Psychomotor Interaction for Enhanced VR User Experience

Stavros Kateros

Thesis submitted in partial fulfilment of the requirements for the
Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Associate Prof. *George Papagiannakis*

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

This work has been supported by the Foundation for Research and Technology – Hellas (FORTH), Institute of Computer Science (ICS)

UNIVERSITY OF CRETE
COMPUTER SCIENCE DEPARTMENT

Psychomotor Interaction for Enhanced VR User Experience

Thesis submitted by
Stavros Kateros
In partial fulfilment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Stavros Kateros

Committee approvals: _____
George Papagiannakis
Associate Professor, Thesis Supervisor

_____
Antonios Argyros
Professor, Committee Member

_____
Dimitris Grammenos
Principal Researcher, Committee Member

Departmental approval: _____
Antonios Argyros
Professor, Director of Graduate Studies

Heraklion, October 2019

Psychomotor Interaction for Enhanced VR User Experience

Abstract

Virtual reality (VR) as a medium is starting nowadays to be widely embraced from consumers. Despite being considered as a new field, it is currently going through its fourth attempt in market adoption. VR was first commercially introduced four decades ago but failed due to hardware technology being too young, and the concept itself not being truly understood. Supported applications were not created with a dedicated VR design in mind to take advantage of this new medium's potential. Thus, software (and design) were both, not able to back up this new way of experience. Nowadays hardware is mature enough to comfortably support system requirements while being inexpensive enough to be consumer accessible. Although the software and design part of VR is following along hardware's progress, this field is still in its first steps, more specifically in design and implementation of the VR user experience (UX). UX is extensively researched and understood in other computing areas. But virtual reality is quite different because each individual does not just use an application with its interface being the bridge between human and machine but is "part" of the application, i.e. immersed in the three-dimensional world where traditional 2D interfaces do not necessarily always apply. Interacting in VR is difficult to be standardised due to its highly personal and subjective matter.

In this thesis we research the foundations of what can be standardised for an optimal VR user experience. The most basic concept that required re-thinking was embodied user interaction with the virtual world. To enhance the experience, we introduced implementations that simulate physical, psychomotor interactions, imitating dexterous use of hands (and body) in real world. These are accompanied by specific user-assisting procedures devised to bridge the gap of the medium limitations (e.g. field of view, user focus etc.). Furthermore, general VR design principles are introduced, to complement this endeavour in taking advantage of VR's true potential and staying true to term *reality*. These VR design principles bear on user's consciousness and feeling of "being and doing there". From understanding human's senses and using them to boosting immersion, and from acknowledging the stark differences between virtual and real interactions, to respecting user's comfort and ease of use in VR.

Ψυχοκινητική Αλληλεπίδραση για Βελτιωμένη Εμπειρία Χρήστη σε Περιβάλλοντα Εικονικής Πραγματικότητας

Περίληψη

Η εικονική πραγματικότητα ως μέσον έχει αρχίσει τη σημερινή εποχή να είναι μαζικά αποδεκτή από τους καταναλωτές. Παρόλο που σαν κλάδος θεωρείται σχετικά νέος, στην πραγματικότητα βρισκόμαστε στην τέταρτη απόπειρα αποδοχής του από την αγορά. Τα πρώτα βήματα της εικονικής πραγματικότητας έγιναν τέσσερεις δεκαετίες πίσω αλλά χωρίς επιτυχία επειδή η τεχνολογία στα μηχανήματα υπολογιστών βρισκόταν σε πολύ αρχικό στάδιο, και λόγο ότι, η ιδέα αυτή δεν ήταν ακόμη πλήρως κατανοητή. Υποστηριζόμενες εφαρμογές δεν είχαν σχεδιαστεί έχοντας υπόψιν την εικονική πραγματικότητα για να εκμεταλλευτούν πλήρως της δυνατότητες του νέου αυτού κλάδου. Οπότε ούτε το λογισμικό ούτε τα μηχανήματα μπορούσαν να υποστηρίξουν τον νέο τρόπο εμπειρίας. Πλέον, η τεχνολογία στο υλικό των υπολογιστικών συστημάτων έχει ωριμάσει αρκετά για να μπορεί να υποστηρίξει τις απαιτήσεις του συστήματος και ταυτοχρόνως να είναι αρκετά φθηνό για να μπορεί να είναι προσβάσιμο από καταναλωτές. Παρόλο που το λογισμικό και οι σχεδιαστικές τεχνικές στην εικονική πραγματικότητα ακολουθούν την πρόοδο του υλικού, και τα δύο βρίσκονται ακόμα στα πρώτα τους βήματα, πιο συγκεκριμένα στο σχεδιασμό και υλοποίηση της εμπειρίας χρήστη σε περιβάλλοντα εικονικής πραγματικότητας. Ο τομέας της εμπειρίας χρήστη έχει εκτενώς ερευνηθεί και κατανοηθεί από άλλες περιοχές έρευνας στον κλάδο των υπολογιστών. Όμως η εικονική πραγματικότητα διαφέρει αρκετά, λόγο ότι ο κάθε χρήστης δε χρησιμοποιεί την εφαρμογή με τη διεπαφή της να είναι η γέφυρα μεταξύ ανθρώπου και μηχανής, αλλά ο ίδιος είναι ΄μέρος΄ της εφαρμογής, δηλ. είναι βυθισμένος σε ένα τρισδιάστατό κόσμο όπου παραδοσιακές δυσδιάστατες διεπαφές δεν ταιριάζουν. Η κανονικοποίηση σχεδιασμού αλληλεπίδρασης στην εικονική πραγματικότητα είναι δύσκολη επειδή είναι ένα πολύ προσωπικό και υποκειμενικό θέμα.

