TJHSST Computer Systems Lab

Senior Research Project

3-D Modeling with Oculus Rift and Leap Motion Controller

2014-2015

Thai Le, Austin Ly, Ellis Tsung, Jonathon Vogel

15 June 2015

**Abstract**



Current industry standard computer-aided design (CAD) programs require users to interact with 3-D models on a 2-D plane through the use of a mouse and a keyboard. This design hinders user interaction with their models. Our project aims to remove this interaction barrier between a user and his or her model by creating a more “hands-on” and immersive modelling program. Through use of virtual reality via the Oculus Rift and hand sensing via the Leap Motion Controller, our project frees users of an uncanny learning curve created by traditional hardware constraining 3-D object interaction.

Man using the Oculus Rift and Leap Motion

**Introduction**

For decades, virtual reality (VR) remained an idea that existed only in concept. Contemporary technologies had proven too lacking to provide a user the convincing experience of viewing and interacting with a generated world, with impractically cumbersome CRT display technologies, immature motion tracking, and low fidelity graphics falling short of the immersion expected of VR (Drummond et al., 2014). As the requisite technologies advanced through the 1990s and 2000s, solutions began to emerge from cutting-edge research, but were predominantly expensive, specialized implementations and therefore not suitable for mainstream consumer use (Dye, 1995). As a result, VR existed as a niche interest until 2012, when company Oculus VR drew renewed interest back into the field by crowdfunding development of the Rift, a head-mounted VR solution consisting of a high resolution display and integrated head tracking with a consumer-friendly price tag of $350 (Redner & Schumacher, 2012). The origins of our research thus reflect the resurgence of attention the Oculus Rift has drawn.

While the Rift is a suitable response to the immersive VR problem, issues remain in how users may interact with these generated worlds. For video games, currently the most popular application of VR, the most common devices for control are the conventional keyboard and mouse or a dedicated gaming controller. Although these are sufficient for fine-grained control, with the former having the advantage of being ubiquitous in computing, they fall short of directly translating the user’s movements 1:1 in the virtual world. This is where the Leap Motion Controller excels as an input device.

Benefitting from a drop in price of 3-D sensors, the Leap Motion Controller is, fundamentally, a compact hand sensor with a built-in gesture and position tracking system. Placed on a flat surface, the Leap Motion Controller detects hand movements and shapes in a wide field of view along all three axes in three-dimensional space. Readily available APIs for several popular languages makes it an ideal choice for inexpensive, yet purportedly accurate and precise motion control. With a current MSRP of $80 we felt that together with the Oculus Rift, the Leap Motion Controller sits at an optimal balance of availability, accessibility, and functionality.

**Purpose**

Traditional modeling software has been used to create almost every 3-D object people interact with, both in virtual environments and when they are brought into the real word via CNC machines or 3-D printers. These traditional programs have been incredibly successful, and are used by artists all over the world. However, they share a fundamental flaw: their displays and input methods are both 2-D, leading to a cognitive mismatch between the users and their projects.

Many technologies have developed to address this situation in the form of new hardware, such as 3-D vision and motion tracking. Even holographic displays, with augmented reality capabilities, have been developed. Despite these advancements, there is still no software standard for interacting with 3-D environments, much less a software solution to true 3-D CAD.

The purpose of our project is to address these concerns by introducing immersive 3-D and standard, natural hand movements to the 3-D workflow.

