TJHSST Computer Systems Lab
Senior Research Project

OcuPhysics: Modeling Electricity in Virtual Reality

2015-2016

Kevin Chaplin, Sherry Wang, Bernice Wu

5 June 2016

Abstract

Current methods of instructing physics students in electromagnetic concepts make use of 2D diagrams, which are ill-suited to illustrating 3D phenomena involving electrical and magnetic forces that change as charges move through space. Our project aims to supplement traditional whiteboard instruction with an interactive program that allows the user to move electrical charges and watch electrical forces update in real time.

Introduction

Point charges generate electric fields. An isolated positive charge generates a field pointing radially outwards while an isolated negative charge generates a field pointing radially inwards. The magnitude and direction of the field at each position in space relative to the collection of point charges are dictated by Maxwell's Equations. The arrangement of multiple point charges creates a complex, 3D field. Traditional representations of electric fields are limited by their 2D nature, failing to capture the third dimension. Especially for more complicated fields, the traditional method of representation is a detriment to problem solving techniques dependent upon a realistic visualization.

Virtual reality presents a solution to this problem. For years, virtual reality has been relegated to the realm of science fiction not due to lack of trying, but rather to the lack of appropriate hardware. The state of technology before 2011 prevented any kind of virtual reality simply because screens could not render fast enough to avoid nausea caused by lag greater than 20 milliseconds. LCD screen take at least 15 milliseconds to change color, so the Oculus team used AMOLED screens, which can do the same thing in less than a millisecond. Position tracking was implemented with an external camera, a gyroscope, an accelerometer, and a magnetometer in part to get the most nuanced readings possible, and in part to predict the user's motions in order to pre-render the screen and shave off precious milliseconds (Rubin).

With Facebook's acquisition of Oculus in 2014, the developers no longer had to worry about funding, so they were able to get custom parts and create the high-quality product virtual dreams are made of (Rubin). Facebook was an integral step towards viable virtual reality because unlike smartphones or personal computers, a headset must be perfect. A slow computer leads to frustration; an imperfect virtual reality headset leads to nausea and guaranteed commercial failure.

A consumer-ready version of the Oculus Rift was released on May 7th, 2016 (Oculus VR), making virtual reality available to the public only four short years after development began in earnest (Chafkin). Beyond the gaming implications of deployment--original purpose for developing the Oculus Rift (Rubin)--a commercially available headset allows projects such as ours to reach their full potential. With the hardware widely available, a 3D visualization of electric and magnetic fields becomes a valuable tool for beginning physics students to more easily comprehend these phenomena. With a more sophisticated physics engine, this could become a modeling tool for industry.

Figure 1. The new Oculus Rift.

Purpose

The greatest challenge facing beginning physics students studying electricity and magnetism is visualizing 3D phenomena with only 2D diagrams. Electric and magnetic fields exist in a 3D world but are usually represented with 2D diagrams that fail to elegantly convey the third dimension. For beginning physics students, the lack of a full representation of reality impairs their problem-solving ability since its is more difficult to consider the behavior of a 3D phenomenon that has only been represented in 2D.

Virtual reality allows users to have an immersive, interactive experience in 3D. Not only does the motion-tracking allow the user to understand the 3D nature of that which is represented, the virtual nature of the system permits updates in real time. Thus, it is possible to integrate user input in an interactive system.

By modeling electric and magnetic fields in virtual reality, we are able to overcome the limitations of 2D projections of 3D phenomena and show the way the fields change in real time. Our aim was to create an intuitive program to help beginning physics students visualize electric and magnetic fields as they exist in reality; in three dimensions instead of two, and interactive instead of static.

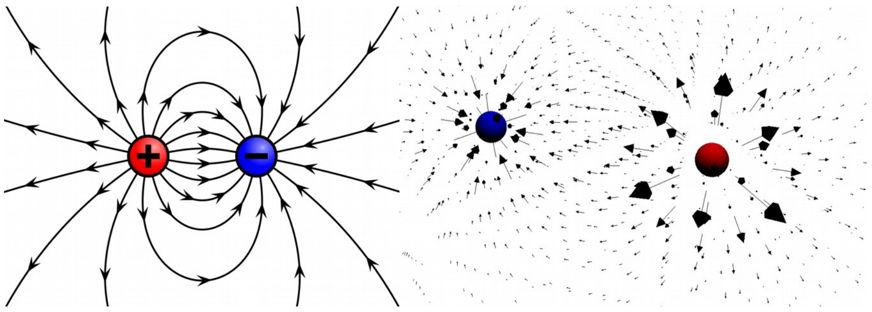


Figure 2. Electric field generated by two point charges. Left: 2D. Right: 3D.

Prior Work

The main inspiration for our project is a 3D computer-assisted design (CAD) software created by a 2015 research group comprised of Thai Le, Austin Ly, Ellis Tsung, and Jonathan Vogel. Their project combined an Oculus Rift with a Leap Motion device to create an immersive virtual reality experience and reduce the cognitive disparity between creating 3D designs in a 2D environment. We built upon their work with Leap Motion gestures in order to build our electromagnetism modeling program. As a result, our project shares features with theirs, such as the ability to select and move objects. We also used many of the same dependencies, such as node.js and Three.js.

