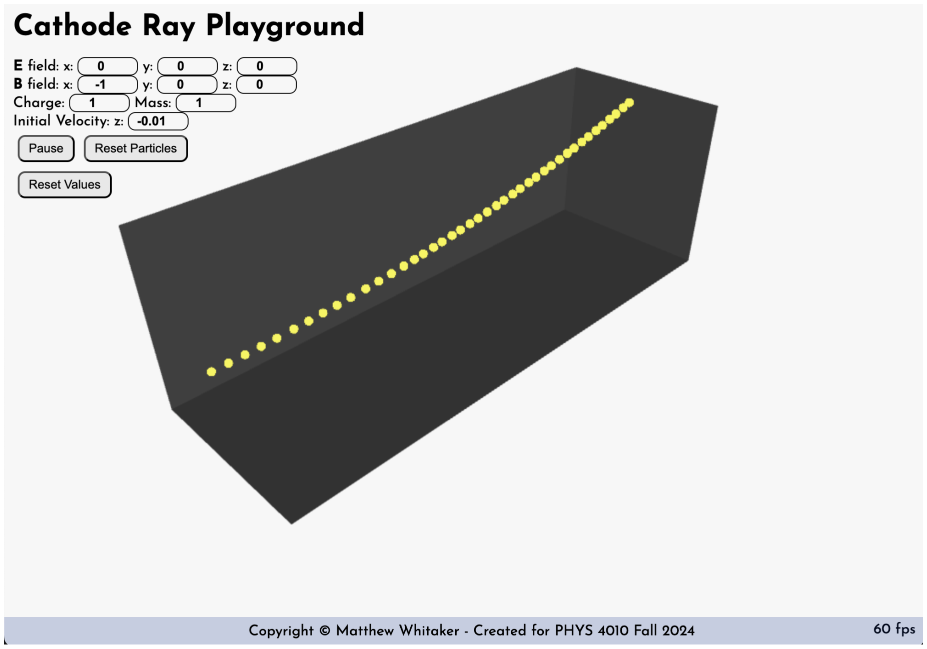
Third: the charges are not to scale. It turns out that electrons are very very small. This makes it difficult to render them to-scale. You can try if you'd like. If you are really concerned about this, you can pretend that the particles in this simulation are spherical charged bananas.

Some Examples

A few configurations you can try to see some interesting physical phenomena

The Deflected Ray

This is the result of using the default values:



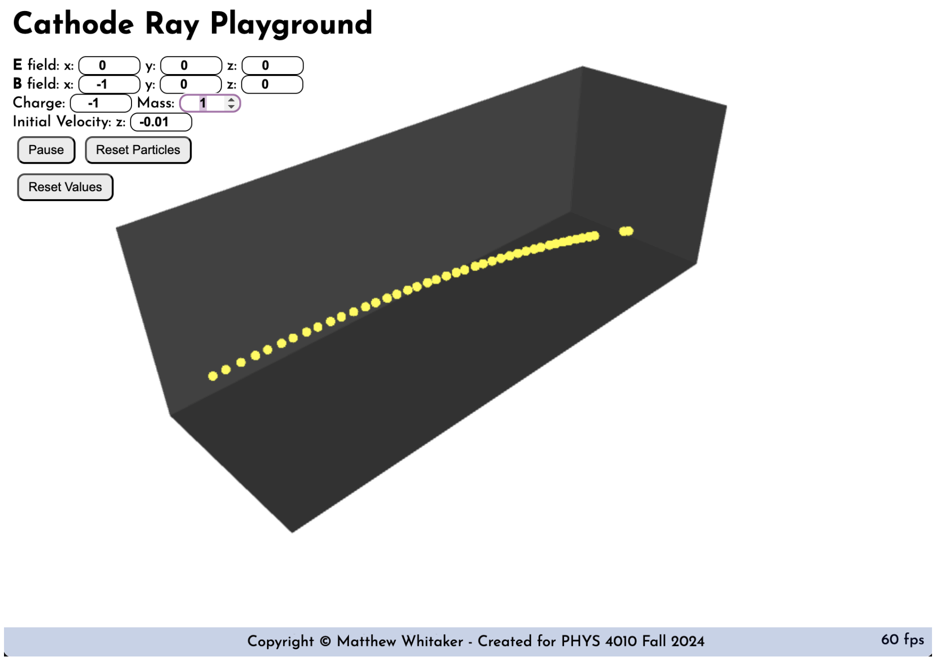
Cathode Ray Playground

E field: x: 0 y: 0 z: 0
B field: x: -1 y: 0 z: 0
Charge: 1 Mass: 1
Initial Velocity: z: -0.01