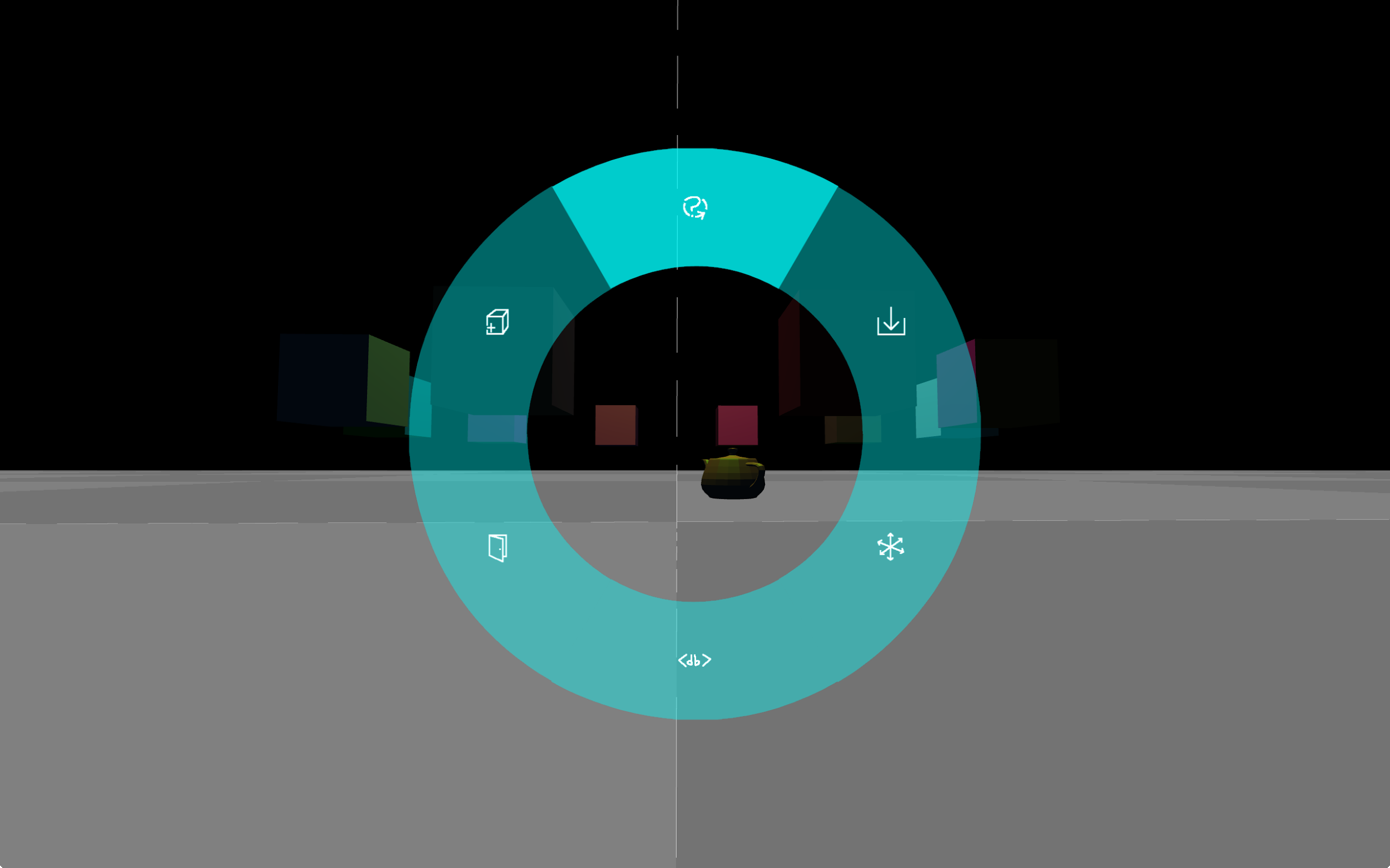
**Prior Work**

CAD-like applications have been rendered in a VR environment before. The popular voxel-construction game Minecraft has been modified to render on the Oculus Rift, allowing users to construct an immersive 3-D world with cubes (Lang, 2014). However, Minecraft does not utilize hand-tracking, instead relying on a mouse and keyboard. Placed objects are also snapped to a grid and limited to cubes of a fixed size. CAD applications themselves may be rendered in stereoscopic 3-D on a supported 3-D monitor, giving users a view of their models along the z-axis, but remain constrained to 2-D input devices, leaving users able to view but not interact along the z-axis. This setup is also impractical as it requires expensive displays, graphics cards, and 3-D vision kits, as the higher framerates required to run in 3-D necessitate specialized equipment.