There are 3D physics simulators that are widely used. However, most of these simulate Newtonian mechanics and are intended for industrial purposes, such as creating video games or animation, rather than for educational purposes. There are also 3D electromagnetic simulators such as MaxFem, but these are built for simulating much higher-level physics concepts than are commonly taught in high school physics classes, and are complicated to use. The high barrier to entry defeats the purpose of our project as a simple simulation for beginning physics students. Additionally, while there are physics simulators made to work with Oculus Rift and with Leap Motion such as Illustris and the Leap Motion Interaction Engine, ours is the only project that uses both.

We created our display using Three.js, a JavaScript library for creating and animating 3D computer graphics. Physics engines have been created using Three.js before, and there is a physics plugin called Physi.js, but these engines are also geared towards Newtonian mechanics rather than electromagnetism.

Figure 3. Leap Motion Sensor with hand recognition.

Procedure

The first part of the project was creating a display in Three.js to illustrate electric point charges and electric field vectors in three dimensions. The display consists of red and blue spheres, representing point charges, and white arrows, which have arrowheads of varying sizes to represent the direction and magnitude of the electric field in space. A PointCharge class was created that stores a vector, a charge value, and a radius as its fields. It also has a function called contribute_to_electric_field, which takes an array of position points chosen in order to cover a predetermined space in which the electric field will be displayed.

```
this.contribute_to_electric_field = function(field) {//field is list of position vectors
        if (this.contribution == null) {
                this.contribution = new Array(field.length);
                field.forEach(function(F, index){
                        var r = new THREE.Vector3();
                        r.subVectors(F.pos, this.vec);
                        var mag_r = r.length();
                        var e = K * this.charge / mag_r / mag_r / mag_r;
                        r.multiplyScalar(e);
                        this.contribution[index] = r;
                        F.field.add(r);
                }, this);
        } else {
                field.forEach(function(F, index) {
                        F.field.add(this.contribution[index]);
                }, this)
        }
}
```

Figure 4. contribute_to_electric_field() method.

The program finds the effect of the PointCharge's electric force on each vector in the field using Coulomb's Law, an equation that relates electric force to charge and distance between charges.

$$\vec{F} = \frac{kQ_1Q_2}{r^2}\hat{r} \tag{1}$$

From Coulomb's Law, we derived a formula for the electric field due to a single point charge.

$$\vec{E} = \frac{kQ}{r^2}\hat{r} \tag{2}$$

As each force vector at a position point is calculated, it is added to a running total that represents the overall effect of the point charges' electric field at that coordinate in the position array. A method called make_electric_field is then called over the array that creates an arrow at each position showing the electric field.

```
function make_electric_field(s, charges) {
        var field = new Array();
        for (x = -30; x <= 30; x += 5) {
                for (y = -30; y <= 30; y += 5) {
                        for (z = -15; z <= 15; z += 5) {
                                field.push(
                                {pos: new THREE.Vector3(x, y, z),
                                        field: new THREE.Vector3(0, 0, 0)});
                                // position vector and field strength vector
                        }
                }
        }
        charges.forEach(function(charge) {
                charge.contribute_to_electric_field(field);
        });
        field.forEach(function(F) {
                mag = F.field.length();
                dir = F.field.normalize();
                F.arrow = make_arrow(s, F.pos, dir, mag);
        });
        return field;
}
```

Figure 5. make_electric_field() method.

The next part of the program was animating it and adding interactivity. To distinguish interactive objects such as point charges from cosmetic objects such as vectors, we created the point charges using a library called objects.js, which was first created by the senior research group from last year and then modified by our group this year. Instead of creating a normal mesh for the spheres, objects.js creates a special mesh with code that allows gestures to select and move the mesh for animation. The bulk of the project's animation code takes place in its render() function, where it performs a set of updates if necessary and then rerenders the scene. In this function, the program checks the current position of all the PointCharge objects and then checks them against an array of the positions of the PointCharges from the last time the render() function was called. If any of the positions have changed, the function updates the array to be used for its next call and removes all the arrows from the electric field. The program then calls contribute_to_electric_field for each PointCharge and make_electric_field, thus creating the updated electric field. By doing this every time render() is called, the program display updates in real time with the movement of the point charges, creating a more fluid experience for the user and enabling the user to better understand how the positions of electric charges affect the electric field. The PointCharge's contribute_to_electric_field method is cached so that only the charge(s) that moved have their field recalculated.

```
function update_field(s, charges, field) {
        field.forEach(function(F) {
                F.field = new THREE.Vector3(0, 0, 0);
                scene.remove(F.arrow);
        });
        charges.forEach(function(charge) {
                charge.contribute_to_electric_field(field);
        });
        field.forEach(function(F) {
                mag = F.field.length();
                dir = F.field.normalize();
                F.arrow = make_arrow(s, F.pos, dir, mag);
        });
}
```

Figure 6. update_field() method.