Copyright © Matthew Whitaker - Created for PHYS 4010 Fall 2024

The Deflected Cathode Ray

If you want a true cathode ray, you'll want to change the charge from 1 to a -1:



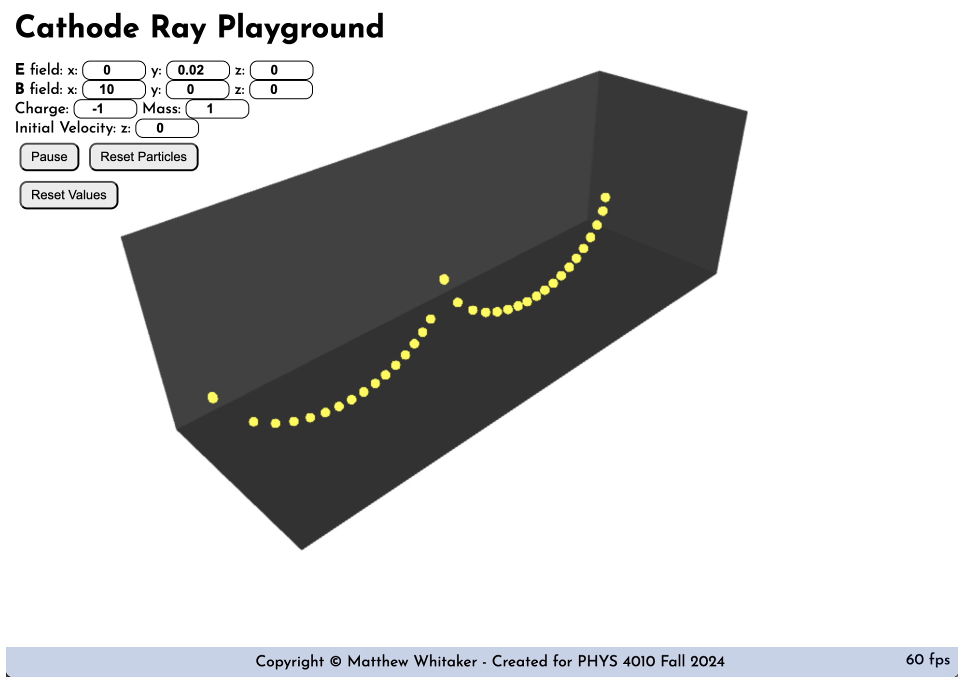
Cathode Ray Playground

E field: x: 0 y: 0 z: 0
B field: x: -1 y: 0 z: 0
Charge: -1 Mass: 1
Initial Velocity: z: -0.01

Copyright © Matthew Whitaker - Created for PHYS 4010 Fall 2024

Cycloid Motion

With perpendicular magnetic and electric fields, we get the classic half-circle motion



