Magnetodynamics

This document describes version 1.0 of Magnetodynamics. It is intended for both beginners and advanced users of Unity, and for both novices and experts in physics. If a section seems too pedantic, skim through it. If something doesn't make sense, skip over it.

Magnetodynamics is available for sale on the Unity Asset Store, and you can play with some projects built from it at `http://www.coffeeshopphysics.com/magnetodynamics`. Unity (`http://www.unity3d.com`) is a software framework for developing 3-D computer games. I use it to make interactive physics demos.

The first part of this document introduces the concepts and the tools by walking you through two of the demos. The second part describes each tool on its own, starting with a quick reference and ending with a tutorial demo that highlights the tool. The third part is about scripting— most of your interaction with Magnetodynamics is likely to be with prefabricated GameObjects, but there's a programming interface for the few things that need to be coded. At the end is a list of troubleshooting tips.

There's always a danger in writing long documentation— it might give the impression that the project is more complicated than it really is. I have erred on the side of trying to provide enough information, even if it looks unnecessarily foreboding.

Contents

1 Introduction: what this package does

1.1 Fast and pretty accurate physics

Magnetodynamics adds realistic magnetic fields and forces to your Unity project. It is a collection of GameObjects and associated C# scripts that magnetize your existing objects so that they can apply magnetic and electric forces, respond to fields, draw field lines, and simulate particle beams and plasmas. It is not a full calculation of Maxwell's Equations, the fundamental laws of electricity and magnetism, but an approximation to quickly compute their most visible effects with enough accuracy to "feel right" to the user.

Magnetic field calculations for science and engineering divide space or material bodies into many small volumes called finite elements. Each contributes to the electric and magnetic fields that fill the universe. The force on each element depends on the field from all the other elements, so the time required to compute the forces on everything scales with the number of elements *squared*. For accuracy, some calculations use so many elements that they require months of supercomputer time.

That's out of the question for a game engine, which must be fast. The Magnetodynamics package reduces the general description of magnetic sources to a few well-placed infinitesimal dipoles. An infinitesimal dipole is a magnetic field source that is equivalent to an infinitesimally small loop of circulating electric current. Figure 1 shows a dipole: its electric current is represented by a white cylindrical surface with red arrows and the magnetic field lines it generates are green. It can be described by a 3-D vector (shown in magenta: the arrow points toward the dipole's magnetic north). Now imagine shrinking the white cylinder down to a point of zero size while retaining the same amount of electric current. The magnetic field, especially far from the source, doesn't change much. If you embed a few infinitesimal dipoles in the things that should generate the strongest magnetic fields, such as bar magnets, cranes, MRIs, and antimatter confinement systems, the magnetic forces will be accurate enough to create the feeling that these are real magnets. The field calculations are mathematically exact, though the sources are unrealistically small.

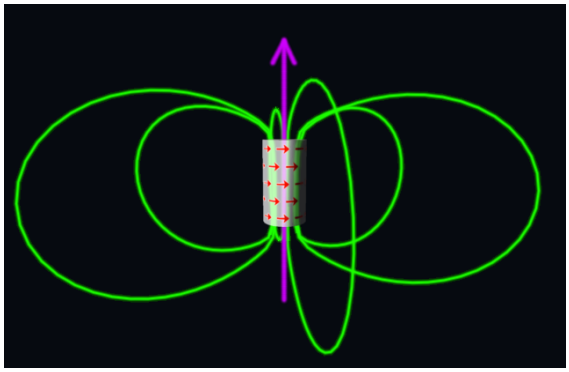

Figure 1: A not-quite infinitesimal dipole.

Any magnetic field configuration can be described as an assembly of infinitesimal dipoles (finite element calculations often put an infinitesimal dipole in each of their tiny volumes), but the purpose of this package is to minimize the number of calculations. For field configurations that would be awkward to describe with dipoles, Magnetodynamics allows you to write down the equation describing the field— all torques and forces would then be applied correctly. In this documentation, I call fields implemented this way "external fields," and include some of the most important special cases.

Perhaps our most common experience of magnetism is when permanent magnets attract metal objects. This is also one of the most complicated phenomena and the most difficult to simulate. Magnetic fields generated by orbits of electrons and their quantum mechanical spin in the permanent magnet induce similar, though disordered, domains in the metal to line up with the direction of the applied field. The metal becomes a magnet, too, and it happens to be pointing in the right direction to be attracted by the permanent magnet (and maybe draw a few more metal objects, like a train of paperclips). An important aspect of magnetization is that it could happen anywhere in the metal and is usually unequal in different parts of the metal because it depends very strongly on distance (not an inverse-square law, but an inverse-seventh power law!). Magnetodynamics includes tools for magnetizing materials, though they have some limitations that are described in detail.

The opposite of magnetization and magnetic attraction is the anti-magnetization of a superconductor. Though we don't experience this much in our daily lives, it is a cool effect and might find a home in some video game. The simulation of magnets levitating over superconductors is more faithful to real physics than the simulation of magnetization, but it is limited to only a few shapes of superconducting material (large plane and small sphere).

If you have the Vectrosity package[1], you can unlock a feature in Magnetodynamics that draws magnetic and electric field lines. Field lines follow the direction of a magnetic field through space and either form a closed loop or stretch out to infinity. They provide an intuitive way to visualize fields, either for development or as part of the game. Vectrosity does the actual drawing, while Magnetodynamics computes the shape of the curve.

Though Magnetodynamics focuses on the most common effects in macroscopic magnetism, it also simulates static electricity and plasmas of charged particles. The attraction and repulsion of static electricity is like a simple version of magnetic forces: instead of vector-valued magnetic dipoles, electric sources are single-valued charges (positive or negative but with no direction), and while magnetic fields and their derivatives apply torques and forces on dipoles, electric fields apply only forces on charges in proportion to the strength of the field.

Unity's ParticleSystem makes it possible to visualize low-density plasmas, which are fluids of charged particles. We simply replace the forces the ParticleSystem would apply to its Particles with forces derived from the electric and magnetic fields. The plasma (or "ionized gas") accelerates in the direction of electric fields and spirals around magnetic field lines.

In real life, changing electric fields creates magnetic fields (Ampère's law) and changing magnetic fields creates electric fields (Faraday's law). While this is a deep and important concept, it doesn't often lead to visible effects (just nifty things like radio and light). Magnetodynamics does not simulate it explicitly. I may write about certain magnetic field configurations being produced by "electric currents," but those currents are in my mind, not the simulation. In this package, the electric field and the magnetic field are simply two independent fields.

[1]Vectrosity is a third-party package, also available on the Asset Store: `http://starscenesoftware.com/vectrosity.html`

The most difficult part of the physics simulation is handling collisions. Unity's internal physics engine does a good job with low-speed collisions, but when objects move fast enough for two Colliders to overlap or even skip over each other in a single time-step, the simulation becomes visibly unrealistic. Magnetic forces tax this part of the simulation because they can produce large accelerations over short distances. In a typical project, you will adjust the field strengths until they are large enough to produce an effect at an appropriate distance scale. However, magnetic forces scale rapidly with distance: the force between two dipoles is an inverse fourth-power law and the force between a dipole and a magnetized object is an inverse seventh-power law. The magnetic strength needed to just barely budge an object across the room is painful up close.

Much of the internal work in Magnetodynamics is for handling collisions more gracefully. Magnets turn off their attraction when their Colliders overlap and the magnet-magnetizable interaction is simulated by varying the spring strength in a SpringJoint. (SpringJoints already handle collisions rather well.) One of the tutorial projects, Flying Wrenches, shows the various approaches used to handle an extreme case.

In case any of my physics friends complain about the name of this package, the word "magnetodynamics" is meant to emphasize this focus on the human-scale magnetic part of electromagnetism. I chose it because it is not often used in scientific literature, and when it is, its meaning varies. Also, it contains "neat-o!"

1.2 Package contents

The Magnetodynamics package installs two top-level directories in your project: Magnetodynamics and ExampleScenes. Only Magnetodynamics is needed for the package to work: ExampleScenes provides the demos discussed in this document. You may want to mine them for examples of how to implement special cases, like horseshoe magnets, particle beams, static clingy sheets, well-spaced field lines, etc.

Within the Magnetodynamics folder, ObjectsToPlaceInYourScene contains prefabricated objects to embed in your GameObjects. Most of them are invisible; the few that have a visible MeshRenderer are translucent to help you position them in your Scene. You'll probably want to turn off the MeshRenderer when it's in the right place. The subfolder, _ScriptsThatMakeThemWork, contains scripts that have already been attached to the prefabricated objects. You probably won't need to go into this folder, though you can.

One reason you might need to go into _ScriptsThatMakeThemWork is to let Magnetodynamics know that you have the Vectrosity package and enable field lines. Instructions are provided in the reference section on FieldLine.

The second folder in Magnetodynamics, ScriptsToEnhanceExistingObjects, are script Components that you attach to your object. These don't have an associated prefab because no positions or orientations need to be aligned in the Scene view.

The ExampleScenes folder has a Scene to demonstrate each tool. They're not organized very well and they borrow each other's assets. If you want to run any one of them, install all of them. But again, if you only want to use the Magnetodynamics package and not look at any of the examples, you don't have to install the ExampleScenes folder.

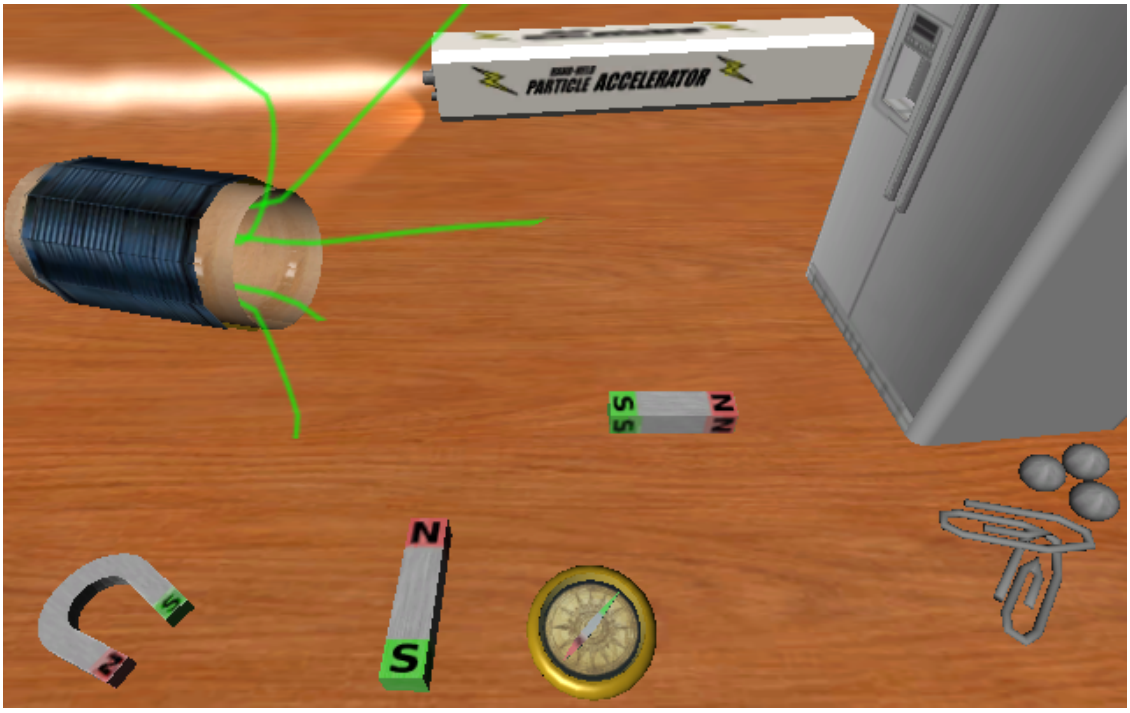

Figure 2: The TabletopExperiments Scene contains a little of almost everything: magnetic dipoles (two bars and a horseshoe), a dipole with a fixed position (compass), magnetizable materials (paperclips, ball bearings, and little refrigerator), an external field (solenoid of wires wrapped around a toilet paper roll), and plasma (particle accelerator, hand-held).

1.3 Walk-through of magnetic simulations

Open and run the TabletopExperiments Scene (in the ExampleScenes folder) or play it through the Unity web interface at `http://www.coffeeshopphysics.com/magnetodynamics` (shown in Fig. 2). This scene was designed to give an overview of (almost) all the tools. While the demo is running, you can move most of the objects with the mouse. Play around with them and see what they do.