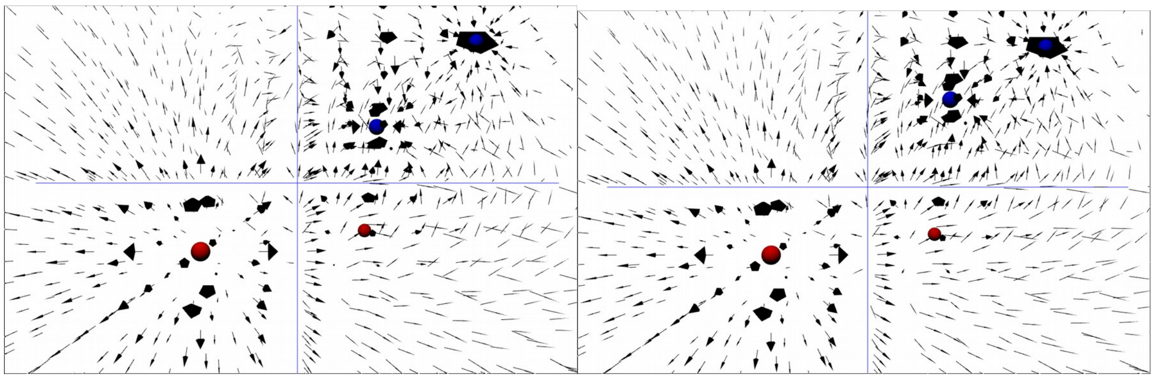


Figure 7. Electric field generated by complex arrangement of point charges, before (left) and after (right) one of the negative point charges is moved upwards.

Here, the blue point charge closest to the origin has moved up in the positive z direction, and the field has changed accordingly.

The next part of our project was learning to work with the Leap Motion Controller, a motion-sensing device that the computer used to recognize hand gestures implemented with the program. The code dealing with motion detecting is stored in several libraries: the move.js, select.js, and repeat.js libraries which deal with recognizing their respective gestures, and gestures.js, which contains the bulk of the motion detecting code and alerts main.js when a library's gesture has been recognized. We also needed a tutorial explaining the controls. Last year's senior research group created a library called tutorial.js that displays the tutorial text and moves through steps as they are completed. We altered the text contained in the tutorial and used repeat.js to add the ability to restart the tutorial at any time.

Move.js, select.js, and repeat.js have comparatively simple code compared to gestures.js. Each of these libraries' gesture is categorized into one of three types: onHand, which means the gesture only uses the right hand, offHand, which means the gesture only uses the left hand, and secondary, which means the gesture uses both hands. The main components of these libraries are the methods canActivate() and activate(). canActivate() checks if a gesture's requirements have been met. Here is select.js's canActivate() function:

```
this.canActivate = function(hand) {
        return hand.type == gestures.settings.handedness && hand.sphereRadius <
        gestures.settings.triggerSphereRadius && hand.indexFinger.extended;
};
```
Figure 8. select.js canActivate() function

select.js is an onHand gesture, as its gesture consists of pointing with the right hand. As a result, its canActivate() function takes one hand as a variable. It checks two conditions: the first being that the hand is the correct hand for the gesture type, and the second being that the hand's index finger is pointed and the rest of the fingers are in a fist, creating the pointing gesture.

activate() consists of the code for the feature intended by the activation of the gesture. select.js's activate() function is very simple, containing only a command to add a red line to the scene that aids in aiming the user's line of selection at objects. repeat.js's activate() function consists of using a listener to send a message to gestures.js that its gesture has been detected. move.js has the most complex activate() function, as it must change the scene in order to reflect the movement of point charges. Here is an excerpt from move.js's activate():

```
this.prevPosition = new THREE.Vector3();
var tip = primaryHand.data('riggedHand.mesh').fingers[1].tip.getWorldPosition(),
mcp = primaryHand.data('riggedHand.mesh').fingers[1].getWorldPosition();
this.prevPosition.copy(tip);
```

Figure 9. move.js activate() function

In this code, move.js finds and tracks the position of the user's fingers, needed in order to ensure the point charge's position changes in accordance with the user's movements.

Lastly, we needed to connect the program to an Oculus Rift. With the subtle controls provided by the sensors in the hardware and a display that fills the user's entire field of view, the Rift creates presence, the sensation of reality (Rubin). For our project, we took advantage of the motion-tracking built into the Oculus to provide the user with a way to explore the 3D fields with the same intuitive controls as with a 2D representation. The external, infrared camera tracks the movement of the Oculus while sensors within the headset provide more detailed predictions for the user's movement to render the display quickly and to adjust according to the position of the user's head (Rubin). Since these controls were already built into the Oculus SDK, we did not write further code for this portion of the hardware.