Στην εργασία αυτή ερευνούμε τα θεμέλια των βασικών κανόνων για τη βέλτιστη εμπειρία ενός ατόμου μέσα στο κόσμο της εικονικής πραγματικότητας. Η πιο βασική ιδέα που χρειάστηκε επανεξέταση ήταν η ενσωματωμένη αλληλεπίδραση χρηστών στον εικονικό κόσμο. Για να εμπλουτίσουμε την εμπειρία αυτή, εισάγουμε υλοποιήσεις όπου προσομοιώνουν φυσικές, ψυχοκινητικές αλληλεπιδράσεις, αντιγράφοντας τις επιδέξιες κινήσεις των χεριών (και σώματος) στο πραγματικό κόσμο. Αυτές συνοδεύονται από συγκεκριμένες διαδικασίες για τη διευκόλυνση χρηστών, σχεδιασμένες να γεφυρώσουν το κενό που δημιουργούν οι περιορισμοί του μέσου αυτού (π.χ. γωνία θέασης, εστίαση χρήστη, κτλ.). Επιπροσθέτως, γενικευμένες σχεδιαστικές αρχές εισάγονται για να συμπληρώσουν στην προσπάθεια αυτή της πλήρης εκμετάλλευσης των δυνατοτήτων της εικονικής πραγματικότητας ώστε να είμαστε σωστοί απέναντι στον όρο *πραγματικότητα*. Οι σχεδιαστικές αρχές συσχετίζονται με τη συνείδηση του χρήστη και την αίσθηση ‘βρίσκομαι εκεί και πράττω’. Από τη κατανόηση των ανθρώπινων αισθήσεων και τη χρήση τους για να ενισχύσουμε την εμβύθιση, και από την αναγνώριση των κύριων διαφορών μεταξύ εικονικών και πραγματικών αλληλεπιδράσεων, στο σεβασμό της άνεσης του χρήστη και την ευκολία χρήσης μέσα στην εικονική πραγματικότητα.

Attestation

I understand the nature of plagiarism, and I am aware of the University's policy on this. I certify that this dissertation reports original work by me during my University project except for the following:

- Unity 3D, game engine this project is based on
- Newton VR, Unity plugin from Tomorrow Today Labs
- Steam VR, Unity plugin from Valve Corporation
- Volumetric Light Beam, Unity asset from Tech Salad
- Turbo Squid, website for in scene 3D objects
- Freesound, website for in scene sounds
- Final IK, Unity plugin from RootMotion

Acknowledgements

There was great help from my supervisor Professor George Papagiannakis. His investment in time and effort from the days when I was undergraduate came forward into completing this chapter in my life. In this long and exciting journey there where a lot of obstacles, achievements, ideas and vision that were shared and I couldn't be more grateful. During this journey, my path was crossed by many important people that contributed to its completion. People that we shared space inside our lab and worked together in various projects. A lot of things that I accomplished for my thesis came from their inspiration and ideas. Thank you, Nikos Lydatakis, Stelios Georgiou and Pavlos Zikas. My final gratitude goes of course to my family for believing in me all these year during my studies and never left my side.

Table of Contents

List of Figures

List of Tables

1 Introduction

There are multiple interpretations about what defines a well, thought-out, design regarding the user's experience inside a VR application. Firstly, we have to ask what our VR application's purpose is. Is it just a regular app that is ported into a new media or is it a completely new, designed and built, from ground-up project to represent what this concept is all about, virtual *reality*?

The answer should be obvious despite not being an easy task to explain what lies ahead. It is a relatively new technology platform and regarding UX we are stepping into relatively unexplored territory. In this thesis we attempt to puzzle out parts of this unexplored field (psychomotor UX in VR) and to suggest implementations that can be standardized by concentrating on two simple principles, the preservation of the user's immersion and natural interactions.

1.1 First Principle: Immersion

While using the application, the enjoyer is absorbed in another world as he/she is completely cut-out from the real one. It is not self-evident but even the smallest, most insignificant -unexpected- incident inside the app is going to break that beloved presence. Once this happens it's all over. It might be not possible for the user to "re-connect" with the virtual world. These incidents can vary from frame drops, to the application crashing or -in general- to experience sudden unnatural behaviours during the experience.

The constant preservation of immersion is the leading necessity (1 p. 47). All elements that compile the application should be tested exhaustively with the key-objective being that nothing can be tolerated. A smooth flow of hardware and software that will not cause any issues, and, application design to keep the user's interest alive. The first part of the end-goal is, and should, be the same for any type of application, a polished product for user satisfaction. If not done correctly, it can break user's engagement, nonetheless, it does not mean that it assists to this engagement, if done right. Immersion falls into the design.

Immersion by itself is an abstract concept. The elements that keep the user invested inside the virtual world can be considered as controversial as we cannot strongly define what is to "blame". Staying on the subject of mimicking the real world, the initial focus should depend on real world stimuli that do have feasible implementations. Apart from the obvious, vision, a lot can be achieved with the proper use of spatial sound and illusion of touch with the controller's use of vibration. The in this thesis focus lies mostly on these, because as fields they are well known, easy to experiment on and the most importantly, hardware available.

1.2 Second Principle: Interaction

The second principle invests on the implementation of all interactions inside VR to be achieved in a natural, instantly understandable / familiar way. Since reality is the main ideology, all physical (object) interactions should mimic the physical world to further assist on the user's immersion as he/she can interact virtually as intuitively as physically. If a user must be taught in the virtual environment how to perform a simple, everyday action that he/she would easily do in the real world, then the, already difficult to implement, immersion, starts to become impossible.

First obstacle to this idea is the hardware limitations. Even though we do get 6 DoF controllers there is a limited amount of space the user can perform on. Also, since mimicking real world interactions is the goal, how do these types of actions get implemented for objects being smaller from the user's controllers in hand? This is a good question to comprehend the upcoming problems. For cases like this one, different algorithms that have nothing to do with the ways of interaction in the physical world must be created. There is a "thin line" the developer must walk

on, between code (and idea) to function unrealistically yet perfectly, and the user feeling the performing action to be realistic.