If you have Vectrosity and you want to enable the field lines (green lines in Fig. 2), open Magnetodynamics/ObjectsToPlaceInYourScene/_ScriptsThatMakeThemWork/FieldLineScript and remove the '`//`' before '`#define I_HAVE_VECTROSITY`'. Then save and re-run.

This Scene has a total of seven infinitesimal dipoles (MagneticDipole prefabs) embedded in four Rigidbodies. By contrast, a full finite element simulation could have thousands to billions of elements. Each bar magnet is modeled with two dipoles, one near each end, both pointing in the same direction. The horseshoe magnet also has two, pointing in opposite directions. The compass needle has one MagneticDipole with a fixed position— it is only allowed to rotate. For more accuracy, they could be modeled with a larger number of weaker dipoles, approaching the granularity of a finite element simulation. However, the simple field shapes that this demo requires wouldn't justify the additional calculations.

In Unity's Scene and Hierarchy views, you can see how MagneticDipoles are attached to objects. Figure 3 is an example showing the horseshoe magnet. The horseshoe is a regular GameObject— it has a MeshRenderer to be visible, a Rigidbody to apply forces and torques, and a Collider to identify collisions. The MagneticDipoles are new objects nested within it (they are indented in the Hierarchy view). Since the positions of nested objects are glued to their parent's position, Magnetodynamics only has to generate fields and apply forces to the MagneticDipoles and it will look like the fields and forces come from the visible horseshoe.

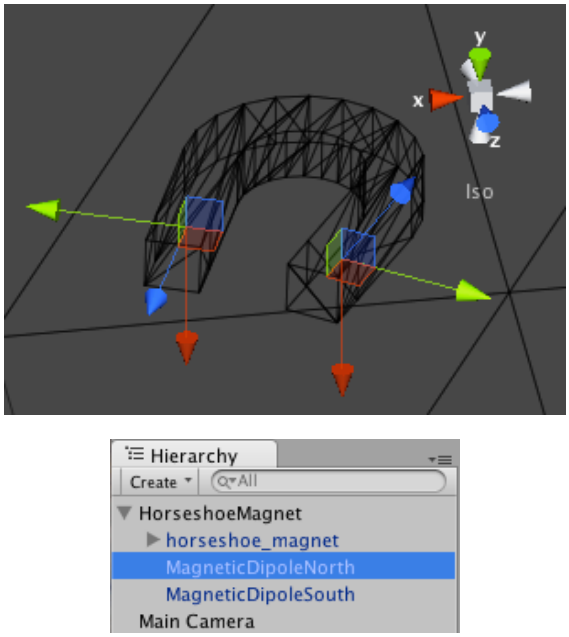

Figure 3: Anatomy of a horseshoe magnet. Note the directions of the blue arrows.

The paperclips and ball bearings on the right of Fig. 2 are examples of magnetizable materials. They don't contain nested objects, just a MagnetizableMaterial script. It has no positions to fine-tune in the Scene view.

Drag a magnet toward one of the paperclips. When it is close enough, the force between them will overcome the table's friction and the two will snap together very quickly. This is because the force between them scales as distance to the inverse seventh power. In principle, that applies equally throughout the volume of the paperclip's material, but the distance dependence is so steep that one point on the surface of the paperclip will contribute much more than the rest. Therefore, we only compute the force due to one point, the point that is closest to the dipole, and effectively magnetize the whole volume of its Collider with an inexpensive algorithm.

The (very small!) refrigerator at the right of Fig. 2 contains a MagnetizableMaterial with a fixed position. You can drag magnets onto it and they'll stick.

On the left of Fig. 2 is a toilet paper roll wrapped in a coil of wire. Running an electric current through the wire would create a solenoid, an electromagnet whose field is nearly constant inside the tube and splays out of both openings. This field is simulated with an ExternalFieldSolenoid object. This object is a translucent cylinder that you can position as needed in the Scene view (see Fig. 4). Scaling and rotating the cylinder changes its magnetic field in the appropriate ways, so you can arrange this abstract cylinder to line up with your visual representation of the electromagnet's wires. Once aligned, you'll probably want to turn off the ExternalFieldSolenoid's MeshRenderer so that it becomes invisible.

If you have Vectrosity and the FieldLines are enabled, you should see five green field lines reaching out of the solenoid's mouth like a creature of the deep. These FieldLine objects

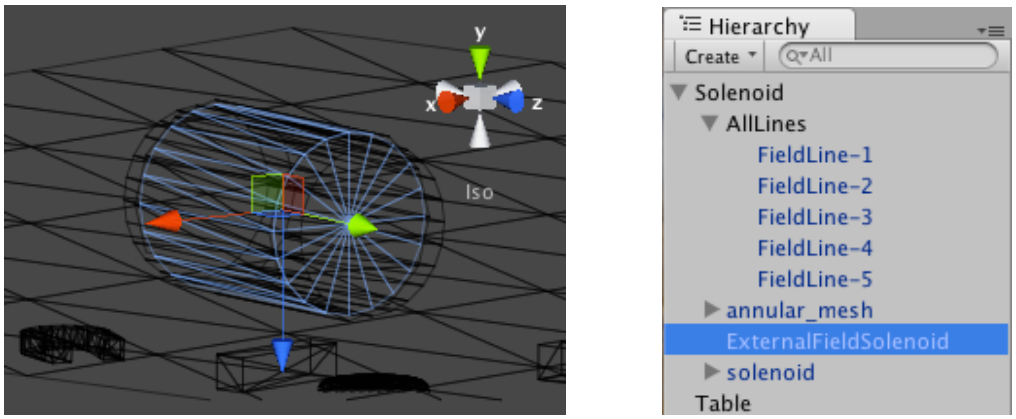

Figure 4: ExternalFieldSolenoid in the Scene and Hierarchy views.

are also prefabs, dropped into the Scene from the Magnetodynamics package. They mark the starting points of curves that follow the flow of magnetic field vectors from south to north. You can set their positions with the mouse, but their orientations are unimportant (the magnetic field defines the directions, even at the beginning of the curve).

Finally, in the back of Fig. 2, there's a hand-held particle accelerator (wish I had one). It emits a beam of charged particles, a Unity ParticleSystem whose Particles have been modified to obey the Lorentz force law. If you drag the accelerator to the magnets, or the magnets to the accelerator, you should see the beam bend.

You can add a plasma like this using the prefabricated PlasmaStream object, and then adjust the ParticleStream parameters as desired. (The parameters related to forces won't work because Magnetodynamics hijacks the kinematics of all the Particles). Alternatively, you can plasmafy an existing ParticleStream by attaching the PlasmaIonizedGas script to it.

Last and most importantly, the scene contains an ElectromagneticFieldController. The position of this GameObject means nothing, but it must be present for any magnetic or electric fields to work. It acts as a central point in the communication of field information between the magnetic and charged objects.

1.4 Walk-through of static electricity simulations

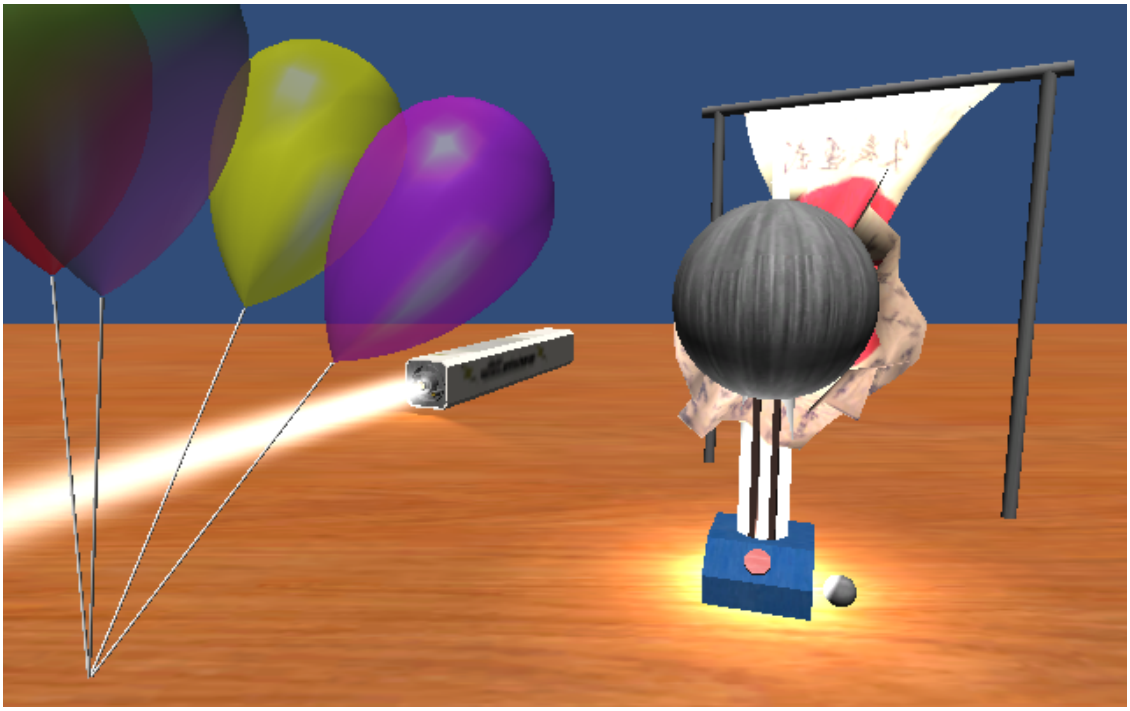

Figure 5: The VanDeGraaffExample Scene demonstrates the simulation of static electricity. A flag (InteractiveCloth), balloons, and a ball bearing are attracted to a Van de Graaff generator. There's also a beam of particles that can actually orbit the generator.

Though Magnetodynamics is primarily focused on magnetic fields and forces, it can simulate static electricity as well. To see some examples of this, open the VanDeGraaffExample Scene and run it. You can move the ball bearing (bottom right) and the particle accelerator as before, and you can turn the Van de Graff generator on and off by clicking its red button.

StaticCharge objects are embedded within GameObjects with Rigidbodies and Colliders, just as the MagneticDipoles were in the previous example. Each balloon has one StaticCharge, as does the ball bearing and the dome of the Van de Graaff generator. The on/off switch modifies the `strength` parameter of the Van de Graaff's charge through a script.

Although it would be more realistic to model these electric field sources on the surfaces of the balloons and the metal balls, a charge in the center is mathematically equivalent. That is, it produces the same electric field outside the sphere, but inside the sphere the field is wrong because it ought to be zero. Since nothing in the game can get inside the balloons to discover our fakery, we're safe.

The static clingy flag uses Unity's InteractiveCloth feature, which simulates a waving, pliable mesh. I added static charges to the cloth with twelve low-mass Rigidbody Colliders, each containing a static charge, at regular intervals. They are not nested in the hierarchy— the InteractiveCloth has an `attachedColliders` parameter in which we can set

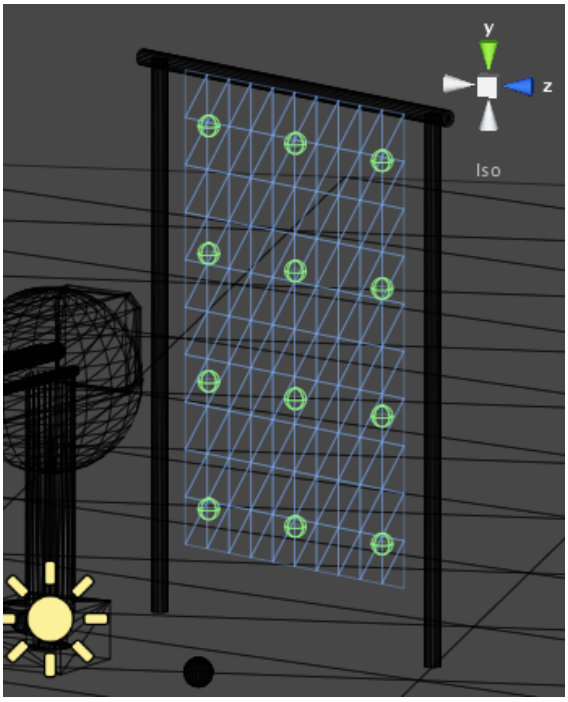

Figure 6: StaticCharges in Colliders attached to the InteractiveCloth of the flag.

a `twoWayInteraction` link between the cloth and each of the charges. Figure 6 shows how they are arranged.

If you drag the particle accelerator close to the base of the Van de Graaff generator while it's on, you can get the particles to orbit its dome in a full ellipse that follows Kepler's laws like a belt of asteroids. Cool!