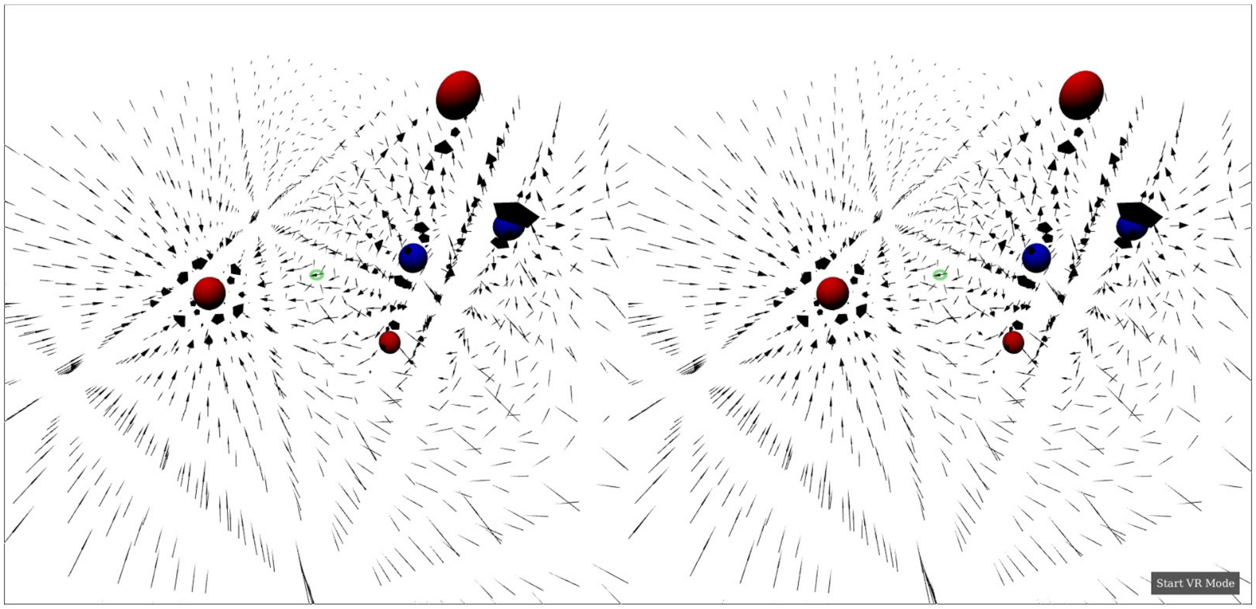


Figure 10. Point charge simulation with split screen.

Results and Conclusion

Our project was successful in that we created an interactive program that can simulate electromagnetic concepts in three dimensions. The program displays electric point charges and the electric field between charges. It also allows the user to alter the position of these point charges through the use of motion detection and will update the electric field vectors to reflect the changes in position in real time.

Our main goal in building the program, which was creating an accurate display of electromagnetic concepts, was achieved. We built our simulation using the graphics library Three.js which we chose for its ease of use. The program displays electric point charges and field lines in three dimensions, allowing users to understand the 3D shape of an electric field more easily. By implementing the Oculus Rift, the virtual reality aspect of the project creates a more immersive experience, placing users directly in the world containing the electric field. Controls that can rotate and move the camera provide a way to look at the scene from many different angles, further improving the user's experience.

Another important feature of the program is the ability to move electric point charges, facilitated by the Leap Motion device. The gestures used to select and move charges are intuitive and a tutorial is provided to ensure that users understand the device and its controls. By giving students a simple way to manipulate objects within the program, they can alter the scene to see what the electric field looks like in different situations. Additionally, the program updates the electric field as point charges are moved, meaning vectors will change as a student drags an object across the screen. By watching how objects interact with the direction and magnitude of the electric field, students can gain a better understanding of how electric forces work.

However, our end project was not a complete success. The Leap Motion device is not always accurate, sometimes reading gestures differently than the user intends. As a result, certain actions like restarting the tutorial can be difficult, as it takes time for the device to recognize the correct gesture being made. Additionally, we found that displaying too many electric field vectors slowed the program down and made movements laggy. This means we had to find a balance between displaying enough vectors to preserve the shape of the field while limiting the number of vectors to prevent lag. In addition, we only implemented electric fields generated by point charges, not magnetic fields or more complex objects.

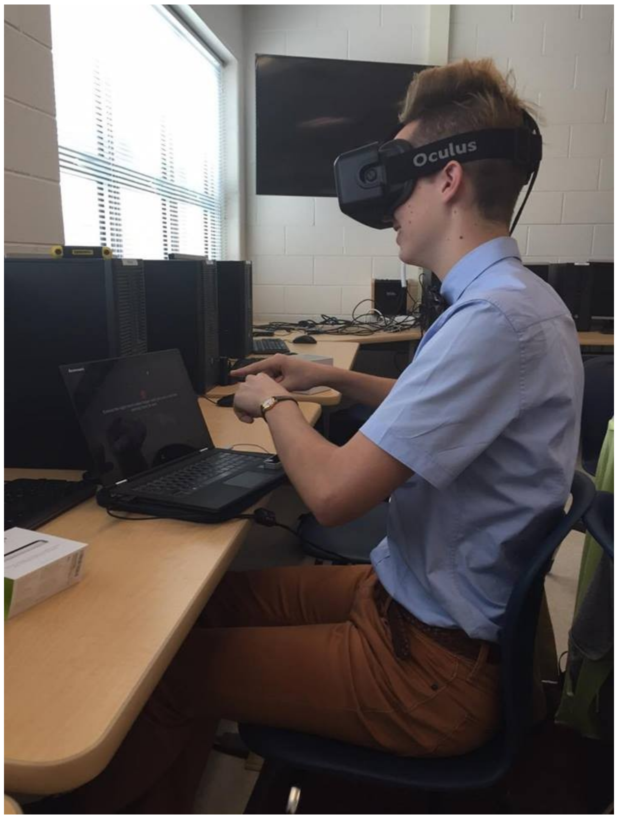


Figure 11. User using Leap and Oculus simultaneously; uses left hand fist and right hand pointing gesture to select and move a charge.