Other projects include VRClay, which uses the Razer Hydra controller to shape virtual “clay” (Jamriska & Krs, 2014). The model is then displayed through the Rift. While this approach was effective for its niche use, the interface was unintuitive, hindered by the Hydra controller (Munier-Richard, 2011).. Furthermore, the controller was limited by wires, which required the user to stay facing the controller during use. Finally, Hydra controllers are no longer being manufactured, so a newer interface is necessary (Miller, 2013)..

At the onset of our project, there were already few Leap Motion Controller with Oculus Rift projects in development available as proof-of-concept on the Leap Motion development site. These projects focused on demonstrating the usefulness of the Leap Motion Controller’s hand tracking abilities in a virtual reality setting. As such, a typical example of a virtual reality project on the Leap Motion Development portal illustrates Leap Motion rigs within a virtual reality simulation. While these projects proved that integrating the Leap Motion into an Oculus Rift environment was possible, none attempted to tackle the same space as our project.

**Procedure**

Our project involves two main areas: virtual reality visualization and user interaction.

Menu options (clockwise from top):

* Exit / Restart Tutorial
* Import .obj file
* Toggle axis snapping
* Toggle debug menu
* Exit application
* Create object
  + Cube
  + Sphere
  + Torus
  + Cylinder

We approached user interaction development by breaking its development into atomic, modular gestures. A gesture is ‘activated’ by the user performing a signal movement to the Leap Motion controller. This signal movement, such as an outstretched index finger or a clenched fist, activates the mapped gesture, switching the user interactions to control that mode. For example, opening the menu is done by pointing forward with the index finger of the user’s left hand (configurable for left-handed people). After the menu is activated with this “signal” movement, the user can select different options in the menu by pointing that outstretched finger at the various options. To close the menu, the user simply retracts their outstretched finger.

These gestures can also be composed on top of each other for greater functionality: the select gesture, which is similar to the menu gesture but performed with the other hand, acts as a “primary” gesture. The user uses it to select objects, by simply pointing at them. Once an object is selected with this gesture, the user can, using their other hand, perform various “secondary” gestures—moving the object, resizing it, and so on. Our environment is set up so it is very easy for other programmers to add additional gestures to this system, hopefully in time creating a rich ecosystem of third-party extension gestures for all sorts of use-cases and workflows.

Internally, our gestures are centered around an event-driven API. LeapJS emits events into a self-contained event loop, separate from the event loop that drives the main rendering code. The gestures consume these events and transform them into higher-level ones, which are then emitted to other parts of the rendering subsystem. For example, the gesture subsystem would consume an event notifying that a hand entered the frame, and transform it into any number of other events—a menu toggle event, an object selection event, and so on.

The gesture subsystem is built to be generic and extensible. Gestures are implemented as self-contained modules, corresponding to a simple API. For example, here is an excerpt from our camera movement gesture:

var GestureMoveCamera = function() {

this.id = 'MOVE\_CAMERA';

this.type = gestures.gestureTypes.offHand;

this.MOVE\_THRESHOLD = 50;

this.initialPosition = undefined;

/// omitted helper code

this.canActivate = function(hand) {

return hand.sphereRadius < gestures.settings.triggerSphereRadius && !hand.indexFinger.extended;

};

this.activate = function(hand) {

this.initialPosition = hand.palmPosition;

};

this.update = function(hand) {

if (hand.sphereRadius > gestures.settings.triggerSphereRadius || hand.indexFinger.extended) {

return false;

}

return moveCamera(hand);

};

this.deactivate = function() {

this.initialPosition = undefined;

}

};

As this code demonstrates, gestures need to provide several properties and methods to be understood by our framework. The id property is used for event naming and debugging output. This allows, for example, the tutorial module to simply listen for an activate:MOVE\_CAMERA event instead of needing to directly couple with this gesture code. The type parameter is used to classify gestures into three types: primary, on-hand gestures such as select, which are performed with the dominant hand, primary, offhand gestures like move camera, which are performed on the non-dominant hand, and secondary gestures like scaling, which are typically performed on the non-dominant hand and require a primary gesture.