2 State-of-The-Art

User Experience (UX) in research terms, is very abstract to be categorized. In this section a brief history of VR will be presented, from the idea being slowly born to its early development that laid the foundations on where it stands today. This will be followed with previously and current ideas being applied, in both hardware and software, to enrich one's experience with this new medium.

2.1 VR Reborn as new Consumer Medium

As an idea, virtual reality, is very old dating back to mid-18th century, but this idea took form in the 60s from a combination of pilling ideas that took form as the first VR headset in 1968 (2). Despite several commercial releases of VR HMDs in previous decades, true success came four decades later were the technology was matured enough to be widely spread to markets while being supported with a lot of third-party content.

Figure 2.1 Left: first VR headset 'Sword of Damocles' from Ivan Sutherland (1968), centre: Virtual Boy video game console from Nintendo (1995), right: Oculus Rift S (2nd gen) from Oculus VR (2019)

During this journey every iterations and design focus was on hardware itself. This counts from the early research days till its first steps as product. Considering also limitations at the time, slow processing hardware, low quality displays and lack of crucial sensors (e.g. gyroscope) that could be all housed under a small package, and it is easy to see that it was an idea ahead of its time. Neither researches at that period were focused on UX, but mostly on proving why and how this new way of interaction with the digital world could be useful for the public (3) (4). During this current decade is where a lot of effort from researchers and companies goes directly to user experience contribution.

2.2 Hardware Contribution

The common modern solution provided to the user for interaction is the controller. These controllers apart from button input play the role of the virtual hand. They, usually, have 6DoF achieved with build in sensors such as accelerometer and gyroscope (rotation) and from external tracking sensors (position). Most common template of the controller is a two-button layout. One for index finger to simulate pitching and another for the rest of fingers to simulate grabbing. Inside VR what the user sees is the virtual hands only where their default hand pose is similar to the way their provided controllers are held. Otherwise a simpler solution is to display only the controller itself virtually. Controllers are well adopted from consumers and as time goes by more iterations are coming forward. Touch sensitive buttons on controllers, analogue buttons (or whole controller input with strap) for visual representation of grabbing strength are some deviations from this concept.

One more design that all controllers are sharing, is haptics (5). This idea is starting to leap out of controllers and land on wearables in order to take advantage of vibration for more parts of the body, beyond hands. One example can be wearable vests with multiple vibrators throughout the body. Despite its simplicity, haptics can be a powerful ally. A controlled vibration (duration, strength) can assist with the touch sense and overall experience (explained in this thesis). The more human senses are manipulated, the better it is for immersion. Currently full advantage is taken of two out of five human senses, sight and hearing. With only controller vibration, there is not effective stimulation regarding touch. With full body wearables this sense comes significantly into play to support this race for full immersion.

Alternatively, user input can be applied to virtual hands without controllers but with hand tracking technology. In front of headsets there are sensors (cameras, infrared, etc.) that see user's hands and simulate their movements to the virtual hands. Movement can be the entire hand in world space and also the hand itself in its own space (separate finger movement). This adds to the immersion as the user can see 1-to-1 hands movement into the virtual word, without being restricted from controllers and their physical capabilities. But lack of vibration takes away one of the three senses. Additionally, this solution is preferable when one is not holding anything and just "plays" with the hands. When any type of interaction occurs, holding nothing at hand is also another setback that must be considered. But the biggest issue, with current state of technology at least, is the end result not being perfect. Cameras losing hands view with certain poses of fast hand movements are not translated well into VR (broken unnatural movements – more realism is pursued the easier it becomes to spot foibles). To achieve a better visual result a lot has to be, currently, sacrificed thus favouring the controllers for user adoption (6).

Figure 2.2 Left: Hand Controllers, centre: hand tracking with cameras and infrared sensors, right: gloves with pressure sensors

Sticking to hand tracking, the same result can be achieved with gloves that have pressure sensors inside them. It is extra gear for the consumer to wear but it produces the same or better results from sensor-based hand tracking, as the posture of the hand will not affect the quality of the animation inside VR (not to mention lower computational costs).

There are a lot more ideas being tested for new hardware to assist the user experience, but a great help can come from far simpler solutions, such as small iterations of existing headset technology. Iterations that do not take a step closer to immersion but ones that are able to eliminate factors that can break user involvement. Untethered headsets (wireless screen-casting or all computational force housed inside headset) and inside out tracking sensors. Currently most commercially available headsets have external sensors tracking the headset and the controllers and they are physically connected to a pc via wire. External sensors have excellent tracking accuracy but are limited to their placed positions. And as far as limitations go, one more candidate is wire restrictions. Both at some time will be enough to limit the user in the real world shattering the bridge between this this and the real one. Inside out sensors are responsible for the same work but are placed inside the headset. These two have their area limitations too (area recognition) but are always following the user decreasing the changes of tracking loss. Lastly, an untethered device makes the benefits over a tethered one quite obvious (one could also add user satisfaction

derived from simplicity; no external sensors to set up on wires to connect and to constantly have in mind not to step on – just one move, wear the headset and enjoy).

2.3 Software Contribution

As mentioned before, first real steps towards user experience tailored to this medium started from the second commercial breakout of VR headsets, this very decade. One step came from the Oculus VR Company with their first headset (commercial gen 1 with controllers). At that time most interaction where without controllers. It involved user gaze and timers. One would see the element he/she wanted to interact with, a ray would hit that element and after a small time had passed (to avoid constant accidental triggers) it would be triggered. When controllers came, they were not taken advantage of their full potential. Users saw inside VR the controllers themselves, and they were used for interaction in a similar way before their introduction, with ray-casting. A ray was extended from each controller and when the ray fall on an interactable element, instead of waiting, the user had to press the controller's button to trigger the element. What oculus did, was to render inside VR a virtual hand instead of the controller, with a hand pose that matched the real hand's pose while holding the controller. By the time a user pressed any of the buttons different animations would play to simulate the real user's movement. Interaction was also changed (ray-casting still sort of existed) from rays and triggers to "real interactions", with the hand itself (this thesis hand behaviour was inspired from this idea).