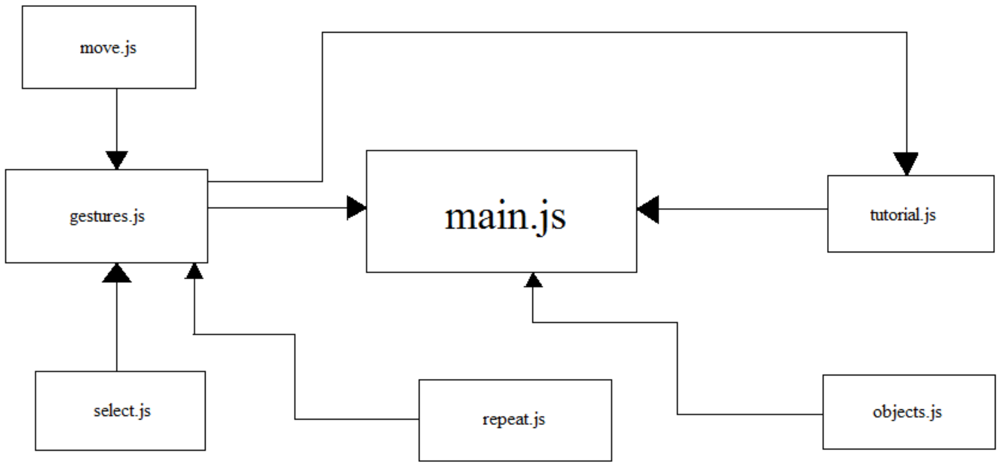


Figure 12. Architectural diagram of libraries used.

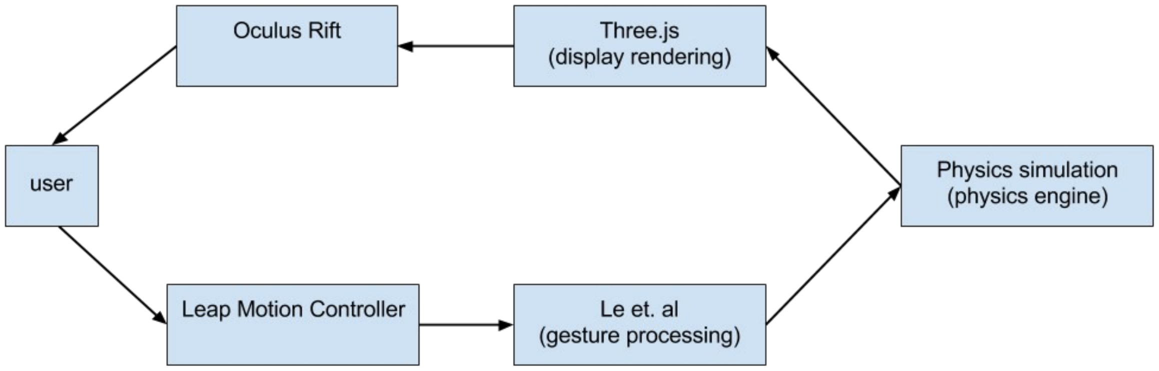
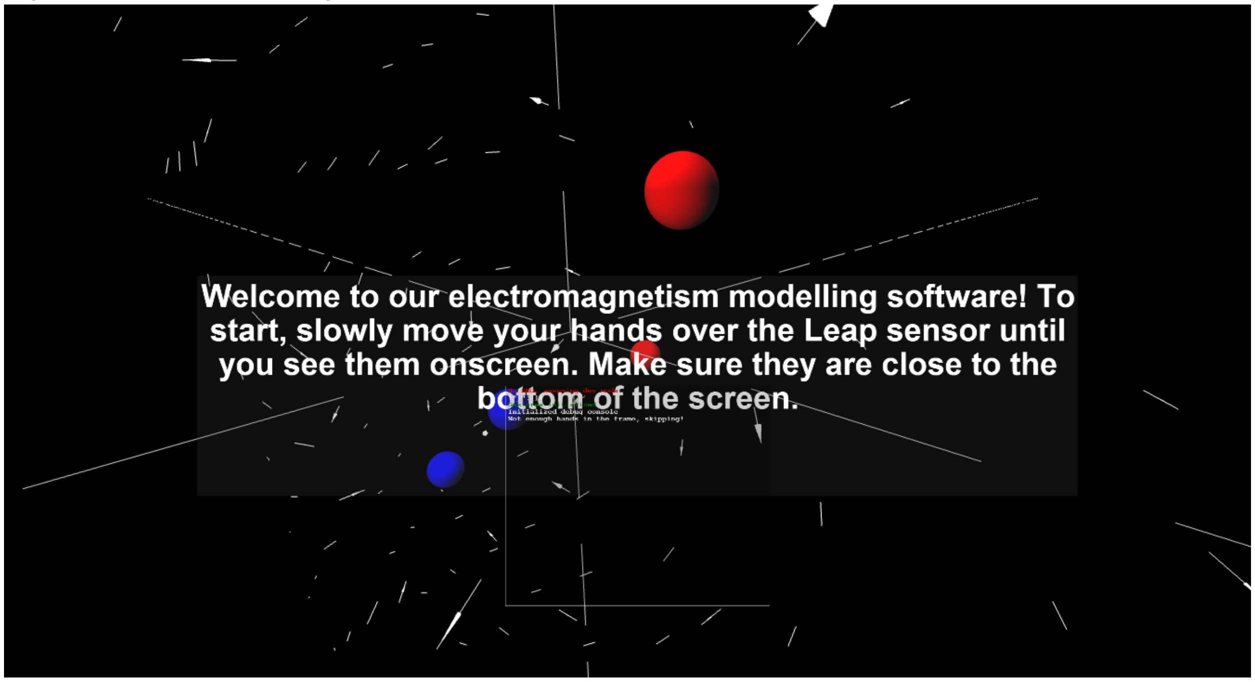