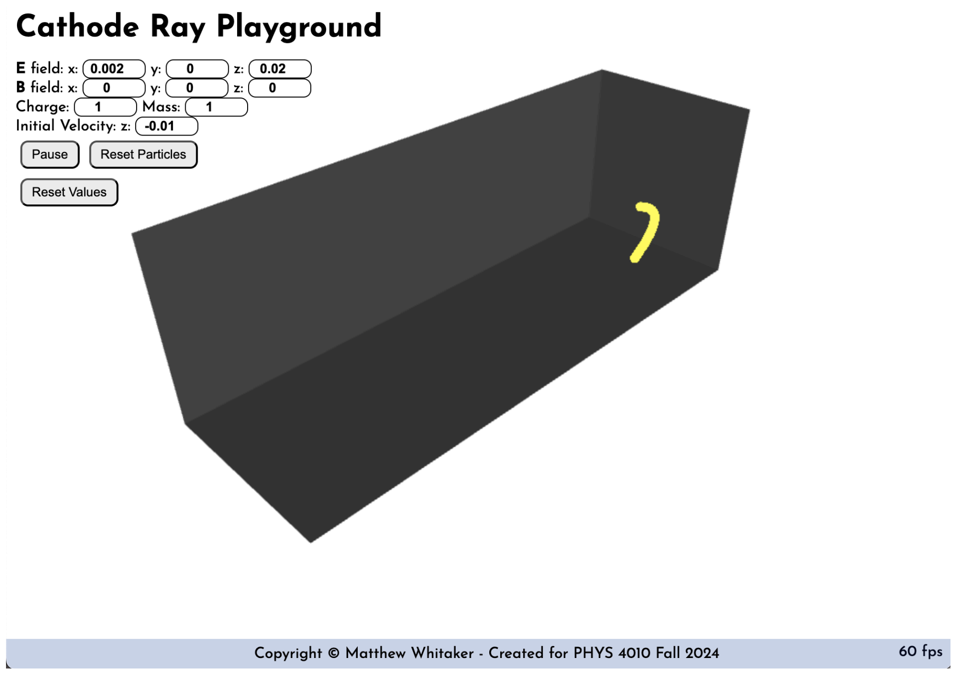
Cathode Ray Playground

E field: x: 0 y: 0.02 z: 0
B field: x: 10 y: 0 z: 0
Charge: -1 Mass: 1
Initial Velocity: z: 0

Copyright © Matthew Whitaker - Created for PHYS 4010 Fall 2024 60 fps

Parabolic Motion

Setting the electric field against the particle's motion, we see a parabolic trajectory.



Cathode Ray Playground

E field: x: 0.002 y: 0 z: 0.02
B field: x: 0 y: 0 z: 0
Charge: 1 Mass: 1
Initial Velocity: z: -0.01

Copyright © Matthew Whitaker - Created for PHYS 4010 Fall 2024 60 fps

Technical Description

The tea - the technical details in all their glory

Overview: The project utilizes WebGL 2.0, a modern graphics framework that allows a web browser running JavaScript code to utilize the GPU for rendering. The code is written primarily in JavaScript, but GPU-specific code had to be written in OpenGL Shading Language (GLSL). The graphics are rendered onto an HTML Canvas. All modern web browsers should support these features — using Internet Explorer may unleash unspoken horrors. I also utilized the JavaScript library gl-matrix by Brandon Jones
and Colin MacKenzie IV, which is open source software available via the MIT License at https://glmatrix.net/. This library brings matrix and vector operations to JavaScript (analogous to the package NumPy in Python).

As mentioned previously, the simulation is available at https://em.matthewwhitaker.me/. If instead, you want to run the simulation from the source code, you'll need a lightweight web server. Running `python3 -m http.server 8000` from the root directory of my project code is a simple way to get up and running. From there, visit http://localhost:8000 in your browser to see it in action.

Traditional GPU Programming: Web-based GPU programming is unique. The GPU's job is to take numerical data (like a list of particle positions and a camera position) and translate this into pixel colors on a screen with the right 3d → 2d projection. Typically, a GPU program follows the following process for every frame rendered to the screen:

Input Data → Shader Programs → Output Graphics

My Approach: The data only flows one way, so the GPU can't directly pass any information back to the CPU, so any physics simulations have to be updated once every frame by the CPU, and the new positions passed to the GPU to be rendered. Unfortunately, running simulations using the JavaScript interpreter can quickly become slow. You can't utilize the full power of your CPU to access features like parallelization or hardware-accelerated computations. (A website that can start interfacing directly with the CPU is a security risk, so a web browser sandboxes each webpage into its own dedicated process.)