Figure 2.3 Left: Oculus rift virtual hand mimicking real hand (touch sensitive buttons), right: Steam VR home with controllers and ray-casting

This type of hand animation had a great impact and started to be used from many applications since it is a great step to improve experience and relatively simple to create. But this has its limitations too. A reminder should always be that as the interactions are approaching realism, the clearer it is to spot imperfections and "damage" the experience. These types of virtual hands have specific animation that only match what the controller can do. Thus, when grabbing a virtual item, the pose of the hand does not match the object's geometry. Adding to this issue, is the ghosting effect where hands go inside objects behaving differently from the rest of virtual world. A new solution came from another Company, Ready at Dawn (7) where they programmed the virtual hand in a way to adjust itself on any object's surface. The hand has physical properties and no matter what the item or surface (large predictable), the hand's posture (palm rotation and fingers) will match that surface. An excellent step a developer can make, and something that is not affected from the application's design.

Figure 2.4 Left: Lone Echo's hand adjusting to surface, right: default hand-grab animation

Realistic hand animation that retains its physical properties is a step to the correct direction. The second one is applying these animations to the whole hand, or ideally, the whole body, because of course presence is not complete without full body tracking. A person's body is constantly in view in real life, not having one inside virtual reality makes it a lot effortless to realise that everything is not real. Body tracking can be achieved in two different directions. Hardware based with many sensors attached to a person's body, and software based with prediction from headset and controllers. The focus on this thesis is on user experience. This counts for both inside and outside of the virtual world. Wearing sensors each time will always be more troublesome for one to experience than just wearing a headset. On software side all body movement is achieved with mechanics that are known in the graphics world, inverse kinematics (8). Many attempts have been made with full body tracking, mostly this is used with the aforementioned IK and prediction algorithms. Still none of the algorithm based implementations are perfect, but it is getting better, mostly on upper body (9) (where most movements happen and being the most difficult to predict).

Lastly, regarding user interface there are two directions that are common in VR world. First one is traditional 2D interfaces that mimic mobile interfaces infused with design rules specifically for VR (10). Two dimensions may seem at first the wrong step for a VR application, but if done correctly they can be a viable solution (for end-users and, for separate reasons, for developers as discussed later in this thesis). The other, more modern, solution is to blend the interface with the natural interactions of the virtual world's elements, where some of the interactive objects will also serve other roles (as for example menus).

Figure 2.5 Difference between 2D and 3D UI menus. Left: Steam VR Home with 2D panel, right: Batman Arkham VR bat-cave with a role of "in-game menu"

No matter how well designed a 2D interface is, it can still fall short to the highest expectations that one can have for this field because it still fails to complete the term reality. No person has encountered in his/her life a floating, bright-coloured, two-dimensional canvas of options; it remains unnatural. Options, menu, notifications, all can be designed and implemented by using the environment's objects (11). There is little to no reason estimating the complexity of this design as everything relies upon the availability of the objects around and the design of the virtual world. Only thing that can be for sure is the lowest complexity cost of this implementation, still exceeds (or be on the same level) the traditional 2D implementations. A modern example of 3D object-based UI is used in Valve's game, The Lab. The enjoyer can enter different world while in the application. Instead of different menus, each world is an orb where inside it shows a miniature of the world one is about to enter. To do so, one must grab the orb and place it on the head as if entering this orb. Real life item that can be familiarized used elegantly as this UI.

Figure 2.6 Valve's "The Lab" world selection VS miniature world orb

The dynamism behind the search for understanding what truly the term *virtual reality* stands for, is incredible. There are many books (one of which is a big inspiration for this thesis (1)), researches, papers and so on, expressing ideas in many different topics for enhancing this experience that is difficult to pin-point them. What is not difficult is to find the one thing all have in common, that is a user-centric design.

3 Psychomotor Interaction for Enhanced VR User Experience

3.1 User Interaction

In-game design's driving force is the user experience (12 p. 41).The most fitting synonym to this term, is user interaction. Interaction encloses all user's movements and handling with the environment he/she is presented with (by environment we refer to a scene containing static and dynamic objects). Static objects and the way the user can connect with them is something straightforward. Static interaction can be flipping a switch on a wall to turn on/off the lights, how many different implementations can this have? The spotlight of new ideas and development must lay upon the dynamic objects and how the user can manipulate them.

Interaction is a field that should revamped for the type of medium in both design and development. In design terms, one basic rule is to implement interactive behaviours to all objects that anyone can interact with. In interactive applications (e.g. video games) this is not a design rule, but in VR it is very important (12 p. 79). In development terms, the rule is to pursuit intuitiveness (1 p. 277) in those behaviours. In the end, the behaviours are not just mechanics to connect human and machine (although it is), but it is a replication of human actions.

3.1.1 Object Interaction

Such type of interaction is very well known (dynamic objects - especially in the gaming world) and a lot of polished implementations exist. The catch is that these existing solutions work perfectly alongside a technology that has remained untouched since the beginning of the computing world (variations do exist but in core they are all the same). For our case we focus on the user input. A game controller or a keyboard and mouse are the dominant controls for the user to interact with objects. Using these tools, the user's interactivity with the objects is indirect, a button press, a small movement (2D) from the mouse restrained from its small surface area, are not 1-to-1 actions the user would normally perform, so the algorithms transform them into different actions inside the 3D virtual world. Despite the important role of designing a well thought out and easy to pick up and play user controls, all this time, this remained the only variable into the developer's user input design equation. This simplistic equation allowed the developers to take a lot of shortcuts (mostly in an effective and cost cutting way) in design.