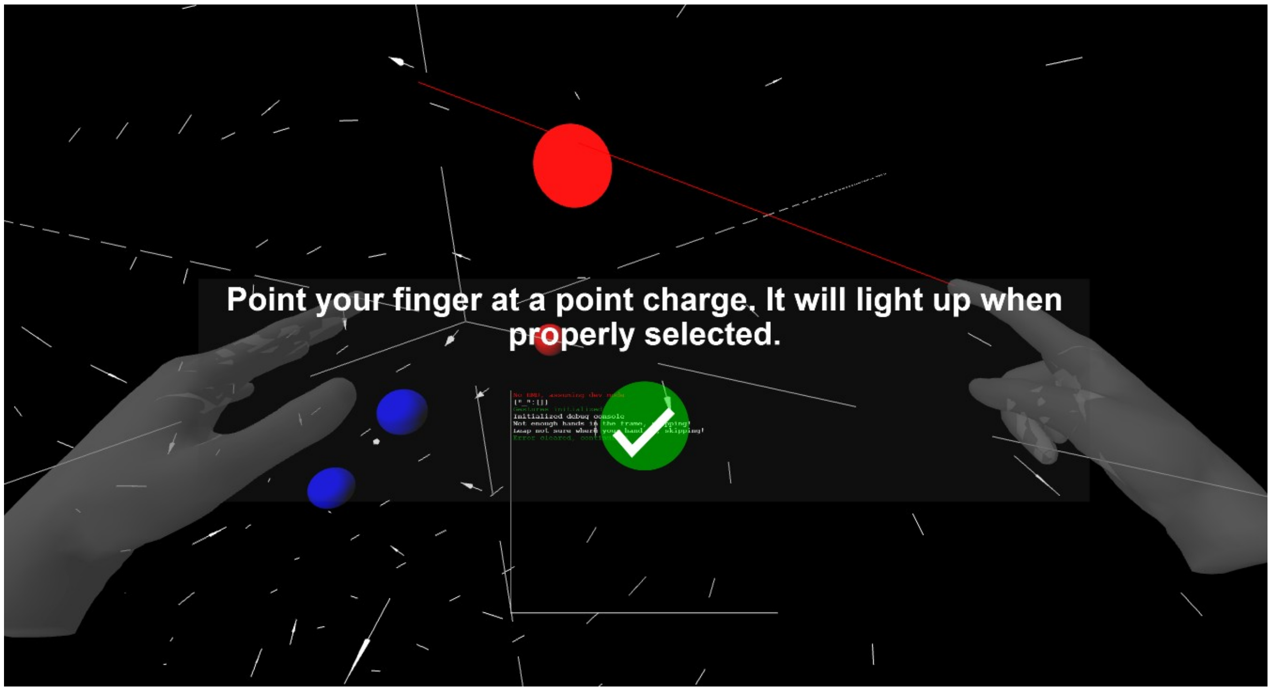
Figure 13. Architectural diagram of information flow.



Welcome to our electromagnetism modelling software! To start, slowly move your hands over the Leap sensor until you see them onscreen. Make sure they are close to the bottom of the screen.

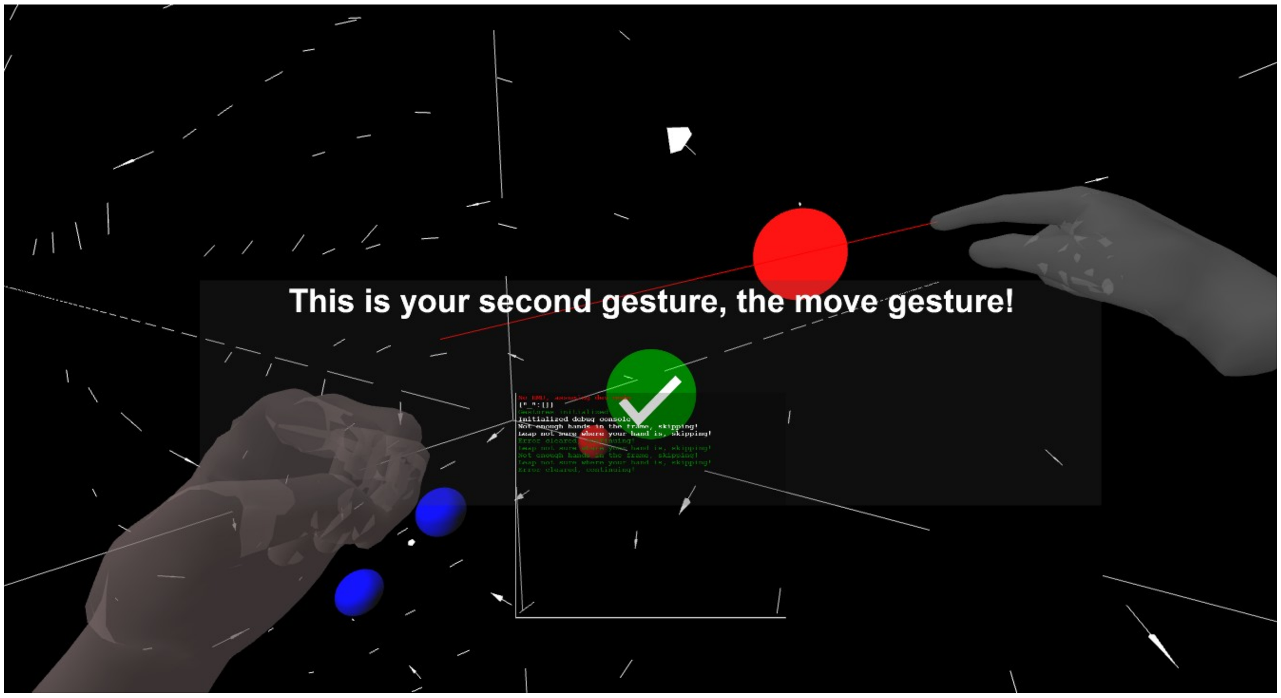
```
No [illegible], assuming dev mode
[illegible]
Gestures initialized
Initialized debug console
Not enough hands in the frame, skipping!
```