2 Scene-building Reference

2.1 ElectromagneticFieldController

2.1.1 Reference

Every Scene that uses Magnetodynamics must contain an ElectromagneticFieldController object. It does not matter where it is located and it has no visible representation: it has only one Component, the ElectromagneticFieldControllerScript. You can change its name to something other than `ElectromagneticFieldController`, but then you would have to explicitly link all magnetic and charged objects to it by hand. If you leave the name as it is, they will find it automatically.

Most of what this object/script does is not important to the end-user. The one option that you may need to change is `collisionLayer` (see Discussion below).

Once you add it to your Scene, you can safely ignore it, with the possible exception of the collisionLayer parameter. Here is a description of all of the ElectromagneticFieldControllerScript parameters.

Name	**Type**	**Default**	**Description**
`ambientElectricField`	`Vector3`	`Vector3.zero`	constant space-filling electric field
`ambientMagneticField`	`Vector3`	`Vector3.zero`	constant space-filling magnetic field
`colliderPoof`	`float`	`1.0`	factor to multiply the size of MagneticDipole trigger-colliders to better handle collisions (see Discussion below)
`dampenLanding`	`float`	`0.5`	exponential decay of velocity during MagneticDipole collision
`dampenLandingSteps`	`int`	`10`	number of steps in velocity decay
collisionLayer	`int`	`9`	layer reserved for MagneticDipole trigger-colliders *(change this if you use layer 9 for something else!)*
`maxMagnetized-ForceOverMass`	`float`	`20.0`	maximum force-divided-by-mass to apply to magnetizable materials

2.1.2 Discussion

The `ambientElectricField` and `ambientMagneticField` parameters duplicate the function of the ExternalFieldConstant object with `onlyWithinBox = false`.

The `colliderPoof`, `dampenLanding`, and `dampenLandingSteps` parameters all control the way MagneticDipoles collide. When magnets move quickly, they can temporarily overlap each other, which can bring the MagneticDipoles into close contact, which can create enormous forces. Sometimes, this leads to a magnet flying away faster than you can blink, and sometimes they just roll around on the ground indecently.

To manage this, each Rigidbody containing MagneticDipoles has two Colliders: the one you give it when you declare it as a Rigidbody, and one generated at run-time, managed by

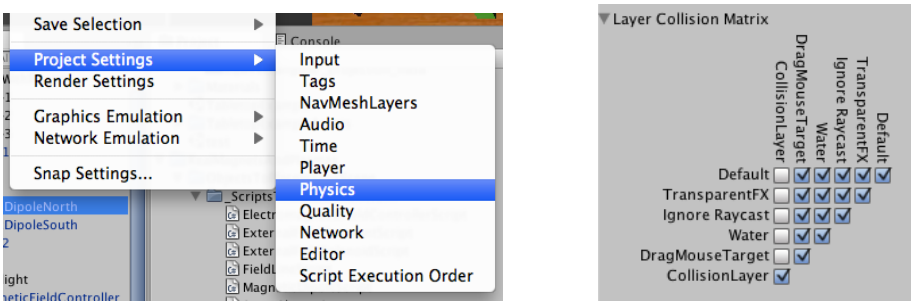

Figure 7: Isolating the magnets' trigger-collision layer for better performance.

ElectromagneticFieldController. The magnets' Collider is a trigger (`isTrigger = true`), so it does not generate contact forces, but it does inform the ElectromagneticFieldController that two magnetic bodies are unphysically overlapping and should be treated carefully. Magnetic forces between MagneticDipoles in these Rigidbodies are turned off and their relative velocities are smoothly brought to a stop before letting Unity apply the contact forces to make the magnets recoil.

Usually, this trigger-collider has the same size as the original Collider because `colliderPoof`, a factor which multiplies the size of the trigger-collider, is 1.0 by default. Sometimes, though, it is necessary to stop magnetic forces early, which can be done by poofing the trigger-colliders out to 1.02 (2%) or 1.05 (5%). The TabletopExperiments example has `colliderPoof = 1.02`. Since some magnets may be much larger than others, MagneticDipoles have an `overrideColliderPoof` that can be used to set trigger-collider sizes independently.

The smooth decay of the relative velocities is controlled by `dampenLanding`, which is a factor multiplied to the velocity `dampingLandingSteps` times. The trigger-colliders are assigned to the layer specified by `collisionLayer`.

If you are not using this layer for anything else, you can reduce the computation load by turning off interactions with non-magnets in the Layer Collision Matrix. As shown in Fig. 7, select the Edit → Project Settings → Physics menu and find the Layer Collision Matrix in the Inspector. I have named my layer 9 "CollisionLayer" so that it will show up in this matrix: you can do that in Edit → Project Settings → Tags. By unselecting interactions between CollisionLayer and everything else, its triggers will only be called when magnet touches magnet, not at other times. The script only acts on collisions between magnets, so it will function properly without this step. This step can only improve the performance.

The `maxMagnetizedForceOverMass` parameter solves a similar problem for the force between MagneticDipoles and MagnetizableMaterials. The active point on a MagnetizableMaterial is always on the surface of its Collider, so they would have to be poofed too much to use the same method. Instead, I put a maximum cut-off on the force per unit mass that it can apply. By Newton's second law ($F = ma$), that's a maximum cut-off on the acceleration (you don't want things to move too fast on the screen), except that it would not be correct to call it a "maximumAcceleration" unless it were applied to the sum of all forces. The physicist in me could not abide such sloppy nomenclature.

2.2 MagneticDipole

2.2.1 Reference

To make your GameObject feel magnetic forces and generate magnetic fields, add one or more MagneticDipoles to it. The MagneticDipoles must be nested within a larger GameObject, as in Fig. 3, and that GameObject must have a Collider (BoxCollider, SphereCollider, CapsuleCollider, or MeshCollider) and a Rigidbody.

A MagneticDipole is a singularity of magnetic field: at the point in space occupied by the dipole, the magnetic field is infinitely strong. As you approach the dipole, it grows without bound. Therefore, make sure to bury the dipoles deep within the Collider like a nerve in a tooth, so that two MagneticDipoles can never get very close to one another.

You can set the position and direction of the MagneticDipole in the Scene view. The blue arrow points north or south (the other two are not relevant).

Name	**Type**	**Default**	**Description**
`electromagneticFieldController`	`GameObject`	`null`	pointer to the scene's ElectromagneticFieldController; only needed if you changed its name
`strength`	`float`	`300.0`	strength of the magnetic dipole in Ampere-square meters (A·m^2); can be negative
`applyTorqueOnly`	`bool`	`false`	if `true`, only apply torques to the MagneticDipole, not forces
`overrideColliderPoof`	`float`	`1.0`	if non-negative, overrides the value of ElectromagneticFieldController's `colliderPoof` parameter for this MagneticDipole only

2.2.2 Discussion

The unit used for magnetic field is Tesla (abbreviated as T, equivalent to Newton-second per Coulomb-meter) and so the unit for a magnetic dipole is Ampere-square meters (A·m^2, equivalent to Coulomb-square meters per second). Typical values are hundreds of A·m^2, but don't be afraid to change the strength by factors of ten to find the right scale for your configuration. If the strength is positive, the MagneticDipole's blue arrow points north, if negative, it points south.

A dipole's contribution to the magnetic field at a point is

$$\vec{B}_{\vec{m}}(\vec{r}) = \frac{\mu_0}{4\pi}\frac{1}{r^3}\Big[3(\vec{m}\cdot\hat{r})\hat{r} - \vec{m}\Big] \tag{1}$$

where μ_0 is the permeability of free space ($4\pi \times 10^{-7}$ N/A^2), $\vec{r}$ is the displacement between the dipole and the point in question, r is its magnitude (distance) and $\hat{r}$ is a normalized vector in the same direction, and $\vec{m}$ is the dipole vector. The `strength` parameter is the magnitude of $\vec{m}$.

The torque on a dipole due to total field $\vec{B}$ at a point is

$$\vec{N} = \vec{m} \times \vec{B} \tag{2}$$

and the force on a dipole due to the same field is

$$\vec{F} = \nabla(\vec{m} \cdot \vec{B}) = \vec{m} \times (\nabla \times \vec{B}) + (\vec{m} \cdot \nabla)\vec{B} \tag{3}$$

$$\text{and by } (\vec{m} \cdot \nabla)\vec{B}\text{, I mean } m_x \frac{d}{dx}\vec{B} + m_y \frac{d}{dy}\vec{B} + m_z \frac{d}{dz}\vec{B}. \tag{4}$$

The torque is a product of the dipole strength vector with the value of the magnetic field, so it depends on dipole separation as $1/r^3$. Like tides in gravitation, the force on a magnetic dipole is a differential force, and the application of the derivative (∇) steepens its distance dependence to $1/r^4$. All derivatives in Magnetodynamics are analytic, not numerical, which improves the numerical stability.

Since the force calculation is a bit more expensive, you can opt to skip it by setting `applyTorqueOnly` to true. This is useful for dipoles with constrained positions, such as compass needles, that would be negligibly affected by the force anyway. (I'm assuming that the compass is heavy enough that it isn't dragged toward the field.)

When calculating the torque and force on a particular MagneticDipole, all field sources except the MagneticDipoles in the same Rigidbody are included. The reason we exclude the torque/force of a dipole on itself is because it is infinite in classical physics. (This "self-interaction" is an interesting topic in quantum field theory!) The reason we exclude the torque/force of other dipoles in the same rigid body is because all they would do is create stresses within that body. If the rigid body is perfectly rigid (it doesn't bend or burst apart due to the competing forces within itself), then it must be compensating those forces with the strength of its internal structure. Hence, the internal forces can't lead to any visible effects, so they aren't even applied.

When two Rigidbodies containing MagneticDipoles come into contact, the force between them is turned off and their velocities are slowed to a stop. This avoids the unrealistically strong magnetic force and/or recoil that would occur if the dipoles get too close to each other. See the previous section on ElectromagneticFieldController, which has some settings to control this collision-handling at a global level.

Magnetic forces are applied using `ForceMode.Impulse`, rather than `ForceMode.Force`. The latter applies forces in a gradual way across several time-steps, but this means that they would become out-of-date as the dipoles move. The simulation calls the Rigidbody's `AddForceAtPosition()`, rather than `AddForce()`, so an off-center MagneticDipole can create a torque in addition to that of Eqn. 2.

2.2.3 Tutorial project: Ball Magnets

To get some practice setting up MagneticDipoles, open the BallMagnetsTutorial scene and run it. It contains a ball with an "N" painted on one side and "S" on the other. Nothing in the scene has been magnetized yet.

From the ObjectsToPlaceInYourScene folder, drag a MagneticDipole into the ball magnet. It is important for the MagneticDipole to be nested within BallMagnet-1, as in Fig. 8 (and also Fig. 3). Make sure that it is centered in the ball.

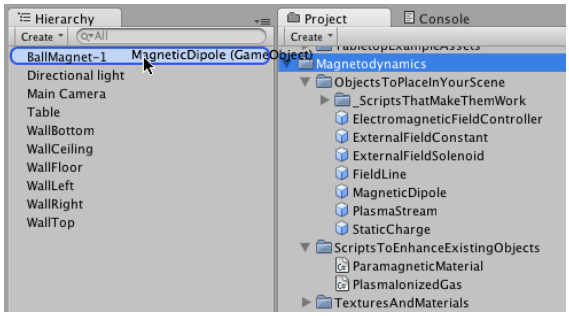

Figure 8: Dragging a MagneticDipole into BallMagnet-1.

Next, switch to an overhead view (click on the Y axis in the Scene view) and rotate the MagneticDipole so that it is aligned with the graphical "N" and "S" on the surface of the ball (Fig. 9). Check it from a few other views to be sure that the MagneticDipole is in the center of the ball.

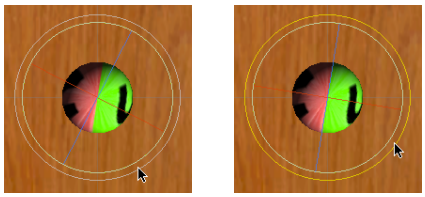

Figure 9: Rotating the MagneticDipole so that it is aligned with the "N" and "S."

Drag an ElectromagneticFieldController into the scene so that the MagneticDipole will work. (Without the controller, the MagneticDipole will raise an exception at run-time.) Make many copies of BallMagnet-1 and make sure they're in different places on the table. Run it again.