Taking advantage of this -not so apparent to the user- disconnection, the developers can implement a lot of obstacles to stop the user from misbehaving inside the virtual world (e.g. confine player's movement to limit the amount of programming and design). Furthermore, they can "cheat" by applying the same type of interaction to all types of objects even if they do not actually apply, realistically thinking. As long as the enjoyer does not realise any of these shortcuts or his/her constant flow of positive experience does not come to a halt, it proves to be a win-win situation.

The biggest portion of this ideology will not work in VR! Still controllers are used for the interaction, but in an entirely different way. They have 6 DOF (position tracked) and they take the role of the enjoyer's hands. This immediate realization is that user's actions are (or seem to be in our eyes) 1-to-1 with the actions performed inside the virtual world. Shortcuts are now gone, at least the known ones, and implementations cannot remain the same as there is a new goal to chase, presence. The equation has now more than one parameter. Below the most common way that objects interact with the user are explained, why they will not work in VR, and the suggested-replacing method is unfolded.

3.1.1.1 "Parenting"

As described previously the attention will be on dynamic objects only. Interacting with static objects most of times will not change their state. It will trigger an animation or change the state

of another object but not itself. As in dynamic objects, their state is directly manipulated from the user. Now a deeper examination is required to detail the "state" part of the object and what other behaviours must be watched out. Every object has its transform values relative to the origin of the world, or scene, that it is in. This consists of the position, rotation and scale. Small note, scaling is not common for a real-world object, as most are rigid bodies, hence only positional changes can apply to them (position and rotation). These values can be handled from the developers via scripting behaviours. Furthermore, considering realism as the goal, it can be clear that these objects will obey to the laws of physics. To achieve so, the object must be observed from the physics engine (usually a separate process running the whole scene through physics simulation). Its transform values are now manipulated from two different mechanisms, the developers scripted behaviours and the physics engine. This is where the problem hides (it will be examined in the next part).

When the user grabs an object, it should follow the hand movements in the world. Since the hand is an object by itself, it is easy to know its transformational changes. Imagine the user is holding a brick and then raises the hand above him. The hand is translated on Y axis by T. To make the brick seem that it is held by hand, the same T transformation is applied to it, simple. But the when the hand rotates in some degrees R, the same transform cannot be applied directly to the brick as it will rotate around itself and not around the hand. These two objects have an offset.

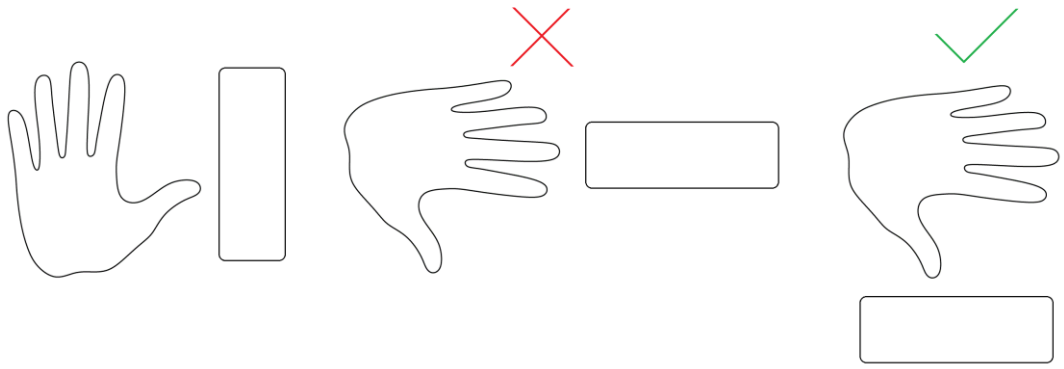


Figure 3.1 Hand-object correct parent transformation

The solution is very simple as when an update occurs to the hand's transformational values, the changes are applied based not to the hand's origin (Object Space) but to the world's origin (World Space). In the same principle the brick must obey to the hand, thus its changes are based on the hand's origin. These are basic affine transformations, well known into the world of graphics and are implemented inside every game engine. It is called parenting (13). The basics work in the same way in terms of transform, only more conveniences are added for the developer. This solution supports many objects at the same time, infinite depth of grandchildren, ease of access and usually behavioural changes in physics simulation (simulating from the parent's origin and not the child-object). Before parenting starts, the starting positions and rotations of the parent and child objects are held in order to always keep that offset during transformational changes. Then on every internal update the child-object position is updated similarly to the parents and the rotation is calculated based on parent's rotation with the additional starting offset.

```
Function: Update Object Position Depending on Parent Transform

Preconditions: This Objects Transform (this_Object_Transform)
               This Objects Starting Position (this_Object_Start_Pos)
               This Objects Starting Rotation (this_Object_Start_Rot)
               Access to parent Object Transform (parent_Transform)
               Access to parent Starting Rotation (parent_Start_Rot)


parent_Position_Vector <- Get Position From Transform(parent_Transform)
parent_Rotation_Vector <- Get Rotation From Transform(parent_Transform)
parent_Scale_Vector <- Get Scale From Transform(parent_Transform)

// TRS - Translate Rotate Scale
parentTRS_Matrix <- Convert Vectors to Matrix (parent_Position_Vector ,
                                               parent_Rotation_Vector , parent_Scale_Vector)

Set Position (this_Object_Transform) <- Find Position in Relation To(parentTRS_Matrix)

Set Rotation (this_Object_Transform) <- (Get Rotation(parent_Transform) * Inverse_Quaternion(parent_Start_Rot))
                                          * this_Object_Start_Rot
```

Figure 3.2 Obeying to transformation changes of parent transform

This is an excellent solution, as the results for the user are spot on and for the developer an easy and certainly fast solution. Not for VR. A good example can be a car racing in a game. The wheels are the car's children. Any changes to the car's transform are updated to the wheels and also the car is regarded as one rigid body from the physics engine thus the wheels do not need a separate treatment, apart for a behavioural script for wheel rotation when the car is moving (this can be too implemented into the physics engine). The user's hand inside the VR is different. It receives continually updates from the sensors for its position (controller in real world – to hand in virtual world) to such degree that it cannot be observed from the physics engine. The immediate aftermath is that the item the hand is holding will lose its physical properties and will become a "ghost" penetrating any solid (as the user sees) item inside the scene. The connection between real and virtual world inside the brain is severed.