Each gesture also needs to provide the canActivate, activate, update, and deactivate methods. These are used to control the activation and lifecycle of the gesture. Generally, gestures state their activation criteria in canActivate. For the camera movement gesture, the trigger is making a fist with the left hand and extending the index finger. Because the gesture is tagged as offhand, canActivate will be called with a left hand, so it simply has to check the fist and index finger criteria. Once canActivate returns true, the gesture subsystem activates the gesture, which blocks the activation of any conflicting gestures and calls the activate method. This method simply saves the current position at activation so the gesture can use it to compute the offset afterwards. Then, on every data update event received from LeapJS, the update method is called. If the first check succeeds, the helper function moveCamera (not reproduced) takes care of offsetting the camera with the new movement data. If it fails, due to the user opening their hand or retracting their index finger, update will return false, and the gesture subsystem call deactivate. The deactivate method resets the gesture to its initial state, and afterwards the user will be able to perform another gesture with their left hand.

After the LeapJS data is sent through the gesture subsystem, it is processed into specific events denoting what gesture has been performed. The task then is to apply these events to the THREE.js 3-D models. This is done through the objects subsystem, which implements an event-driven wrapper over THREE.js meshes.

The objects subsystem build on the Node.js events module, and more specifically the EventEmitter class. This class is similar to what LeapJS uses to push events, but is implemented in native code. This high-performance event implementation allows us to rely on events for almost every part of the object system, from rendering to the aforementioned gesture events.

Here is another code excerpt from our project, demonstrating how a THREE.js mesh is wrapped into an object:

exports.makeObjectFromMesh = function (mesh, name, pos) {

var obj = new Entity('object.' + (name || 'unnamed'), ['renderable', 'selectable']);

obj.on('init', function () {

this.mesh = mesh;

if (pos !== undefined) {

this.mesh.position.copy(pos);

}

this.mesh.userData.parent = this;

});

obj.once('render', function (\_, scene) {

scene.add(this.mesh);

});

obj.on('select', function() {

this.mesh.material.emissive = new THREE.Color(this.mesh.material.color);

this.addTags('selected');

});

obj.on('deselect', function() {

this.mesh.material.emissive = new THREE.Color(0x000000);

this.removeTags('selected');

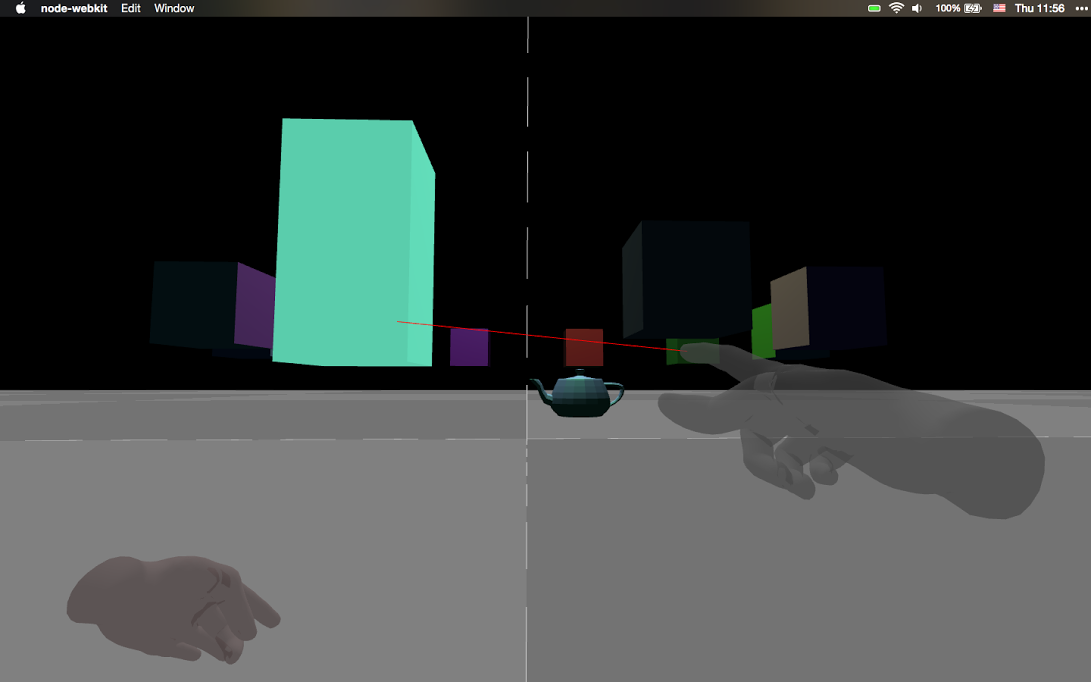
});