The ball magnets will try to group themselves in ways that minimize their total energy

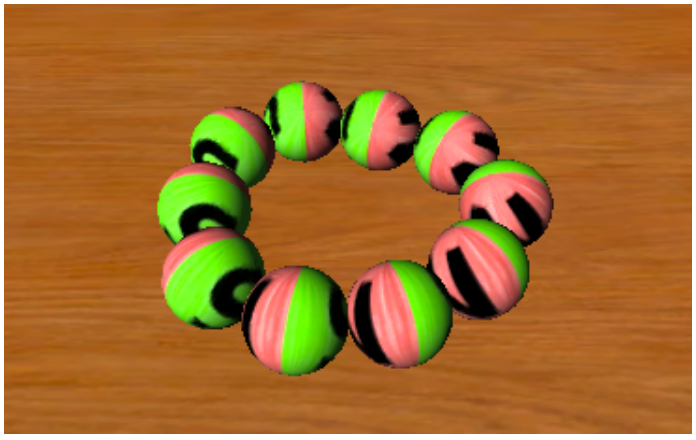

Figure 10: Ball magnets searching for a minimum-energy state. This ring is not a global minimum; it collapses rather easily.

(Fig. 10). They form all sorts of interesting structures because they not only want to be close together, they also want to be aligned. To keep them from oscillating for a long time (dissipating energy too slowly), I've given them a lot of friction and angular drag.

Try selecting them all at once and uncheck `useGravity` in their Rigidbodies. The minimum-energy configurations in zero gravity are different from the ones you get when they're confined to the table's surface.

2.3 StaticCharge

2.3.1 Reference

To make your GameObject feel electric forces and generate electric fields, add one or more StaticCharges to it. The StaticCharges must be nested within a larger GameObject and that GameObject must have a Collider (BoxCollider, SphereCollider, CapsuleCollider, or MeshCollider) and a Rigidbody.

A StaticCharge is the electric equivalent of a MagneticDipole. Electricity is a bit simpler than magnetism: an electric charge has no direction, so it doesn't matter what the StaticCharge's orientation is. The distance dependence of the force is also less steep (inverse-square, rather than inverse-fourth power). Though StaticCharges are singularities of electric field, it is less important to bury them within Colliders than it is for MagneticDipoles, because of the less steep distance dependence.

You can position StaticCharges in the Scene view, but the direction of its coordinate axes (the three arrows) is irrelevant.

Name	**Type**	**Default**	**Description**
`electromagnetic-FieldController`	`Game-Object`	`null`	pointer to the scene's ElectromagneticFieldController; only needed if you changed its name
`strength`	`float`	10^{-8}	strength of the static charge in Coulombs
`overrideColliderPoof`	`float`	`-1.0`	if non-negative, overrides the value of ElectromagneticFieldController's `colliderPoof` parameter for this StaticCharge only

2.3.2 Discussion

The unit used for electric field is Volt per meter (abbreviated V/m) and so the unit for charge is a Coulomb (C). Typical values range from 10^{-8} C to 10^{-6} C, so don't be afraid to change the strength by factors of ten to find the right scale for your configuration. If you want two charges to attract each other, be sure to make one positive and the other negative.

A static charge's contribution to the electric field at a point is

$$\vec{E}_q(\vec{r}) = \frac{1}{4\pi\varepsilon_0}\frac{q}{r^2} \tag{5}$$

where ε_0 is the permittivity of free space ($8.85418782 \times 10^{-12}$ F/m or Farads per meter), $\vec{r}$ is the displacement between the charge and the point in question, r is its magnitude, and q is the strength of the charge (the `strength` parameter).

An electric field does not produce torques on static charges. The force on a charge due to total field $\vec{E}$ is simply

$$\vec{F} = q\vec{E}. \tag{6}$$

When calculating the force on a particular StaticCharge, all field sources except other StaticCharges in the same Rigidbody are included. StaticCharges also use Rigidbody's `ForceMode.Impulse` and `AddForceAtPosition()` for the same reasons as MagneticDipoles.

2.3.3 Tutorial project: Sun and Earth

Since the forces on electric charges and gravitating masses both have an inverse-square distance dependence, we can use StaticCharges to simulate planetary motion. Open the SunAndEarthTutorial Scene. It contains two spheres, both of which are Rigidbodies without "gravity" (Unity's simulation of a constant downward force). When you run the scene, the Earth will shoot off into the distance because it has a simple script giving it an initial velocity. We'd like the Sun and the Earth to attract each other so that they go into orbit.

Just as you did with MagneticDipoles, drag a StaticCharge into the Hierarchy view such that it is a child of the Sun and make sure that it is positioned in the exact center of the ball. Drag in another StaticCharge and make this one a child of the Earth. By default both have a charge of 10^{-8} C, which would make them slightly repulsive. To attract, they need to have opposite charges, though it doesn't matter which is negative. By trial and error, I've also found that the charges need to be stronger than the default to produce a nice orbit. ("Nice" means fast enough to see.) Set the Sun's charge to -5×10^{-5} C and the Earth's charge to 5×10^{-7} C. Notice that these choices are in proportion to their Rigidbody masses: only 100 to 1 in this ridiculous toy. For gravity, the strength of attraction (due to mass) is always proportional to the amount of inertia (mass).

Before you can run it, you need to drag an ElectromagneticFieldController into the Scene. If you don't, you'll get error messages and no electric forces.

Now the Sun and the Earth should orbit each other nicely. I say "orbiting each other" because neither is completely stationary (a fact that is used to find extrasolar planets). If you select the Sun while the simulation is running, you'll see the position of its Transform wiggling a little (look in the Inspector). Try making their masses and charges less extreme than the 100-to-1 ratio and you should see the wiggle more clearly. If their masses and charges are exactly the same, they'll both orbit a point halfway in between.

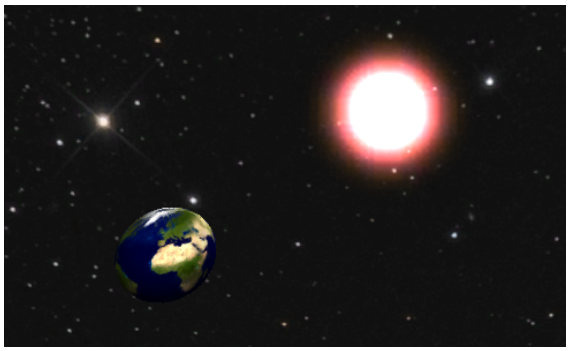

Figure 11: A nice orbit.

You can also change the Earth's initial velocity (StartEarthMoving.js) to get more elliptic orbits or even a hyperbolic orbit (it goes away and never comes back). You may get that case by accident while tinkering with the masses and charges.

2.4 MagnetizableMaterial

2.4.1 Reference

To make an object such as a piece of metal susceptible to magnetic attraction, add the MagnetizableMaterial script to it. Magnetodynamics simulates the force of attraction, but not the induced magnetic fields in the object. The entire volume of its Collider is effectively magnetized. The parameters of the script are:

Name	Type	Default	Description
`electromagnetic-FieldController`	`Game-Object`	`null`	pointer to the scene's ElectromagneticFieldController; only needed if you changed its name
`strength`	`float`	10^{-14}	$\chi/(1+\chi)$ times volume, where χ is magnetic susceptibility
`applyExternalForce`	`bool`	`false`	if `true`, apply forces due to external fields
`applyDipoleForce`	`bool`	`true`	if `true`, apply forces due to MagneticDipoles

2.4.2 Discussion

Magnetization is the least realistic simulation in the Magnetodynamics package because the physics of real magnetization is complex. The field of a magnetic dipole makes a small volume of metal become a dipole on its own. The new dipole is

$$\vec{M} = \chi \vec{H} \tag{7}$$

where $\vec{H}$ is a way of describing the magnetic field that includes all of the induction of everything else in the universe. Just outside the surface of a magnetized material, $\vec{H} = \vec{B}/\mu_0$ (symbols already defined in previous sections). The capital $\vec{M}$ differs from the infinitesimal dipole $\vec{m}$ in that it is dipole per unit volume. The units of $\vec{M}$ are A/m and the units of $\vec{m}$ are A·m^2.

The unitless factor χ describes how magnetizable the material is. Iron is more magnetizable than nickel, for instance. This χ isn't even a constant for most metals, indeed, for anything whose magnetization is strong enough to notice, χ depends on $\vec{M}$ and often the history of past encounters with $\vec{H}$. In some cases, χ isn't a single number, but a tensor that creates induced fields at odd angles relative to the external field.

This package doesn't simulate any of that. It only calculates what the force of attraction would be if a field had been induced at the point on the volume closest to the dipole (for

all dipoles). The field from the dipole $\vec{B}_{\vec{m}}$, calculated from Eqn. 1, is assumed to induce a dipole

$$\vec{m}_{\text{induced}} = (\texttt{strength})\,\vec{B}/\mu_0, \tag{8}$$

which ignores the distinction between $\vec{m}_{\text{induced}}$ and $\vec{M}_{\text{induced}}$. (The infinitesimal dipole $\vec{m}_{\text{induced}}$ represents a small patch of dipole density $\vec{M}_{\text{induced}}$ with some unspecified volume). Instead of generating a field, this induced dipole is immediately used to calculate a force by Eqn. 3.

Since `strength` is a constant with no history dependence, the material is effectively a paramagnet (like liquid oxygen), not a ferromagnet (like iron). Negative values of `strength` are not allowed. Negative values would make magnets *repel* the object, an effect called diamagnetism. Many materials that are not obviously magnetic, such as water, are very weakly diamagnetic. (Search the Internet for pictures of frogs levitating in strong magnetic fields.) For the special case of perfect diamagnetism, see the next section on superconductors.

The magnetic field of a dipole depends on distance as an inverse-cube law, and the force on an induced dipole depends as an inverse-fourth power law. Since the dipole must be induced before it can be attracted, the total force between a permanent dipole and its induced dipole is a product of the two: an inverse-seventh power law. This steep distance dependence causes problems when a permanent dipole and an induced dipole collide. Unity applies contact forces to try to oppose the attraction and keep the two Colliders from passing through each other, but both the magnetic force and the contact force are usually very large. A competition between the two either results in the objects rocketing away from each other or tumbling on the ground in a suggestive manner.

Unity handles collisions between Rigidbodies connected by a SpringJoint rather well, so the force between MagneticDipoles and MagnetizableMaterial is implemented by creating SpringJoints at run-time. A spring produces a force following Hook's law,

$$\vec{F}(\vec{r}) = -k\vec{r} \tag{9}$$

where k is a positive spring constant (`SpringJoint.spring`). Magnetodynamics dynamically adjusts k so that it is always equal to the magnetic force divided by displacement ($|\vec{F}|/|\vec{r}|$). For each MagneticDipole-MagnetizableMaterial pair, two SpringJoints are created: one applies the force on the MagnetizableMaterial due to the MagneticDipole and the other applies the force on the MagneticDipole due to the MagnetizableMaterial. (Newton's third law has to be enforced by hand!) SpringJoints in Unity cannot have negative spring constants, so that's why negative `strength` is not allowed.

External fields can also apply a force on MagnetizableMaterials, and this simulation is more realistic. The force is calculated as above, except that the source is the external field, rather than an infinitesimal dipole. Instead of calculating the force at a single point on the surface of the Collider, the force is calculated at a random position within the Collider, using uniformly random numbers generated by the rejection method to get the shape right. The force is applied gradually using `ForceMode.Force`, rather than `ForceMode.Impulse`. Averaged over many time-steps, this produces the right overall force on the MagnetizableMaterial by smoothed Monte Carlo integration. No SpringJoints are involved.

Forces from infinitesimal dipoles and external fields can be applied or not applied separately using the `applyDipoleForce` and `applyExternalForce` parameters. In future versions of Magnetodynamics, the two techniques will be unified using a technique similar to the current external field approach.

2.4.3 Tutorial project: Flying Wrenches

To demonstrate the difficulties involved in implementing magnetized materials, I've set up an extreme situation. Open the WrenchesTutorial Scene. A negligent physicist left a box of wrenches a little too close to the supermagnet, and soon they'll be flying all over the place. The Scene is already set up with an ElectromagneticFieldController and a magnet that you can turn on (but not off) by clicking on it while the game is running. The magnet is represented by a strong MagneticDipole and also a strong ExternalFieldSolenoid. However, the wrenches are not yet magnetizable, so turning on the magnet does nothing.

Drag a MagnetizableMaterial script into all seven wrenches, one at a time. (You can't do this all at once.) Now select them all and set the `strength` to 10^{-6}. Run the simulation and click on the magnet to turn it on. Since `applyDipoleForce` is `true` and `applyExternalForce` is `false`, only the MagneticDipole applies any force on the wrenches. When the magnetic field gets strong enough, the wrenches slide off their shelves and bounce off the supermagnet like popcorn. Not exactly the right effect.