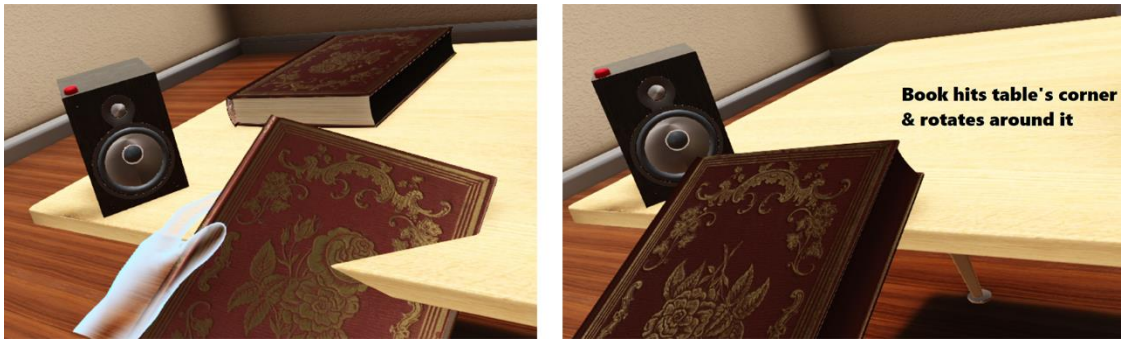


Figure 3.3 Physical behaviour in parenting vs velocity-based

The same principle will not work even in "manual" mode where only transform values are updated. This will keep the object's physical properties when held by the user, but it creates more unsolvable problems. The reason lays at every frame where there is a constant battle to update the item's transformational values. The physics engine on one hand and the behavioural script, running on game logic update, on the other (they are not synchronised, Figure 3.4 below). If the book being held (from the example above), has approached the table at some given point here is what it's going to happen in a frame. The physics engine will try to stop the book from translating any further (not updating its transform). Then the behavioural script starts and observes the book's position is not updated in relation to hand's new position for this frame and it updates the transform. The next frame the physics engine sees that the item has trespassed another rigid item and updates the it's transform opposite to where it's headed. Then it's the script's turn and the whole process repeats itself. Note that physics sim runs separately, so it might be updated after more than one frame where the object by then will be translated even further, becoming more noticeable from the user.

This problem is visualized below (Figure 3.4), displaying Unity's (game engine) internal execution order. The first part is the physics where on each internal update the developer's physics code is first called (Fixed Update) and then the internal physics update is called for all objects

in scene (Internal Physics Update). This is where the velocity changes are updated to all physical objects. After physics are done the system updates are called (Update). In this section all developers code-updates are called, that includes all transformational object updates that are not physical. As shown, these two different system updates are called separately and are also asynchronous. This is not just Unity's structure but generally this is how internal updates are handled in these types of engines.

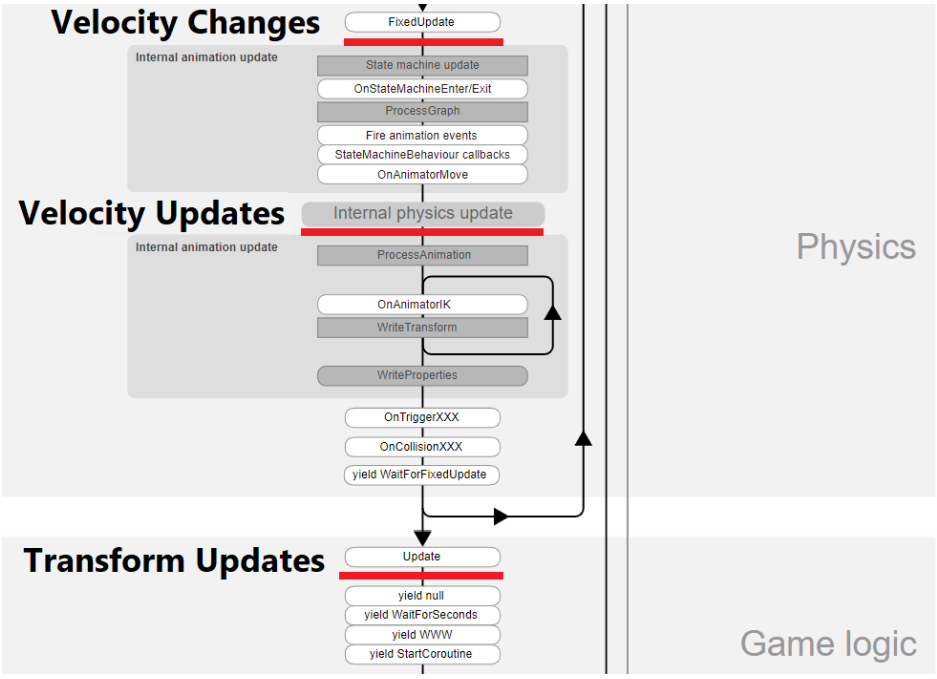


Figure 3.4 Unity's execution order

To continue with the list of problems, a method like this does not offer a solution to the behaviour of the object after the interaction with the user is over. In real life when an object is thrown away, it has a momentum built from the hand till the time it's released. Then the object keeps that momentum and it's gradually losing it from forces such as gravity and friction from the air. Since its movement with parenting methods is updated by directly changing its transformational values in the separate process, this means that inside physics update no changes are observed. Thus, the item has constantly zero velocity. Trying to throw something away will result with a sudden halt, followed by a perfectly vertical drop after the physics engine will get the hold of the released item. All the build up from the user's hand is gone and that makes a world of difference. Especially for real world conducts that occur on day to day basis. The solution must solve these issues, be as robust as the current known methods – to have a reason to be adopted and be easy to work on for developers.