return obj;

}

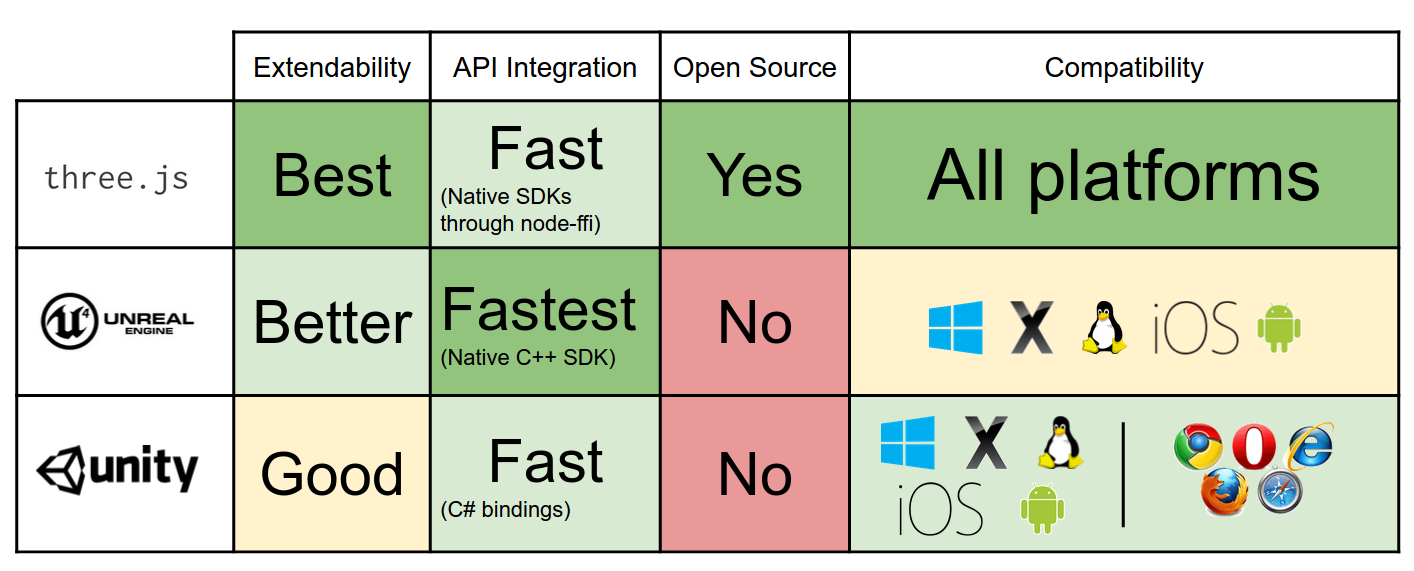
This shows how objects function as event sinks. After being readied for the scene, the object will receive the init event, allowing it to initialize its mesh with scene data. Because the object was tagged as renderable, it will also receive render events every frame. In this case, the render event listener is only triggered once, to add the object’s mesh to the scene. However, some objects listen to this every frame, for example to animate themselves or update textures.

Along with the rendering events, the object also listens for select and deselect events emitted by the gesture subsystem. Here, they’re used to manage tags and render the selection effects, but these events can also be rebound to trigger special object-specific behavior.