In fact, they wouldn't bounce at all if I hadn't reinforced the supermagnet with additional Colliders. To get a flat Collider surface on the cylinder, I replaced its default CapsuleCollider with a MeshCollider (the CapsuleCollider has a rounded top). MeshColliders don't seem to be as good at detecting collisions as the simple shapes, which makes sense because it's harder to determine if you're inside an arbitrary shape than a box or a sphere. (Some meshes might not even be fully closed.) To make sure that the wrenches don't go right through it, I added an invisible BoxCollider named NoneShallPass. Even that is not enough if the magnet turns on suddenly, instead of the eased transition implemented in TurnOnMagnet.cs.

In reality, a wrench flying across the room into a strong magnet probably wouldn't bounce at all— it would probably hit the wall, jangle briefly, and then stick to it, hard. The wrenches in our simulation are hitting the wall with their hilts because this is an extreme point on the surface and we only calculate the force at one point. I'm not sure why they're bouncing: all of the Physic materials involved have `bounciness = 0`.

Now switch to the ExternalFieldSolenoid instead of the MagneticDipole by selecting all wrenches, turning off `applyDipoleForce`, and turning on `applyExternalForce`. As discussed above, this is a more realistic simulation. However, when you run it, the wrenches bounce even farther, and many of them don't come back— they get stuck behind shelves and things. This might even violate energy conservation because they bounce farther away than they started.

In a situation like this, I would fake it. I'd keep the across-the-room attraction accurate by using the ExternalFieldSolenoid with `applyExternalForce = true`, but when the wrenches get close to actually hitting the wall, I'd replace the realistic simulation with a hack that produces the desired effect.

Figure 12: Wrenches flying dangerously through the air.

The Scene contains an object called StopAndStickTrigger, which is an invisible CapsuleCollider that extends in front of the supermagnet. Try turning on its MeshRenderer briefly to see where it is. It has a script called StopAndStickScript that identifies when a wrench is in danger of hitting the wall and immediately replaces its position with one that is stuck to the wall, lengthwise. They fly through the air so fast that no one would notice this teleportation— the discrete jump in position might not be much larger than the discrete time-steps it's already doing. To enable this script, set `wrenches.size` to `7` and drag each of the wrench objects into the `wrenches` List. Now when you run it, it looks right (Fig. 12).

This solution is very specific to the wrenches in this example, but it should give a flavor of the mixture of realism and cheating that is needed to get the right effect.

2.5 SuperconductorInfinitePlane and SuperconductorSmallSphere

2.5.1 Reference

SuperconductorInfinitePlane and SuperconductorSmallSphere are external field objects that that correctly simulate magnetic field distributions outside the superconducting volume. They only apply to two special case geometries: an infinite plane (the superconductor fills the bottom half of the universe; its surface is an infinite plane) and a small sphere (exactly simulates a spherical superconductor in a constant magnetic field and is a good approximation for an object that is small compared to magnetic field variation).

Superconducting objects are prefabs with a translucent mesh to help you position them in your Scene. The scripts that implement the superconductivity have only one parameter:

Name	Type	Default	Description
`electromagnetic-FieldController`	`Game-Object`	`null`	pointer to the scene's ElectromagneticFieldController; only needed if you changed its name

2.5.2 Discussion

One of the strange properties of superconductors is that they have zero magnetic field within the superconducting volume. If you move a superconductor into a magnetic field, electric currents will start flowing within the superconductor that exactly cancel the external field (called the Meissner effect, it is Lens's law with zero electrical resistance). To put it another way, superconductors expel magnetic field lines: the field values smoothly descend to zero as you approach the surface of the superconductor and are zero inside. This magnetic field configuration makes magnets levitate above the superconductor's surface.

Superconducting materials are conceptually similar to MagnetizableMaterials in that they react to other magnetic fields, but their implementation is very different. Magnetizability is a property that you can add to your existing objects with a script, while superconductors are prefabricated objects with pre-defined geometries. MagnetizableMaterials do not correctly simulate the induced magnetic fields but are subject to the forces they would create; superconducting objects simulate the expulsion of magnetic field lines but are not subject to forces. (They are implemented as external fields, so they can apply forces on other things and can be moved by hand, but they aren't pushed by magnetic forces.)

SuperconductorInfinitePlane uses an image method to calculate its response to magnetic fields. To guarantee that the total magnetic field is zero on the surface of the plane, it duplicates the field on the other side of the plane, inverted like a mirror. If the field at some point on the plane due to other sources is $\vec{B}_p$, the superconductor adds $-\vec{B}_p$ and the total field on the plane is zero. Close to the plane, it is close to zero, smoothly descending as you approach the surface. Far above the plane, the fact that the original field has a valid configuration (a solution to Maxwell's Equations) implies that the superconductor's contribution is also a valid configuration and the total field is correct (due to the superposition principle). To a magnetic dipole, the superconductor's surface looks like a mirror: the mirror-dipole does everything that the dipole does in such a way as to prevent either one of them from passing through the looking glass.

The one thing that this method gets wrong is the field inside the superconductor. Both sides of the infinite plane act like the outside of a superconductor. But if you put all of your magnets on one side of the plane, everything will be right.

SuperconductorSmallSphere implements the field distribution of a uniformly magnetized sphere with a magnetization to exactly cancel fields in its center. If it is immersed in a constant magnetic field, it will expel field lines exactly around its spherical surface (with zero field inside, unlike the SuperconductorInfinitePlane). If it is immersed in an an approximately constant magnetic field, it will approximately expel the field lines. In a wildly

varying magnetic field, it will be wrong. That's why it should be a small sphere— small compared to magnetic field variations.

These two cases handle two extremes: one in which the superconductor is much larger than all of your other magnets and another in which the superconductor is much smaller. If you're not too worried about accuracy, you could use the SuperconductorInfinitePlane for large superconductors that aren't exactly flat and you could use the SuperconductorSmallSphere for small superconductors that aren't exactly spherical.

In addition to not feeling magnetic forces, two physical effects are not modeled by this simulation. In real superconductors, the magnetic fields are not exactly zero throughout the superconducting material— fields get exponentially weaker as you get deeper into the material (exponentials scale very quickly to very nearly zero). This is called the London penetration depth. The other is the fact that superconductors can only expel fields up to a certain point: when you reach the critical field strength, superconductivity breaks down just as it would if you were to increase the temperature above the critical point.

2.5.3 Tutorial project: Hovering Magnet

Open the SuperconductorTutorial Scene and run it. It contains a bar magnet and an ElectromagneticFieldController. If you have Vectrosity and have enabled FieldLines, the magnet has six field lines attached to it to show how the field is shaped around the magnet. Below the magnet is a large sample of BSCCO ("bisko") in a bath of liquid nitrogen. But apparently it has not reached its critical temperature yet because the magnet simply falls flat on the surface.

Drag a SuperconductorInfinitePlane into the Scene and position it to correspond with the BSCCO sample. When it is in position, turn off its MeshRenderer to make it invisible. Now when you run the demo, the magnet hovers (Fig 13): as it falls toward the superconductor,

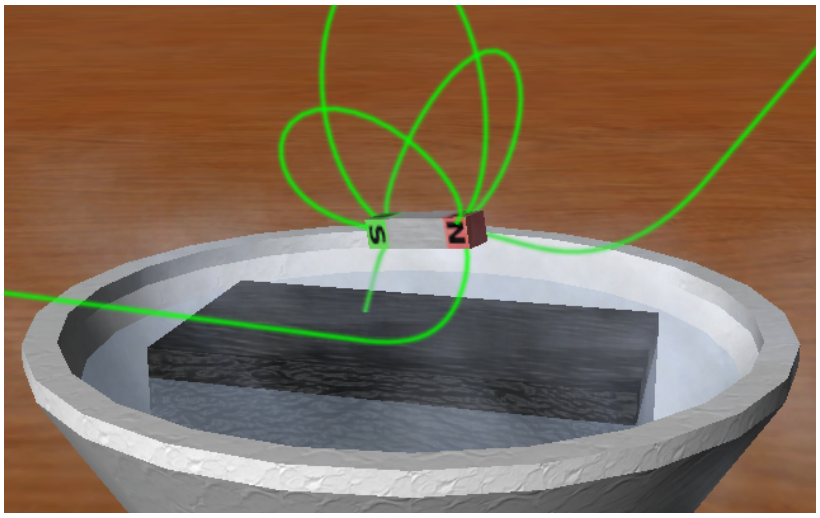

Figure 13: Magnet hovering over a superconductor.

it moves its magnetic field closer to the superconductor and the superconductor responds with its own fields to cancel everything on the surface. The magnet feels this response and the derivative of the new magnetic field applies an upward force. At some distance that depends on the strength of the magnet, this upward force balances gravity. The magnet bounces around this equilibrium point until air friction brings it to rest.

I want to stress that this is a first principles derivation, not a cheat! If the FieldLines are active, you'll see that most of them bend away from the superconductor. For some orientations, one FieldLine points directly into the superconductor surface. This is a limitation of using field lines to visualize fields: the field direction points into the surface, but the strength of the field goes to zero. Field lines only show direction, not the strength of the field.

It is important to give the magnet a substantial amount of air friction to stabilize the calculation. If the magnet is allowed to fall too far, it feels a very large force and bounces away too fast, which causes it to fall farther the next time. Without any air resistance, I got three or four good bounces before enough numerical error crept in to make it bounce away wildly. I increased the Rigidbody `drag` and `angularDrag` until the simulation was just barely stable: errors do not seem to grow, but the air friction is low enough to let it bounce for about a minute before coming to rest at its equilibrium height. You can spin it with the mouse to make it exciting again.

If you have Vectrosity and FieldLines, the SuperconductorTutorial2 shows the opposite configuration: a SuperconductorSmallSphere above a large magnet. FieldLines are required because that's the only realistic part of this second tutorial. SuperconductorSmallSphere does not respond to magnetic forces, so it can't be made to levitate realistically as the bar magnet did. It is suspended in the air by a simple SpringJoint. The value of this tutorial is that the FieldLines wrap around the ball, showing the Meissner effect very clearly (Fig 14).

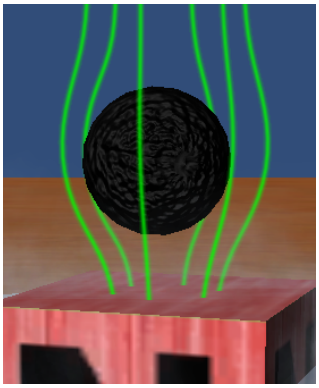

Figure 14: Expelling field lines.

The first time you run it, the FieldLines are straight and pass through the ball because the SuperconductorSmallSphere hasn't been added yet. Drag a SuperconductorSmallSphere into the SuperconductorBall object as you would with a MagneticDipole. Make sure that the SuperconductorSmallSphere is nested within the SuperconductorBall in the Hierarchy view (or else it won't move with the ball). In principle, you should also make sure that the radius is the right size by scaling it, but in this example, the two spheres are already the same size. Now turn off its MeshRenderer and run the simulation. The field lines wrap around the ball to avoid touching it. As the ball bounces around, the field lines always get out of its way.

2.6 ExternalFieldConstant and ExternalFieldSoleniod

2.6.1 Reference

External fields give you more flexibility in creating arbitrary electric and magnetic field configurations, but they are not pushed or pulled by electromagnetic forces. External field objects have a translucent MeshCollider and a script to tell the ElectromagneticFieldController how to calculate the desired field distribution. You can position the object in the Scene view, rotating and scaling it as needed, and then disable the MeshCollider to make it invisible (assuming you want to build the field-generator's visual representation out of other GameObjects). External fields can be moved and nested in GameObjects to make the field mobile as well.

Two examples have been implemented, ExternalFieldConstant and ExternalFieldSolenoid. An ExternalFieldTemplate is provided to help you write your own (see the Scripting Reference). These are the ExternalFieldConstant parameters:

Name	Type	Default	Description
`electromagnetic-FieldController`	`GameObject`	`null`	pointer to the scene's ElectromagneticFieldController; only needed if you changed its name
`electricField`	`Vector3`	`Vector3.zero`	constant electric field value
`magneticField`	`Vector3`	`Vector3.zero`	constant magnetic field value
`onlyWithinBox`	`bool`	`true`	if `true`, the ExternalFieldConstant does not contribute outside its box shape

And these are the ExternalFieldSolenoid parameters:

Name	Type	Default	Description
`electromagnetic-FieldController`	`GameObject`	`null`	pointer to the scene's ElectromagneticFieldController; only needed if you changed its name
`strength`	`float`	`1.0`	electric current in the solenoid loop in Amperes, summed over all windings

2.6.2 Discussion

External fields could be represented as an assembly of many MagneticDipoles and StaticCharges, but only approximately and at a high computational cost. They are for the case where you have an equation in your physics book that you want to put in your game, and you just want to type it in and be done with it.