3.1.1.2 Velocity Based

The reasonable path towards the object manipulation is the physics engine. Setting up the correct parameters for the object's interaction behaviours with the user and then feed the parameters in order to be calculated alongside the physical properties of the item from the same process. In short, the new method will observe changes in the parent transform, in our case the hand, and then apply to the child item in question.

When the object is held by the user's hand, as already mention, it does not "obey" to the hand, meaning its transform is free (otherwise the parent transform would be the hand). Instead what it does is to change its values via the Rigidbody (Unity's physical instance to an object) on each physics internal update. The object's position and rotation are moved towards the hand's position (and rotation). This is what parenting does as well, but instead of manually accessing the transform values, in this case the new desired position is taken (hand's updated position),

then the new position between the object and the hand is calculated and then applied directly to the Rigidbody velocity. This is the same velocity that is manipulated from the physics engine, in other words, this implementation works similarly (and alongside) to the physics simulation.

The benefit is that other physical objects around our scene can interact with the object at hand and have more natural behaviours since this kind of behaviour works the same way as the interaction. But there is also an indirect benefit. A logical behaviour for the user is to interact with an object with both hands (if size allows it). This is not possible with the parenting method (dialling out quick-dirty fixes) due to the way it works; an object can only have one origin, hence, it can be interacted with one hand at a time. With the velocity-based method, the object can be manipulated with both hands as it is just a matter of average between the two forces (ok it has some more math in it to work properly). This is very important for the overall experience, but it is also something that needs extra care from the developers, as if the implementation is not as seamless as with one hand interaction, it can contribute to a worse experience (14).

```
1  Function: Update Object Velocity
2
3  Preconditions: Set maximum allowed Velocity Update (max_Velocity_Update)
4                 Access to target object (target_Object)
5
6  this_Object <- Get Object Physical Instance()
7
8  target_Velocity <- Get Object Velocity(target_Object)
9
10 // Lerp translates object from one target to the other
11 Lerp(Set Object Velocity(this_Object) , target_Velocity, max_Velocity_Update)
12
```

Figure 3.5 Using velocity for positional changes (part of algorithm)

Down to the core of this algorithm the transformational changes are really that simple. Small exception goes to the rotational part as it needs some more calculations. Rotations are calculated with quaternions and Unity does not provide operators to work and furthermore the angular velocity is a vector. The final calculation requires to be a vector (angle axis). First the rotation quaternion must be calculated from the rotations of hand and object (hand rotation * Inverse (object rotation)). Afterwards, Unity, for the quaternion, provides a method to convert quaternions to angle axis. The important thing to remember is to correct the sign of angular velocity due to the right-hand rule (if it exceeds 180 degrees).

So, the algorithm is established and has convincing results, but still it is not perfect. Yes, it does follow the user's hand precisely, although it has a small delay – it is barely noticeable, it does not have a sense of "weight". Holding a one-kilogram speaker and holding a thirty-kilograms hammer will "feel" the same for the user. A different solution is to translate the object is by adding forces instead of direct manipulation of velocity. This provides a more realistic effect (delays in movement) but it creates a bigger problem. When the object is just grabbed and not moved around the added forces are zero. Thus, it will not be held at hand as gravity takes over.

At first glance, both need a fix to be one step closer to perfection. The force based will need "hack" workarounds as the problem lies in the idea itself to move the object with added forces. These types of solutions can never be stable, thus being completely reasonable to be avoided. With the velocity based, its problem can be solved purely with design and not by further software development. If the user can pick and manipulate an object it makes sense to be easily moved around. A careful design of the environment, for example placing within user's reach objects that are, or seem, light enough is a viable solution. For heavier object just like the aforementioned hammer, the interaction can be two handed. Still the light effect will be present but holding with both hands has a potential to take users mind off this unwanted effect. And lastly the physics engine itself lays a helping hand in this scenario. The objects might feel the same when controlled by the user, but when object interact with each other, their mass difference will be present! Hitting the heavy hammer with the light speaker will result in a miniscule translation of the hammer, but on the other way around if the user is going to grab the hammer to hit the

speaker the result will be the latter flying away into the distance. The developer's effort to create this effect is just to fiddle with the Rigidbody parameters for each different object.

3.1.1.3 Assist Object Interaction: Posture

It may not be immediately understood but trying to grab and hold large items (like a book in the example above) it is mostly an effortless procedure. An object with a large area, easy to be spotted and have relevant understanding where it is in the environment and how to reach it. An idea like this will not stay on its feet when taking into consideration interacting with small objects. In real life a human being does not have to think how to hold a marker – for example; when at hand, it is likely to be held directly into the preferable position. If not, adjusting the hand or assisting with the other one will put the marker into place instantly. In VR, where the controller has the role of the hand it is impossible (considering current technology and controller size). One or two digital/analogue buttons to simulate hand's open/close posture and their physical size are the obstacles there is a high change for a small item to not be placed correctly at hand. If the virtual item is going to be smaller in size from the controller, when the user is going to attempt re-adjusting it with the assist of the other hand, this will result to the controllers colliding (in real life). It is by design (physical and virtual) a troublesome and complex situation for something this simple.