Figure 14. First step of the tutorial.


Point your finger at a point charge. It will light up when properly selected.

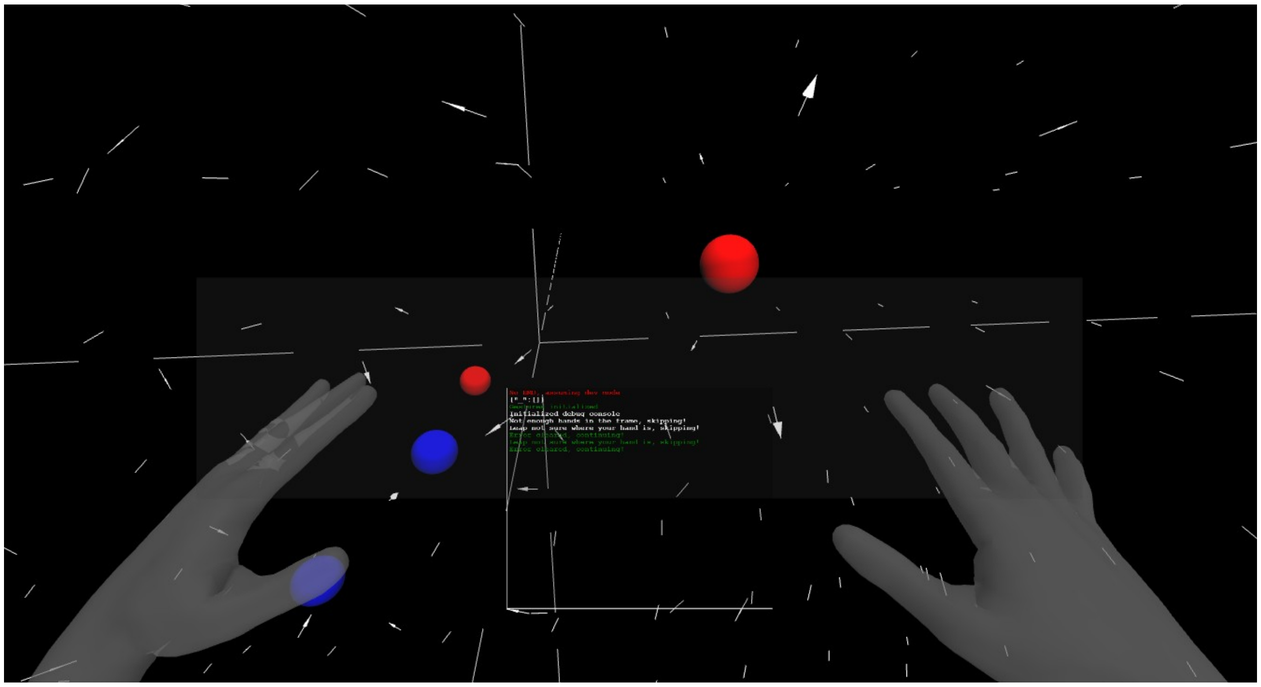
```
No URL, assuming dev mode
["_":[]]
Gestures initialized
Initialized debug console
Not enough hands in the frame, skipping!
Leap not sure where your hand is, skipping!
Error cleared, continuing!
```

Figure 15. Tutorial step explaining the “select” gesture.


This is your second gesture, the move gesture!

```
No URL, assuming dev mode
["_":[]]
Gestures initialized
Initialized debug console
Not enough hands in the frame, skipping!
Leap not sure where your hand is, skipping!
Error cleared, continuing!
Leap not sure where your hand is, skipping!
Not enough hands in the frame, skipping!
Leap not sure where your hand is, skipping!
Error cleared, continuing!
```

Figure 16. Tutorial step explaining the “move” gesture.



```
No HMD, assuming dev mode
["_":[]]
Gestures initialized
Initialized debug console
Not enough hands in the frame, skipping!
Leap not sure where your hand is, skipping!
Error cleared, continuing!
Leap not sure where your hand is, skipping!
Error cleared, continuing!
```

Figure 17. Program after the tutorial has finished.

Further Research

This project could be expanded by adding other objects to the program. For example, standard introductory physics objects like hollow and solid charged spheres, or perhaps even more complicated objects like charged cubes. Also, objects with current and magnetic field lines would make an important addition. Models that show change over time, like an RC circuit, would also be interesting.