The primary limitation of external fields is that they are not pushed or pulled by electric or magnetic forces, as a real field-generating object would be. This is because they are not infinitesimal sources— the total force on a field-generating object involves an integral over a broad distribution of points, implemented in a computer as an expensive sum over many tiny volumes. (For instance, one could calculate the energy in the magnetic field at all points in

space and take derivatives to find the forces.) The purpose of Magnetodynamics is to avoid heavy calculations like this that would slow down a game.

Since external fields excel at describing distributed shapes and because forces are not automatically applied to them, they are most likely to be useful as massive, unmoving field-generators in your Scene. However, they can be moved, and if they are attached to a Rigidbody, you can apply forces to them. If you're working within a special case and you know what force to apply without having to do expensive calculations, you could apply that force yourself.

ExternalFieldSolenoid describes a magnetic field configuration that is used often in science and engineering. A solenoid is a cylindrical coil of wire carrying an electric current in a spiral around the core of the cylinder. The magnetic field due to the flowing electrons in the wire adds up and cancels in just the right way to produce an almost constant magnetic field inside the cylinder and almost no magnetic field outside. For an infinitely long solenoid, this is exact. ExternalFieldSolenoid implements a finite-length solenoid. (It's surprisingly complicated: take a look inside ExternalFieldSolenoidScript!)

The `strength` parameter of ExternalFieldSolenoid includes the number of windings. If you want to represent the field due to a wire wrapped twice as many times around a cylinder (even though it's the same wire and the same current going around and around), double the `strength`.

Solenoids illustrate an important fact about magnetic forces on dipoles and induced dipoles in magnetizable materials. The force on a dipole depends on the derivative of the magnetic field (Eqn 3), so the nearly constant field (nearly zero derivative) in the middle of a solenoid applies very little force, though it can apply a huge torque (Eqn 2). The only part of the solenoid that has large field derivatives is the so-called "fringe field" near the openings. You can see this in the TabletopExperiments demo (Fig 2) if you drag a bar magnet up to the mouth of the solenoid. If the bar magnet is pointing the right way, the solenoid gobbles it in, but doesn't fling it through to the other side. Once inside, it stops. (If it's pointing the wrong way, the solenoid will happily right it for you.)

This means that you can levitate bar magnets in an upward-pointing solenoid, but not in the middle where the field is constant, only at the opening. If the magnets fall below this "sweet spot," they'll fall out of the trap.

2.6.3 Tutorial project: Beam through Solenoid

Big solenoids are often used in particle physics because the magnetic field curves the trajectory of the particles, and since it's a nearly constant field, the radius of bending is directly related to the momentum (or speed) of the particles. Thus, we can measure momenta of particles that are too fast to track with a timer by looking at the shape of its trajectory.

Open and run the SolenoidTutorial Scene. The wire and toilet paper roll solenoid from TabletopExperiments is close to the camera, filling much of the view (Fig 15). The handheld accelerator is in the distance, shooting particles through the solenoid, but not perfectly down the center. The particles spiral while they're in the solenoid, and the tightness of the spiral depends on the strength of the magnetic field. In the "fringe field" region just outside

the openings, the spiral has a wide radius. In the center, where the field is stronger, the spiral is tighter.

While the demo is running, you can rotate the camera's view by dragging the mouse across the screen. I didn't include any in-game handles to move the solenoid, but you can move it in the Scene view. The field will move along with the GameObject in real time.

If you're feeling ambitious, try turning the ExternalFieldSolenoid upright and adding a bar magnet. If the bar magnet has a small enough mass and a strong enough field, it should levitate.

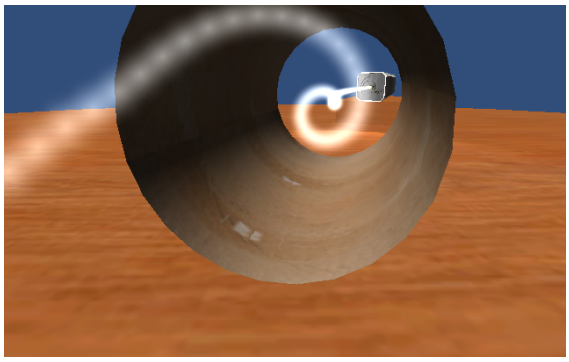

Figure 15: Down the barrel of a solenoid.

2.7 FieldLine (requires Vectrosity)

2.7.1 Reference

FieldLines help you to visualize the electric and magnetic fields by tracing a curve that is always parallel with the field vectors. They won't work without the Vectrosity package, which can be obtained on the Asset Store (`http://starscenesoftware.com/vectrosity.html`). They also won't work unless they are explicitly activated: open

`Magnetodynamics/ObjectsToPlaceInYourScene/_ScriptsThatMakeThemWork/FieldLineScript.cs`

and remove the '`//`' before '`#define I_HAVE_VECTROSITY`' (the first line in the file). If you don't do this, FieldLines in your scene will raise exceptions at start-up. If you want to have FieldLines in your Scene but don't want to see these messages, add a '`//`' before '`#define SHOW_VECTROSITY_ERRORS`'. You need to do this each time you import Magnetodynamics into a new project.

The position of a FieldLine GameObject represents the starting point of the curve. Its direction is not relevant because the direction is entirely determined by the field vectors. FieldLines change with the field in real time, can apply to electric or magnetic fields, and the algorithm that follows the field is highly configurable. The parameters are:

Name	Type	Default	Description
`electromagnetic-FieldController`	`Game-Object`	`null`	pointer to the scene's ElectromagneticFieldController; only needed if you changed its name
`onlyCalculateOnce`	`bool`	`false`	if `true`, only calculate the field line once (fields are static)

Name	Type	Default	Description
`whichField`	`enum`	`Magnetic`	if `Electric`, follow electric field lines; if `Magnetic`, follow magnetic field lines
`numberOfSegments`	`int`	`32`	maximum number of line segments
`segmentLength`	`float`	`0.1`	length of a line segment; if negative, follow the field backward
`accuracy`	`enum`	`SecondOrderMethod`	if `EulersMethod` is fast but inaccurate, `SecondOrderMethod` is in between, and `RungaKuttaMethod` is slow but good
`backTolerance`	`float`	`0.75`	stop computing the curve if the cosine of the angle between two segments is less than `backTolerance` (less than `0.75` is larger than 41.4° angles; keeps a line from becoming jagged near zero field); to disable, set to `-1`
`stopTolerance`	`float`	`0.95`	stop computing the curve if two points are within `stopTolerance` times `segmentLength` of each other (avoids looping); to disable (it costs $\mathcal{O}(\texttt{numberOfSegments}^2)$), set to `-1`
`lineMaterial`	`Material`	`null`	material for rendering the line
`color`	`Color`	`green`	color of the line
`thickness`	`float`	`8.0`	thickness of the line (a material with a fuzzy edge makes the actual thickness somewhat less than `thickness`)

2.7.2 Discussion

A FieldLine effectively solves a differential equation in real-time, once per `FixedUpdate`. It can therefore be an expensive algorithm, which is why it has so many parameters— you can tune it to balance speed and accuracy. The `calculateOnlyOnce` parameter should be used if you know that your fields and FieldLine starting points are unchanging, so that processing time is not wasted by re-calculating the same lines over and over.

The length of the line is `numberOfSegments` times the absolute value of `segmentLength` unless the algorithm gave up because of `backTolerance` or `stopTolerance`. Segments are straight, so if you want a smoother line, you'll need a higher `numberOfSegments`.

Errors accumulate along the line: if it gets a little bit off at the beginning of its run, it can be very far off at the end. The `accuracy` parameter lets you choose an algorithm to reduce these errors. The `EulersMethod` is first-order, meaning that errors are $\mathcal{O}(\texttt{segmentLength})$ and the field is calculated `numberOfSegments` times. The `SecondOrderMethod` has $\mathcal{O}(\texttt{segmentLength}^2)$ errors (much smaller) and the field is calculated $2 \times$ `numberOfSegments` times. The `RungaKuttaMethod` is fourth-order: errors are $\mathcal{O}(\texttt{segmentLength}^4)$ and the field is calculated $4 \times$ `numberOfSegments` times.

If patches of zero field are encountered, the FieldLine continues in a straight line across the patch without changing in length. If the direction changes abruptly, the line stops. Numerical error associated with large cancellations (close to the surface of a superconductor, for instance) and singularities (close to a MagneticDipole or StaticCharge, for instance) cause abrupt changes in direction. The `backTolerance` is intended to identify this situation: it is a limit on the dot product of two neighboring segments— that is, the cosine of the angle between the two segments. If the cosine is larger than `backTolerance`, the line will stop. The default `backTolerance` corresponds to a maximum angle of 41.4°. To deactivate this check, set it to −1.

Correctly calculated magnetic field lines are complete loops, returning to the point where they started (though they may need to be infinitely long to do so). In a numerical calculation, this can lead to overlapping lines: the FieldLine makes a full orbit and then continues where it started. But the accumulation of error makes this inexact and we get two very close lines, which doesn't look good. The `stopTolerance` is intended to identify a full loop and cut it off. If a FieldLine reaches a point that is within `stopTolerance` times the absolute value of `segmentLength` of a point it has been before, it will stop. To do this check, the algorithm has to compute (at most) `numberOfSegments` times `numberOfSegments - 1` distances, which can be expensive if `numberOfSegments` is large.

The `lineMaterial`, `color`, and `thickness` are parameters passed to Vectrosity for drawing. Lines are rendered with with Vectrosity's 3-D method, so they can be hidden behind objects or partly inside of objects.

A good strategy for setting up field lines is to evenly space the starting points with half of them growing in the positive direction (positive `segmentLength`) and the other half growing in the negative direction (negative `segmentLength`). The tutorial shows how to set this up.

2.7.3 Tutorial project: Geomagnetism

Open and run the GeomagnetismTutorial Scene. It contains a sphere painted like the Earth, rotating with a nice tilt. In the center of the sphere is a MagneticDipole, but you wouldn't know it because there's nothing around to be affected by its field.

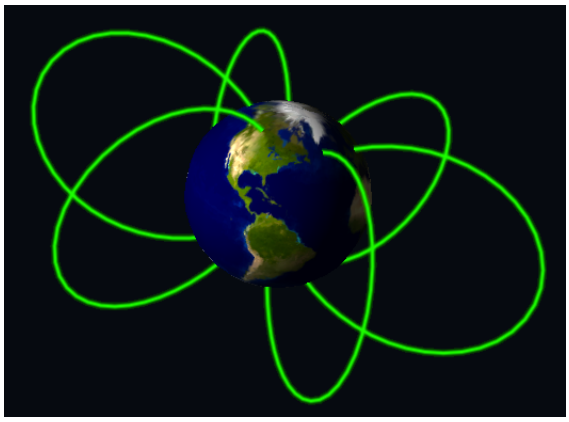

Figure 16: Field lines through the Earth.

Drag a FieldLine into the Scene and run it again. If the FieldLine is close enough to the Earth to feel its field, it will follow the curve. You can move the position of the FieldLine in the Scene view while the game is running to see how its path depends on starting point. (It could be useful to arrange your Scene and Game views side-by-side, since FieldLines don't show up at all angles in the Scene view.)

You can attach the FieldLines to the Earth so that they rotate with it by nesting them

in the Hierarchy view (just as you might do with a MagneticDipole). Put the FieldLine into the Earth (which is inside Tilt) and make five copies of it (a total of six FieldLines). Try to arrange the positions uniformly by hand, or use the local positions tabulated below.

segmentLength	**x**	**y**	**z**
0.1	0.	0.33	0.2
0.1	0.173	0.33	−0.1
0.1	−0.173	0.33	−0.1
−0.1	0.173	−0.33	0.1
−0.1	0.	−0.33	−0.2
−0.1	−0.173	−0.33	0.1

This configuration is an equilateral triangle near the Earth's north pole and another equilateral triangle, rotated by 60 degrees, near the Earth's south pole. If you don't set the `segmentLengths` (that is, if they're all positive), the FieldLines from the southern triangle will try to go through the Earth and come out the top, but they'll be get lost in the MagneticDipole singularity. It's always best to start lines going *away* from the nearest MagneticDipole (or StaticCharge for `whichField = Electric`). It's also nice to have as many backward-going lines as forward-going lines so that the numerical errors in the curves are symmetric and the user is less likely to notice them.