Rendering in our system is set up in 3 stages. First, our code sets up a single-eye camera and scene. The scene is populated with objects and handed off to a 3rd-party Oculus Rift THREE.js extension. This extension sets up two render contexts, each with the same scene but an offset camera, creating the stereoscopic effect needed for immersive virtual reality. Finally, these two offset cameras are rendered and distorted to conform to the Oculus’ lenses.

This rendering pipeline imposes some restrictions our project. The main problem is that, because the scene must be rendered twice, the performance requirements of a typical real-time 3-D application are more than doubled. Instead of the typical 16.6 millisecond time limit per frame, we have less than 6.7 milliseconds (Dunlop, 2003). On top of that, FPS drops inside the Oculus are much worse than in a typical 3-D application on a traditional screen. Oculus applications must maintain a consistent 75 FPS, equal to the display’s native refresh rate, and drops below 60 FPS can cause severe nausea and vomiting in users (Oculus VR, LLC., 2015).



The use of THREE.js allowed us to easily implement the rendering pipeline we needed for the Oculus and for integrating the Leap Motion controller. Early in the project, we explored several other options for a rendering framework, including natively using the OVRSDK through C++, using the Unity engine, or using the Unreal engine. Each of these platforms had advantages that we had to sacrifice by using THREE.js, but we also gained many benefits. By using a high-level language like Javascript instead of C++ like the OVRSDK or Unreal encourages, we gained clarity and conciseness in our code. Our application involved large amounts of data shuffling, and being able to move and transform data with the concise syntax Javascript provides proved invaluable to our project.

Additionally, the use of Javascript reaped more benefits: because our rendering code and HCI code shared a language, we didn’t have to deal with the complexity and performance hits of passing data between two different languages or frameworks. Javascript objects constructed on one side could be passed to the other side with no performance penalty, where trying to pass objects between LeapJS and Unity’s Boo scripting language would entail some sort of complex and slow serialization process.

Finally, and perhaps most importantly, THREE.js is open-source and permissively licensed. Both Unreal and Unity have some sort of licensing fee, and do not provide source code for their engines. Many times during development, digging into the THREE.js source helped us understand bugs and gave examples on how to implement certain features. Both Unity and Unreal charge massive fees and mandate NDAs for this sort of source access. THREE.js is publicly available on Github.

One interesting issue we had with THREE.js was how it wraps OpenGL. THREE.js separates color and ‘alpha’ (transparency, or a value from 0-255 where 0 represents clear and 255 represents opaque) into two separate ‘texture maps’. Each of these textures can be sourced from an image, video, drawing canvas, or anything else, but they’re separate. If the program wanted an object with a translucent black square, it would have to draw the square twice: once black, on the color texture, and once gray, on the alpha texture. This is how we first implemented the menu, but as the drawing code became more complex, so did this odd separation of rendering. Finally, we implemented a fix in the rendering2d module: the alpha canvas helper. This method created a proxy canvas context that would externally mimic the CanvasRenderingContext2D API, but internally split each call into two hidden contexts, one color and one alpha. Barring a few hiccups, this worked very well, and proved invaluable in implementing the various 2-dimensionally textured objects in our VR world.

**Results and Conclusion**

Our project is successful as a proof-of-concept for virtual reality and hand-tracking based CAD. With a variety of user interaction functions, adequately manipulatable on-screen models, and the ability to import outside CAD-program generated models, our project accomplishes its goal of better bridging the gap between a user and his on-screen models.

Overall, we succeeded in building an application with a relatively low learning curve. Compared to traditional CAD applications, a user can quickly our program and within minutes be familiarized with gestures and movements they need to make to in order to interact with their on-screen models in a 3-D environment. The immersive head-tracking offered by the Oculus Rift provides a simple way for the user to look around their environment without using complex camera controls present in traditional CAD applications. Additionally, the camera movement gesture allows simple 3-D motion in the a similarly intuitive style. Instead of using a 2-D mouse,our users can simply move their hands in the direction they want to move the camera.