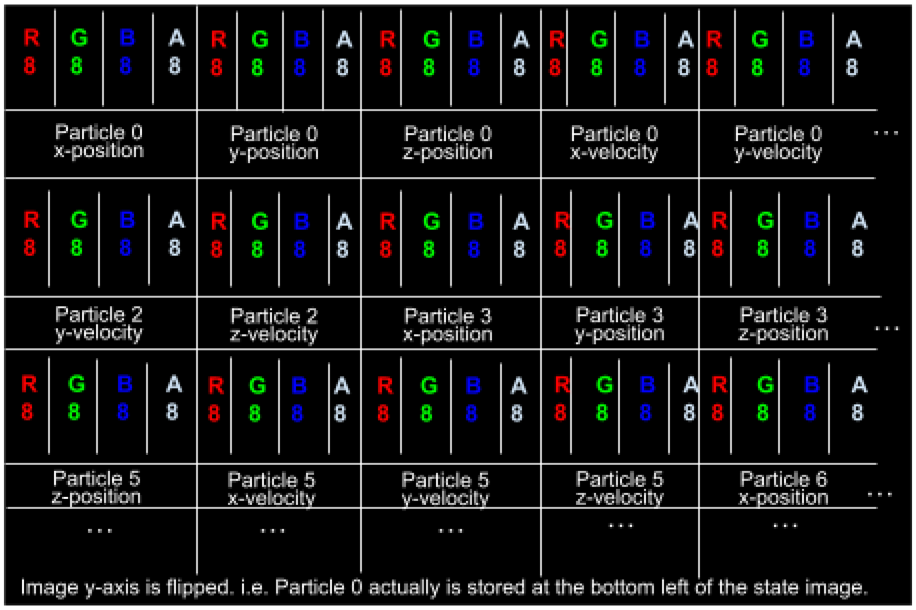
I wasn't happy with this. Simulations, even on the web, should be *snappy* and *smooth*. So, I took advantage of the fact that GPUs are really good at loading and saving images in order to perform the physics simulation *entirely in the GPU*. Here's the modified GPU process I used:

Input Data $\rightarrow$ Physics Shader $\rightarrow$ Shader Programs $\rightarrow$ Output Graphics

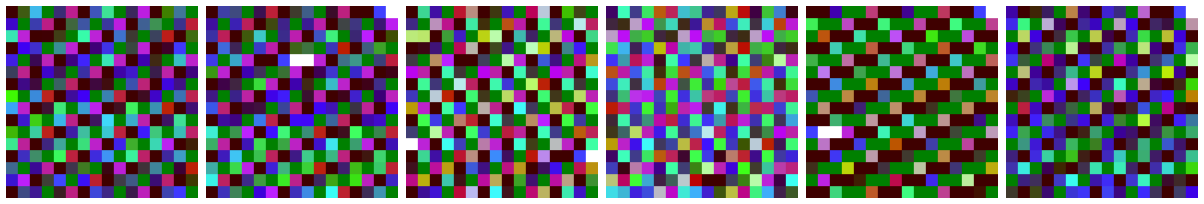
There are some hidden details in the new "Physics Shader" part:

Current State Image (**x**, **v**) $\rightarrow$ Physics Simulation $\rightarrow$ New State Image (**x**, **v**)

Now, the only input data the GPU receives from the CPU is information about the camera position and the electric and magnetic field vectors. The state is stored in memory in two 16x16 images. Rather than storing a 32-bit RGBA color, each pixel stores a 32-bit float corresponding to a position or velocity component:



State Images: One image stores the current state of each particle, and the other image stores the future state resulting from the motion due to forces from the electric and magnetic fields over a small increment in time (as described above). At the end of the computation, the two images are swapped, so that the newly computed state becomes the current state, and the previous state is ready to be overwritten. This "current state" image is then loaded into the Shader Programs, which render the particles to the screen in the correct position based on the current camera position. Here are a couple examples of the state images during a simulation:



You can see clear patterns and features unique to each image (each image is a unique simulation). Ultimately, the data stored in the pixels of each image allow the scene to be reconstructed (and the magnetic and electric fields to be deduced!).

Encodings: To convert from a 32-bit floating point number to a set of 4 8-bit color channels and back, I had to pick an encoding method. Essentially, I treated the RGBA channels as one contiguous chunk of 32 bits, and then used the most common single-precision floating point format, defined by the IEEE 754 standard. The details are on Wikipedia, so I won't bore you (assuming I haven't done so already).

Efficiency: While I decided to keep things small and simulate only 42 particles (to keep the visualization from getting crowded), the fact that the simulations are done entirely on the GPU means that the simulation can be expanded to hundreds of thousands of particles before experiencing any noticeable visual slowdown on the average computer.

Precision: Using 32-bit floats to encode the position and velocity state allows for a large dynamic range. However, there are some precision limits. For example, since we update by a fixed timestep rather than integrating continuously, some quick motions or high-acceleration regimes may show signs of imprecision. For example, a particle that is supposed to move in a quick circular orbit due to a strong magnetic field may instead spiral outwards or drift. Correcting for these errors is possible, but outside the scope of the few hours I could spend on this project.