Now when you run it, it should look like Fig. 16. The real magnetic field in the Earth doesn't point exactly north, so try tilting the MagneticDipole a little bit. Five to ten degrees makes a big difference in the lines (unless you tilt the lines by the same amount, which would be another reasonable thing to do). If the FieldLines are no longer closed, make them longer by extending `segmentLength` or adding more `numberOfSegments`. (They don't have to be powers of two; that's just what I chose for defaults.)

2.8 PlasmaStream and PlasmaIonizedGas

2.8.1 Reference

Magnetodynamics can simulate low-density plasmas that are affected by electric and magnetic fields but do not produce their own fields (which would be computationally expensive). They are modifications of Unity's ParticleSystem to respond to fields through the Lorentz force.

A prefabricated ParticleSystem with an associated script, PlasmaIonizedGas, has been provided. You may want to make a lot of modifications to this ParticleSystem so that it fits your artistic vision— if you already have a ParticleSystem that you like, just attach the PlasmaIonizedGas script to it directly.

There are two configuration parameters that you must check when you add the script. The ParticleSystem's `simulationSpace` must be `World`, not `Local`, and the ParticleSystem's `maxParticles` must be equal to the PlasmaIonizedGas `maxParticles`. The ParticleSystem takes up a lot of space in the Inspector, especially with its Particle System Curves window,

so you may need to scroll down to see the PlasmaIonizedGas parameters. Here are the parameters' definitions:

Name	Type	Default	Description
`electromagnetic-FieldController`	`GameObject`	`null`	pointer to the scene's ElectromagneticFieldController; only needed if you changed its name
`chargeOverMass`	`float`	`1000.0`	charge-over-mass in Coulombs per kilogram (C/kg) of the Particles
`maxParticles`	`int`	`256`	number of Particles to expect in the ParticleSystem

2.8.2 Discussion

A plasma is a state of matter like a gas, but in which some or all of the particles are ionized (charged). They could be fundamental particles like electrons or they could be atoms that are missing one or more electron. Dense plasmas are notoriously hard to simulate (often for nuclear fusion research) because they can interact with themselves. They generate their own electric fields by virtue of being charged and generate magnetic fields when those charges move at high speeds. The shape of the plasma blob is affected by those new fields, which generate more fields, etc.

To avoid that complexity, PlasmaIonizedGas does not generate any fields, it only reacts to them. To emphasize this lack of self-interaction, I've included "ionized gas" in its name. (Ideal gasses do not interact with themselves at all.) All of the Particles in the ParticleSystem are accelerated by the Lorentz force $\vec{F}_L$

$$\frac{\vec{F}_L}{m} = \frac{q}{m}\Big(\vec{E}(\vec{x}) + \vec{v} \times \vec{B}(\vec{x})\Big) \tag{10}$$

where m is a Particle's mass, q is its charge (so the `chargeOverMass` parameter is q/m in Coulombs per kilogram, C/kg), $\vec{v}$ is its velocity, $\vec{E}(\vec{x})$ is the electric field at the Particle's position, and $\vec{B}(\vec{x})$ is the magnetic field at the same position. This causes Particles to fall in the direction of electric field lines and spiral around magnetic field lines.

The number of particles in a typical ParticleSystem (hundreds) is in no way realistic—generally, the ParticleSystem is intended as a visual representation of more particles than the game system can actually track. Artistically, it's a good idea to make the Particle size larger than the typical spacing so that it's easier to believe that the blob is a cloud, rather than fairies.

The `maxParticles` parameter is needed to allocate an array in which the Particles are stored while they're being operated on. The wrong value could lead to index errors.

The force parameters in Unity's ParticleSystem (such as Force Over Lifetime, etc.) won't work because PlasmaIonizedGas completely takes control of the Particle's velocities. The other parameters (color, size, etc.) still work.

Particles should orbit around a strong StaticCharge (see the VanDeGraaffExample Scene) and they should circulate in a constant magnetic field, such as the field in a solenoid (see

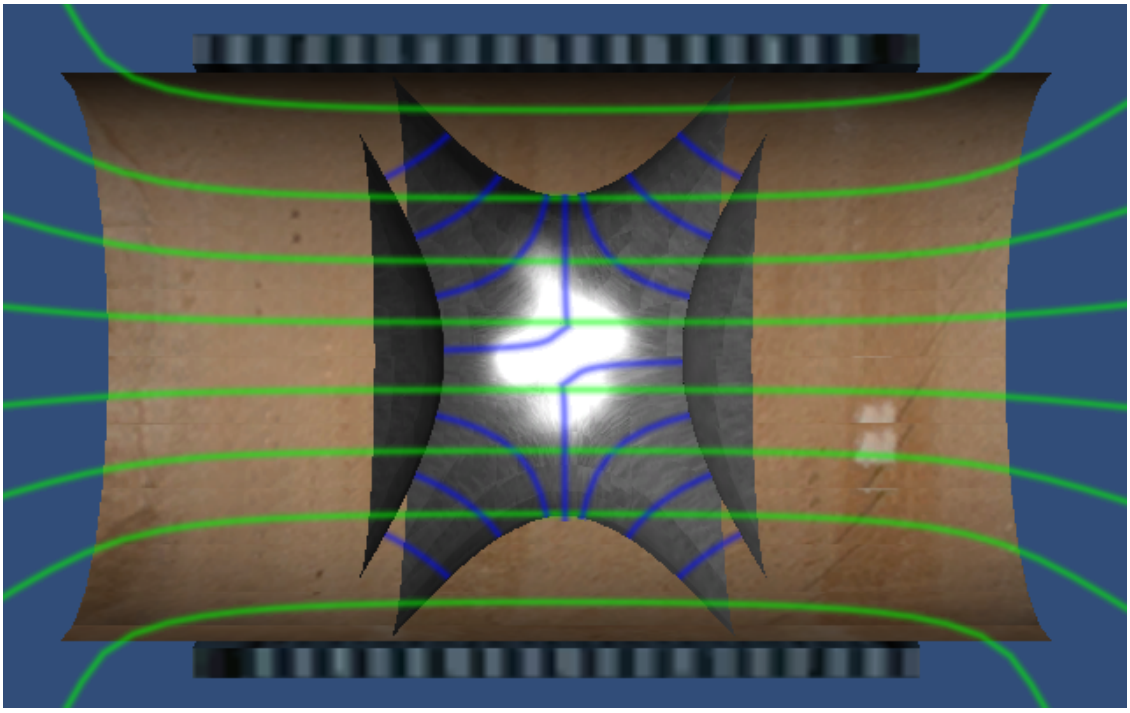

Figure 17: Antimatter (white) caught in a Penning trap. Electric field lines are blue, magnetic field lines are green. You're looking at a side view of the apparatus, cut in half (the mouse allows you to rotate).

the SolenoidTutorial or the tutorial project below), but long-lived Particles and strong fields can accumulate numerical errors and fly away. This sort of problem was solved by damping with air resistance in the SuperconductorTutorial, but air resistance doesn't make sense for ions and electrons. (Maybe someday I'll implement bremsstrahlung.) For ParticleSystems, you can reduce the Particle's lifetime so that the error-tainted Particles are replenished by fresh Particles. If the Particles' size is larger than their typical separations, the Particle-replacement is hard to notice. Reducing the field strengths or `chargeOverMass` also works, though it also weakens the effect, which might not be desired.

To use the same terminology as in the FieldLine implementation, the Particles' trajectories are differential equations solved by Euler's Method. The equivalent of a `segmentLength` is the distance that a Particle travels in one time-step.

2.8.3 Tutorial project: Antimatter in a Penning Trap

Open and run the PenningTrapTutorial. Penning traps are devices used to store charged particles without letting them touch the walls. They are often used to hold batches of charged antimatter, such as antiprotons and positrons, for experiments. The electric and magnetic fields are already set up, but the ParticleSystem ignores them because it hasn't been declared as a plasma yet. Drag a PlasmaIonizedGas script into the Antimatter object and set the

`chargeOverMass` to 1000 C/kg. Now when you re-run it, the Particles are trapped (Fig 17). Note that Unity provides a preview of what the ParticleSystem would look like before you actually run it— this preview doesn't include the electromagnetic forces because the script isn't active. When the game is running, rotate the camera angle by dragging across the screen with the mouse.

The Penning trap requires both electric and magnetic fields to work. The electric field is produced by a hyperboloid-shaped tube and two hyperboloid endcaps held at high voltage (the endcaps have the same charge as the particles, the barrel has the opposite charge). The blue FieldLines indicate the direction of the electric field due to these plates (see how the blue lines touch the metal surfaces at right angles?). Particles want to follow these lines, away from the endcaps and toward the sides of the tube: in other words, inward horizontally and away from the center vertically. The magnetic field is produced by a solenoid, indicated by green lines. Particles want to circulate around the magnetic field lines, which opposes the outward force of the electric field. Without the magnetic field, they'd be squeezed out the middle, and without the electric field, they'd spiral out of the ends. Look at the antimatter blob from an angle to see how it squeezes flat and spirals around like a galaxy.

Select the ExternalFieldSolenoid and uncheck its ExternalFieldSolenoidScript to see what would happen without a magnetic field. Then reinstate it, select the ExternalFieldElectricQuadrupole and uncheck its ExternalFieldElectricQuadrupoleScript to see what would happen with a magnetic field but no electric field.

To see the effects of numerical error, try doubling the PlasmaIonizedGas `chargeOverMass` (to 2000 C/kg). This makes the effect stronger, so at first the blob is squeezed faster and tighter, but it pretty quickly spins out of control. At these higher velocities, the Particles are taking larger distance steps in each time-step, and the errors in Euler's Method grow linearly with this step size. You can compensate this by halving the Particle lifetime (in the ParticleSystem's controls: change the `startLifetime` from 10 to 5).

3 Scripting Reference

For all the uses described above, you don't need to write any code. However, maybe you *want* to.

Though Unityscript (Unity's "Javascript") may be easier to write, it is easier to set up C# scripts. Unityscript has trouble finding the ElectromagneticFieldControllerScript definition, even when you configure a Script Execution Order (under Edit → Project Settings). If you're going to use Unityscript, I recommend moving the whole Magnetodynamics folder into a top-level folder called "Standard Assets" (must be done in every project that you import Magnetodynamics into). If you're using C#, you don't need to do this.

3.1 How to access the magnetic and electric fields

To access field values, you need to call to the ElectromagneticFieldControllerScript. First, make a reference to it in your `Start()` function.

(C#)
```
private ElectromagneticFieldControllerScript controller;
void Start() {
    GameObject efc;
    efc = GameObject.Find("ElectromagneticFieldController");
    controller = efc.GetComponent<ElectromagneticFieldControllerScript>();
}
```

(Unityscript)
```
var controller : ElectromagneticFieldControllerScript;
function Start() {
    var efc : GameObject;
    efc = GameObject.Find("ElectromagneticFieldController");
    controller = efc.GetComponent(ElectromagneticFieldControllerScript);
}
```

Now you can call `ElectricField`, `MagneticField`, and `MagneticFieldDerivatives` as often as you like to get the field values or their derivatives in global (world) coordinates. (The components of the vectors are global *x, y, z* and the derivatives are with respect to global *x, y, z*.)

(C#)
```
Vector3 pos = whatever;
Vector3 E = controller.ElectricField(pos);
Vector3 B = controller.MagneticField(pos);
Vector3 curlB;
Vector3 ddxB;
Vector3 ddyB;
Vector3 ddzB;
controller.MagneticFieldDerivatives(pos, out curlB, out ddxB, out ddyB, out ddzB);
```

Unityscript requires additional parameters that are supposed to be hidden in the public API. Use the values specified below (`rigidbodyId = -1` and `excludeSuperconductors = 0`).

```
var pos : Vector3 = whatever;                                          (Unityscript)
var E : Vector3 = controller.ElectricField(pos, -1, 0);
var B : Vector3 = controller.MagneticField(pos, -1, 0);
var curlB : Vector3;
var ddxB : Vector3;
var ddyB : Vector3;
var ddzB : Vector3;
controller.MagneticFieldDerivatives(pos, curlB, ddxB, ddyB, ddzB, -1, 0);
```

These fields include contributions from all StaticCharges, MagneticDipoles, external fields, and superconductors. Since MagnetizableMaterials don't create induced fields (they just experience forces as though they had induced fields), there are no contributions from them. The derivatives are 3-D vectors:

$$\texttt{ddxB} = \frac{d}{dx}\vec{B}, \texttt{ddyB} = \frac{d}{dy}\vec{B}, \texttt{ddzB} = \frac{d}{dz}\vec{B} \tag{11}$$

and

$$\texttt{curlB} = \nabla \times \vec{B} = \texttt{ddyB.z} - \texttt{ddzB.y}, \texttt{ddzB.x} - \texttt{ddxB.z}, \texttt{ddxB.y} - \texttt{ddyB.x} \tag{12}$$

is calculated from `ddxB`, `ddyB`, and `ddzB`.