On top of this intentionally simple control scheme, we tried to provide a professional, easy-to-follow tutorial that would walk users through their encounter with our application. Our tutorial coaches our users through our gestures, explaining how to activate them and provides visual confirmation when a user has correctly performed a gesture. The tutorial walks users through from their first step of aligning their hands with the Leap Motion to the movement and scaling gestures, and finally, operating the in-world menu.

However, our goals for an intuitive interface had several shortcomings. The Leap Motion gestures are sometimes difficult to activate, due to hardware limitations. The Leap Motion has a small detection range and it is easy for users to unintentionally move their hands out of that range, deactivating active gestures. Furthermore, hand detection is often jittery, due to the nature of IR sensing.

**Further Research**

Virtual reality is a rapidly expanding field with incredible amounts of promise. A few years ago, none of this technology existed; there is no telling where new developments may take the industry.

Oculus, along with other startups, have worked on positional tracking in 3-D space using a combination of accelerometers, gyroscopes, and magnetometers. In Oculus’ case, the headset was tracked around wirelessly. This provides an opportunity to control our program by walking around, which is more intuitive than learning hand gestures. Other products such as the Perception Neuron provide a more comprehensive detection of the user’s position in space, while also monitoring limb movement. Although it loses accuracy due to integrating acceleration down to position, it makes up for it by providing much finer grain control and adjustable resolution—while Perception Neuron can capture the user’s arms and legs, users can move sensors to capture individual finger joints or just the user’s legs.

Microsoft’s upcoming HoloLens provides an augmented reality experience similar to the Oculus Rift’s virtual reality, allowing the user to view an immersive 3-D environment overlaid on top of their existing surroundings. Just like how our project empowers artists to more easily create immersive virtual realities, a HoloLens-enabled version of our project would empower artists to create more immersive augmented reality applications.

Similarly, more advanced hand tracking devices are beginning to arrive on the market that surpass the capabilities of the Leap Motion. The Kinect 2, released with Microsoft’s Xbox One, can track fingers like the Leap Motion while being far enough away to also track the rest of the user’s body. This solves many of the bounds problems with the Leap Motion, while also enabling gestures involving the entire user’s body. Other devices like the Myo wristband do not rely on optics at all, instead detecting the flexing of the user’s muscles to detect their finger positions. This removes the line-of-sight problem from the equation entirely, trading off positional accuracy with flexibility to perform actions out of sight.

**Advice**

The largest obstacle to progress in our project was by far working with the Leap Motion Controller. It is a device that has been getting extensive attention recently, not undeservedly so, for its exciting applications in human-computer interactions (HCI). However, we would like to temper this excitement with real facts.

First and foremost, the Leap Motion Controller is unreliable. Due to its use of poor resolution cameras, with a very small interaction box due to a very short viewing depth, it it is easy to lose track of a user’s hands if they move too far or too fast. On top of that, even if it can detect the correct hands, there is no guarantee that it can detect all the parts of the hands—we faced constant problems where the controller could not see a finger because it was occluded by the rest of the hand, and simply tried to guess a correct position. Because of the issues with detecting occluded limbs, the sensor does not work well with closed fists viewed from the back of the hand, which kept us from using the Leap VR mount.

On top of this, the most common implementation of hand rendering for the Leap, the leap-rigged-hand extension, has a further problem. The hand model used in that extension is only one size. Humans have a surprising amount of variation in their hand sizes. When the leap-rigged-hand plugin internally scales position coordinates to account for this, it leads to a significant difference between the point the gestures believe the hand is and where the hand is rendered on-screen to the user. We worked around this, at a cost of accuracy, by pulling coordinate data from the leap-rigged-hand plugin instead of the controller itself, but for a production-ready system this would need to be fixed in a more elegant way.

The leap-rigged-hand extension was also lacking in that it could not interpolate positions between frames. As a result, the hand model frequently lost track of the user’s hands, which caused random jittering and disappearing hand models. We created a robust system for detecting the hands when they reenter the frame and for rejecting bad frames, but the movement of the hand models is still not fluid. On the other hand, there are examples online that track and display hands more reliably and smoothly, so there must be better way.