Since the Oculus has become commercially available, the software could be updated to conform to the updates to the hardware. Optimizing the project for the more widely available version of the hardware would be ideal for a smoother user experience and facilitate more wide-spread use.

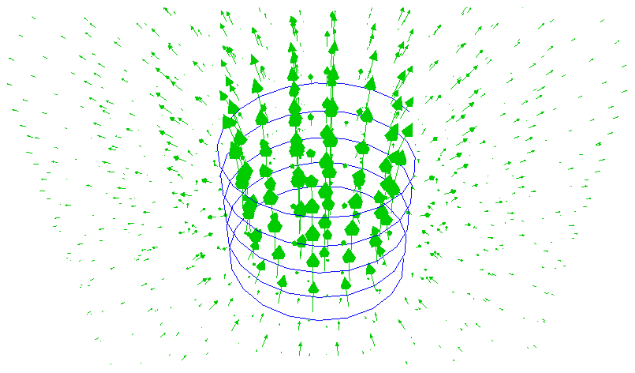


Figure 18. Solenoid with current (blue) generating magnetic field (green).

Advice

Since many of the problems we dealt with were due to incompatibility issues with the Oculus and the Leap, we think it would be advisable for future researchers to seriously consider all options. For alternatives to the Leap, we recommend looking at this Reddit page[1]. Many large companies are producing their own VR devices, so there are many choices available as alternatives to the Oculus[2]. However, code written to be compatible with the Oculus Rift will not necessarily be automatically compatible with such devices as the HTC Vive, especially now that Oculus is actively preventing compatibility.

We would also recommend writing a detailed environmental setup tutorial for posterity. Given that one of the biggest hurdles we overcame was setting up the environment used by Le, Ly, Tsung, and Vogel so that we could build upon their work, a detailed tutorial to help us recreate their environment would have saved significant time and energy.

In general, we would recommend checking dependency compatibility before beginning the installation process. We were unable to install on a Mac OS X Yosemite system because of architecture incompatibility between node.js, node-webkit, and the 64x operating system. We were unable to connect the Leap to desktop computers in the Syslab because the only Linux distribution the Leap supports is Ubuntu--not Gentoo, which is what the Syslab computers run. We had trouble installing the code written by Le et. al on a Windows 7 system because we failed to install Visual Studio and all of its components. Therefore, to ensure compatibility with the most components of this project, we would recommend developing on a modern Windows system (7+) and closely following the environment setup tutorial.

Older versions of the Oculus SDK support Windows, Linux, and Mac, but the software is buggy and has trouble displaying to the Oculus as a headset rather than as a second screen, so the display fails to split in half to better fit the two lenses. However, Oculus has gone Windows-only in newer versions of the SDK and runtime, including the commercial version, so we recommend future developers seriously consider developing on a Windows system. We also found that the Leap Motion device would have trouble connecting through any non-powered USB ports and virtual machines, so we would recommend using a dual-boot system over a virtual machine.

While we wrote our own physics engine, we used only Coulomb's Law and the superposition principle to model the electric fields. For more complicated behaviour (such as induced

1 https://www.reddit.com/r/oculus/comments/34mup0/the_most_comprehensive_list_of_vr_input_devices/
2 http://beebom.com/oculus-rift-alternatives/

magnetic fields), we would recommend implementing a more sophisticated physics engine. While we considered using an existing physics engine, we were not able to find a suitable one compatible with our system. However, given the compatibility issues we already faced with the hardware, we would recommend writing a custom physics engine to avoid further complications with translating between systems.

For those who would like to build off of our project, we would like to direct you to our GitHub repositories:

https://github.com/oboekevin/OcuPhysics contains the final version of our code as well as directions for setting up the environment.

https://github.com/oboekevin/Three/tree/gamepad contains a browser-based version of our program that uses an Xbox controller instead of the Leap. There is also a separate program that models the magnetic field generated by a solenoid with current.

References

Chafkin, M. (2015, October). Why Facebook’s $2 Billion Bet on Oculus Rift Might One Day Connect Everyone on Earth. *Vanity Fair*. Retrieved from http://www.vanityfair.com/news/2015/09/oculus-rift-mark-zuckerberg-cover-story-palmer-luckey

Chordia, P. (2015, June 17). NodeJS - How to resolve “Cannot find module” error [Online forum post]. Retrieved from stackoverflow website: http://stackoverflow.com/questions/9023672/nodejs-how-to-resolve-cannot-find-module-error

npm. (2015, April 27). Fixing npm permissions. Retrieved from npmjs.com website: https://docs.npmjs.com/getting-started/fixing-npm-permissions

Oculus VR. (2016, May 2). Oculus Rift Retail Experience Kicks Off at Best Buy Locations May 7 [Blog post]. Retrieved from Oculus Blog: https://www.oculus.com/en-us/blog/oculus-rift-retail-experience-kicks-off-at-best-buy-locations-may-7/

Rubin, P. (2014, May 20). The Inside Story of Oculus Rift and How Virtual Reality Became Reality. Retrieved from Wired website: http://www.wired.com/2014/05/oculus-rift-4/

sindresorhus. (2014, October 19). Install npm packages globally without sudo on OS X and Linux. Retrieved from github website: https://github.com/sindresorhus/guides/blob/master/npm-global-without-sudo.md

three.js / documentation. (n.d.). Retrieved from http://threejs.org/docs/