3.2 How to change field strengths through a script

You may want to use a script to change the strength of a field so that you can turn it on or off in response to some action. Suppose that you have a MagneticDipole, a StaticCharge, a GameObject with MagnetizableMaterial, or an ExternalFieldSolenoid named `obj` (not the parent of one of these objects, but the object itself). You can change its strength with

```
obj.GetComponent<XXX>().strength = value;                              (C#)
obj.GetComponent(XXX).strength = value;                                (Unityscript)
```

where `XXX` is one of the types in the table below.

Type	**Default**	**Units**
MagneticDipoleScript	300.0	$\text{A}\cdot\text{m}^2$
StaticChargeScript	10^{-8}	C
MagnetizableMaterial	10^{-14}	$\chi/(1+\chi)$ times volume
ExternalFieldSolenoidScript	1.0	A

ExternalFieldConstantScript has an `electricField` (V/m) and a `magneticField` (T) that can be changed at run-time. Superconductors have no `strength` parameters because they just respond to the field around them. The `PlasmaIonizedGas` script has a `chargeOverMass` (default: 1000 C/kg) and `FieldLineScript` has a `segmentLength` (default: 0.1 distance units) that can also be changed at run-time.

3.3 Writing your own external field

You can add your own external field objects by writing a script that gives a field function to the ElectromagneticFieldControllerScript. See above for how to access the ElectromagneticFieldControllerScript or just make a copy of ExternalFieldTemplateScript. The field functions

```
Vector3 ElectricField(Vector3 pos) { ... }                    (C#)
Vector3 MagneticField(Vector3 pos) { ... }
Vector3 MagneticFieldDx(Vector3 pos) { ... }
Vector3 MagneticFieldDy(Vector3 pos) { ... }
Vector3 MagneticFieldDz(Vector3 pos) { ... }
```

should return the electric field, magnetic field, or magnetic field derivatives in global coordinates, given a point in global coordinates. Not all of them are required. If you want to see field lines, implement `ElectricField` or `MagneticField`. If you want to apply a torque on MagneticDipoles, implement `MagneticField`. If you want to apply a force on MagneticDipoles, implement the derivatives. If you want to apply a force on MagnetizableMaterial, implement the `MagneticField` and its derivatives. The derivative functions are always called as a group: `MagneticFieldDx`, then `MagneticFieldDy`, then `MagneticFieldDz`, so you may want to cache values in your class to avoid computing them three times.

Next, you need to pass your functions to the ElectromagneticFieldControllerScript so that they will be used. Call

```
controller.RegisterElectricField(ElectricField);                (C#)
controller.RegisterMagneticField(MagneticField);
controller.RegisterMagneticFieldDerivatives(MagneticFieldDx, MagneticFieldDy,
    MagneticFieldDz);
```

in your `Start()` function or the first time it is needed. I have only tested this in C#.

Unless you know that there are no objects in your Scene that would call the field functions many times per `FixedUpdate()`, avoid putting `Debug.Log()` messages in it. The high rate of messages can lock up Unity so that you need to kill it ("Force Quit"), possibly losing work. A single FieldLine or PlasmaIonizedGas can call field calculations dozens or hundreds of times per `FixedUpdate()`. (It depends on `FieldLine.numberOfSegments` and `PlasmaIonizedGas.maxParticles`.) A MagneticDipole, StaticCharge, or MagnetizedMaterial may only call it once, but if you have dozens or hundreds of these objects, the number of calls will be large.

You can make your external field script depend on the position, orientation, and scale of an associated GameObject to make it easier to configure. (Moving the GameObject would move the field.) ExternalFieldConstant works this way (it is associated with a box), as are ExternalFieldSolenoid (a cylindrical mesh), SuperconductorInfinitePlane (a plane), and SuperconductorSmallSphere (a sphere). In these examples, the field functions first transform the input point `pos` from global to local coordinates with `transform.InverseTransformPoint(pos)`,

work in the local frame, then transform the resulting magnetic field from local to global coordinates with `transform.TransformDirection(field)`.

Transforming derivatives is more complicated. If you calculate $\frac{d}{dx}\vec{B}$, $\frac{d}{dy}\vec{B}$, and $\frac{d}{dz}\vec{B}$ in a local frame, then the components of the vector would be local and also the direction of the derivatives would be local. That requires two transformations (equivalent to a single similarity transform). Suppose you have `ddxloc_Bloc`, `ddyloc_Bloc`, and `ddzloc_Bloc` (all `Vector3`) with local components and local derivatives. You can use this method to convert them:

(C#)

```
// get the transformation matrix (Jacobian)
Vector3 right = transform.InverseTransformDirection(Vector3.right);
Vector3 up = transform.InverseTransformDirection(Vector3.up);
Vector3 forward = transform.InverseTransformDirection(Vector3.forward);
float dxloc_dxglob = right.x;
float dyloc_dxglob = right.y;
float dzloc_dxglob = right.z;
float dxloc_dyglob = up.x;
float dyloc_dyglob = up.y;
float dzloc_dyglob = up.z;
float dxloc_dzglob = forward.x;
float dyloc_dzglob = forward.y;
float dzloc_dzglob = forward.z;

// apply it to transform the derivatives
Vector3 ddxglob_Bloc = new Vector3(
    dxloc_dxglob*ddxloc_Bloc.x + dyloc_dxglob*ddyloc_Bloc.x + dzloc_dxglob*ddzloc_Bloc.x,
    dxloc_dxglob*ddxloc_Bloc.y + dyloc_dxglob*ddyloc_Bloc.y + dzloc_dxglob*ddzloc_Bloc.y,
    dxloc_dxglob*ddxloc_Bloc.z + dyloc_dxglob*ddyloc_Bloc.z + dzloc_dxglob*ddzloc_Bloc.z);

Vector3 ddyglob_Bloc = new Vector3(
    dxloc_dyglob*ddxloc_Bloc.x + dyloc_dyglob*ddyloc_Bloc.x + dzloc_dyglob*ddzloc_Bloc.x,
    dxloc_dyglob*ddxloc_Bloc.y + dyloc_dyglob*ddyloc_Bloc.y + dzloc_dyglob*ddzloc_Bloc.y,
    dxloc_dyglob*ddxloc_Bloc.z + dyloc_dyglob*ddyloc_Bloc.z + dzloc_dyglob*ddzloc_Bloc.z);

Vector3 ddzglob_Bloc = new Vector3(
    dxloc_dzglob*ddxloc_Bloc.x + dyloc_dzglob*ddyloc_Bloc.x + dzloc_dzglob*ddzloc_Bloc.x,
    dxloc_dzglob*ddxloc_Bloc.y + dyloc_dzglob*ddyloc_Bloc.y + dzloc_dzglob*ddzloc_Bloc.y,
    dxloc_dzglob*ddxloc_Bloc.z + dyloc_dzglob*ddyloc_Bloc.z + dzloc_dzglob*ddzloc_Bloc.z);

// also transform the field vector components
ddx_B = transform.TransformDirection(ddxglob_Bloc);
ddy_B = transform.TransformDirection(ddyglob_Bloc);
ddz_B = transform.TransformDirection(ddzglob_Bloc);
```

A How the code is organized

The equations for magnetic forces and fields are just copied from textbooks, but some organization is needed to apply them appropriately. Unity's scripting environment is decentralized, but every magnetic component needs to know about every other magnetic component to compute the total field and thus the force. The ElectromagneticFieldController coordinates all of that traffic— everybody reports their positions or fields to the controller and the controller applies forces and torques on everybody.

The controller also needs to keep track of which MagneticDipoles and StaticCharges are in the same Rigidbodies so that Rigidbodies do not apply forces on themselves. (See the discussion section on MagneticDipoles). The primary data structure is therefore a Dictionary of Dictionaries: the first Dictionary maps `rigidbodyIds` to a Dictionary that maps `magneticDipoleIds` to dipole positions and moment vectors ($\vec{m}$). All of these ids are unique integers (even MagneticDipoles in different Rigidbodies have distinct identifiers). This data structure mirrors the loop structure: the outer loop over Rigidbodies contains an inner loop over MagneticDipoles. This whole structure is repeated for StaticCharges. The `AssignMagneticDipoleId()` and `AssignStaticChargeId()` functions add dipoles and charges to the Dictionaries and return an id number so that the MagneticDipoles and StaticCharges know who they are.

Since the dipoles and charges are stored in Dictionaries, you can delete them at run-time. The controller also needs to keep track of the Rigidbodies that contain dipoles and charges, which it keeps in a List with list indices equal to the `rigidbodyIds`. You shouldn't delete these Rigidbodies at run-time because the List indices would get out of sync.

External fields are registered with the controller using `RegisterElectricField()`, `RegisterMagneticField()`, and `RegisterMagneticFieldDerivatives()`, as you have seen above. These take functions as arguments, stored in Lists, and executed in order with each request for the total magnetic field. There's also a superconductor version of these functions, since superconductors need to know about the total magnetic field except for other superconductors to calculate images. Superconductor images are added one at a time by recursively calling `MagneticField()` and its derivatives with `excludeSuperconductors` increasing until there are no more superconductors in the list.

Other than that, everything else is a special case. The comments in the code should help you to understand what's going on.

B What to do when something goes wrong

B.1 Nothing moves (no perceptible magnetic/electric forces)

- If there's an error in the console saying, "Could not find ElectromagneticFieldController," you need to add an ElectromagneticFieldController to your Scene. (There must be one in every Scene.)
- Are the `strength` parameters of both actors strong enough? Try raising them one order of magnitude (factor of 10) at a time.
- The Rigidbody might be blocked. Are any of the Freeze Position or Freeze Rotation constraints checked? Does the object respond to other forces, when applied?
- Is the MagneticDipole or StaticCharge complaining that it "must be a child of a GameObject," "must be a child of a GameObject that has a Rigidbody component," or "can only be embedded in objects with a BoxCollider, SphereCollider, CapsuleCollider, or a MeshCollider" in the console?

B.2 Magnetic/magnetizable objects explode or roll violently

- The MagneticDipoles may not be buried deep enough within the Collider. If something is allowed to get close to the MagneticDipole, the forces will be huge.
- The Collider might not be keeping the objects apart. MeshColliders can sometimes be bypassed by sufficiently fast objects, especially if the mesh is not air-tight. Try adding a BoxCollider or SphereCollider to reinforce the MeshCollider for high-speed impacts.

B.3 The dynamics looks right at first, but then goes crazy

- It could be the accumulation of numerical error. Try adding friction or drag (dynamic friction in the Collider's Physic material, the friction in the Collider it's touching, or the Rigidbody's `drag` and `angularDrag`).
- If it's a ParticleSystem (plasma), there is no friction to combat numerical error, but you can reduce the lifetime of Particles (ParticleSystem's `startLifetime`) or the strength of the forces (PlasmaIonizedGas's `chargeOverMass`).

B.4 Field lines do not appear/unwanted errors about FieldLines

- Have you uncommented `I_HAVE_VECTROSITY` or commented out `SHOW_VECTROSITY_ERRORS`? See the FieldLine reference about how to do this. (It must be done in every project that Magnetodynamics has been imported into. Also remember that it won't work without Vectrosity.)

- Is `segmentLength` too short for the scale of your Scene? (Or is `numberOfSegments` too small?)
- Is the FieldLine located in a position that would have it grow into an opaque object? Magnetic field lines grow in the north direction (toward south poles) and electric field lines grow in the positive direction (toward negative charges).
- Is the FieldLine located in a position that would have it grow into a singularity (MagneticDipole or StaticCharge)?
- Is the FieldLine in a patch of zero or very low field? Try increasing the strength of sources.

B.5 ParticleSystem becomes invisible when the game is running

- The `chargeOverMass` could be too high. Try reducing it one order of magnitude (factor of 10) at a time.

B.6 OverrideColliderPoof isn't expanding trigger-collider

- Are there multiple MagneticDipoles/StaticCharges in the Rigidbody? Only one of them controls the poofing of the Collider, so set the `overrideColliderPoof` of all of them.