Our gesture detection system was also suboptimal. In order to trigger different gestures, we hard-coded them in. This meant that we had to hard-code collisions between gestures. For example, a closed fist needs to be superseded by a fist with an extended pointer finger, and thus the fist gesture had to detect if the pointer finger was extended, and then cancel itself for the other gesture. Given the few number of gestures used by our project, collisions were not very common. However, our current solution, given more collisions, would result in hard-coding the exact configuration of extended fingers, hand orientation, and more, which would quickly become unfeasible. Thus, the detection system must be reworked.

There are two ways of accomplishing this; the first is to use a hierarchy of gestures, where a higher priority one displaces a lower priority. We have a structure similar to a hierarchy via the list of enabled gestures. However, the program must then consult that list every frame to select the first gesture that can activate, and if it is not the current gesture, the program must then swap active gestures. This would result in a lot of recomputation of hand metrics, and this method does not address hard-coded gestures.

Conversely, one could employ a framework similar to “LeapTrainer.js,” which can learn different poses and detect the best match. By checking every frame, similar to the previous method, LeapTrainer could detect the best gesture to activate through more efficient methods, and gestures would not have to be hard-coded. Although the development of the framework appears to have stopped, the detection methods used (geometric detection, neural networks) could serve as an effective example.

**References**

Drummond, K., Hamburger, E., Houston, T., Irvine, T., Tieu, U., Lai, R., Lathrop, D., … Mazza, C. (2014). The Rise and Fall and Rise of Virtual Reality. The Verge. Retrieved from http://www.theverge.com/a/virtual-reality

Dunlop, R. (2003). FPS Versus Frame Time. *MVPS*. Retrieved from https://www.mvps.org/directx/articles/fps\_versus\_frame\_time.htm

Dye, L. (1995, February 22). The Cutting Edge: COMPUTING / TECHNOLOGY / INNOVATION : Virtual Reality Applications Expand : Imaging: Technology is finding important places in medicine, engineering and many other realms. *Los Angeles Times.* Retrieved from http://articles.latimes.com/1995-02-22/business/fi-34851\_1\_virtual- reality

Jamriska, O., Krs, V. (2014) About. *VRCLAY*. Retrieved from http://vrclay.com/ about/

Lang, B. (2014). Guide: Play Minecraft on Oculus Rift DK2 Using the Minecrift Mod. *Road to VR.* Retrieved from http://www.roadtovr.com/guide-how-to-install-minecraft- oculus-rift-dk2-minecrift/

Miller, E. (2013). Razer Hydra successor the STEM System launches on Kickstarter, hits funding goal in mere hours. *VG247.* Retrieved from http://www.vg247.com/2013/09/ 13/razer-hydra-successor-the-stem-system-launches-on-kickstarter-hits-funding-goal-in-mere-hours/

Munier-Richard, T. (2011). Razer Hydra Review. *DigitalVersus*. Retrieved from http://www.digitalversus.com/gamepad-controller/razer-hydra-p11655/test.html

Oculus VR, LLC. (2015). Oculus Rift Best Practices. *Oculus Documentation.* Retrieved from http://static.oculus.com/sdk-downloads/documents/0.6.0/Oculus\_Best\_Practices\_ Guide.pdf

Redner, J., Schumacher, E. (2012) New Virtual Reality Gaming Headset From Oculus VR™ Gets Kickstarted. *Oculus VR.* Retrieved from https://www.oculus.com/press/new- virtual-reality-gaming-headset-from-oculus-gets-kickstarted/

[Untitled photograph of the Leap Motion Controller]. Retrieved from http:// www.food4rhino.com/ sites/default/files/leap\_3\_quarter.png

[Untitled photograph of the Oculus Rift]. Retrieved from http://towleroad.typepad. com/.a/6a00d8341c730253ef01a73dc8893c970d-pi