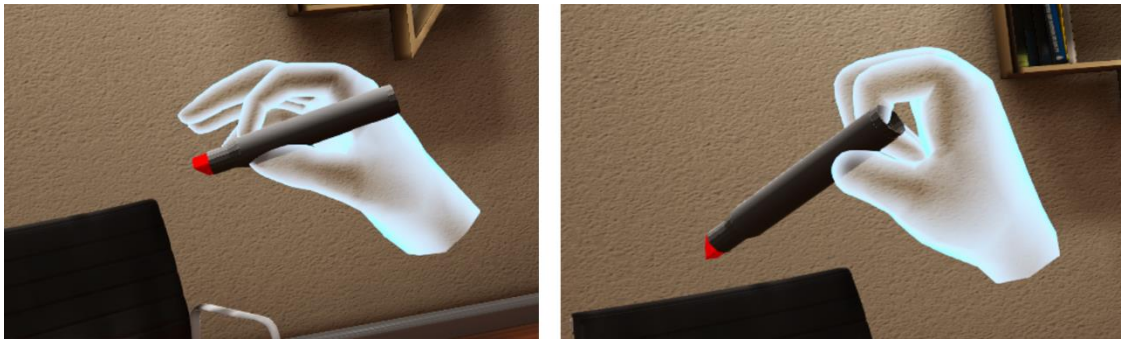


Figure 3.6 Fixed object position (marker is correctly self-adjusted between fingers) VS free (marker retains posture when grabbed from user)

A proposed solution is to introduce a self-adjusting mechanic for these small items. They will be objects that can be grabbed with only one specific posture. When the individual has started the interaction, no matter the way it's being held, the item will re-adjust itself into the correct position inside the hand (starting from the same frame the user just pressed the controller button to interact with the item). The developers will set a new origin for both left and right hand, so that the item can immediately adjust into those two origins depending on what hand is holding it. The freedom of interaction is restrained, but this is a consequence that a developer should not fear, as it is hidden from physical limitations (with current technology in mind where most advanced VR HMDs use controllers).

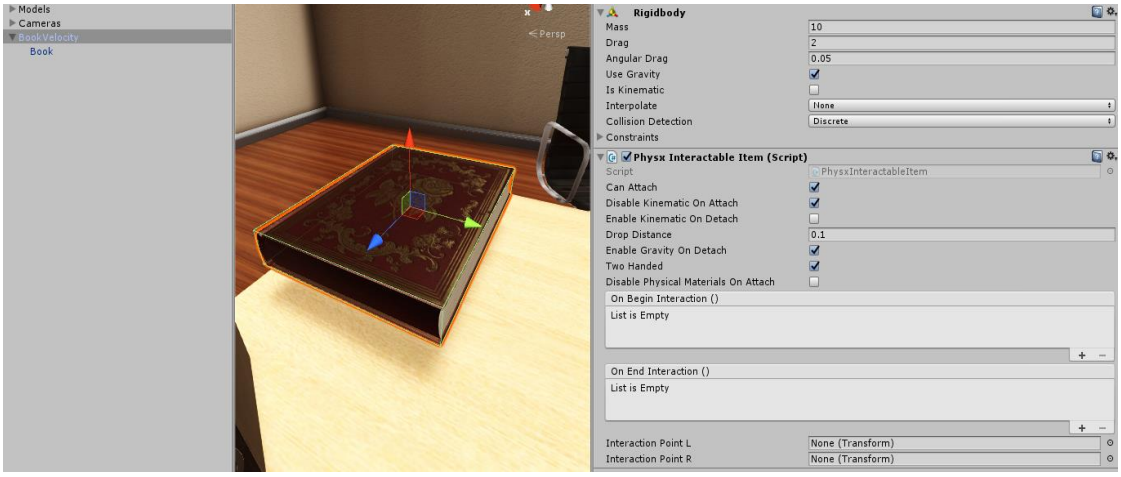


Figure 3.7 Physx Interaction Script: containing the interaction points for fixed object postures for both left and right hand (bottom right)

Using the interactable method inside Unity, it is going to be simple work for the developer. The script responsible has two Unity Action types, On Begin & End Interaction (Figure 3.7). These are trigger functions called once when the item has just started or ended respectively its interaction with the user. What is required next is to play the animation, a common way to implement it is with the use of Unity's coroutines (15). Also from the provided Device Controller Class both right or left hand are available, so their respective transform (to take the position for the item to be placed to), from the Physx Interactable Item it is possible to get what hand has interacted with the item and if this item is listed as a small object (with a specific posture), the dictionary is available to get its target rotation at hand. The interpolation is ready to commence.

```
Function: Initialize Object Interaction

Precondition: Force-based Interaction Script

// Add function call when object starts interacting with user hands
On Begin Interaction() <- delegate Function: Correct Position On Attach()

-------------

Function: Correct Position On Attach

Precondition: Access to dictionary with stored object posture for both hands (object_Posture_Dict)

current_hand <- Get Hand Holding This Object()

this_Object_Name <- Get Object Name(this_Object)

if(current_hand is NOT empty)
    target_Position <- Get Hand Position(current_hand)

    if(object_Posture_Dict Contains this_Object_Name)
        target_Rotation <- object_Posture_Dict(this_Object_Name , current_hand)
    else
        target_Rotation <- Get Object Target Rotation(this_Object)

    Update Object Transform(target_Position , target_Rotation)
```

Figure 3.8 Part of implementation for object fixed interaction

3.1.1.4 Assist Object Interaction: Distance

There cannot only be one design to work for all interactions (in terms of providing a natural feeling). As explained in the section above it is essential to have a list of smaller mechanisms that work "under the radar" to make all interactions feel natural. In this area falls the distance issue. It is observed that users underestimate distances in VR (depending on how the environment is laid down) (16). The newly introduced mechanic can be a separate collision area attached to the item at times when it is not interacting with the user. That area's role is to only await hand interactions. The size will depend on the item's shape, spawn distance from user,

importance and so on (in Unity's terms, this is a triggered collider with a layer that can be observed only from user's hands). Furthermore, this interacting area should be dynamic. When the user's hand is observed to be relatively close to the object itself, if several attempts to grab it are registered, the area then will auto-expand until the item is successfully grabbed from the user (Figure 3.9). Its assistance is crucial as it is difficult to understand what is going on at the given moment. Not being able to interact with an object where an interaction is supposed to occur, only adds to frustration.

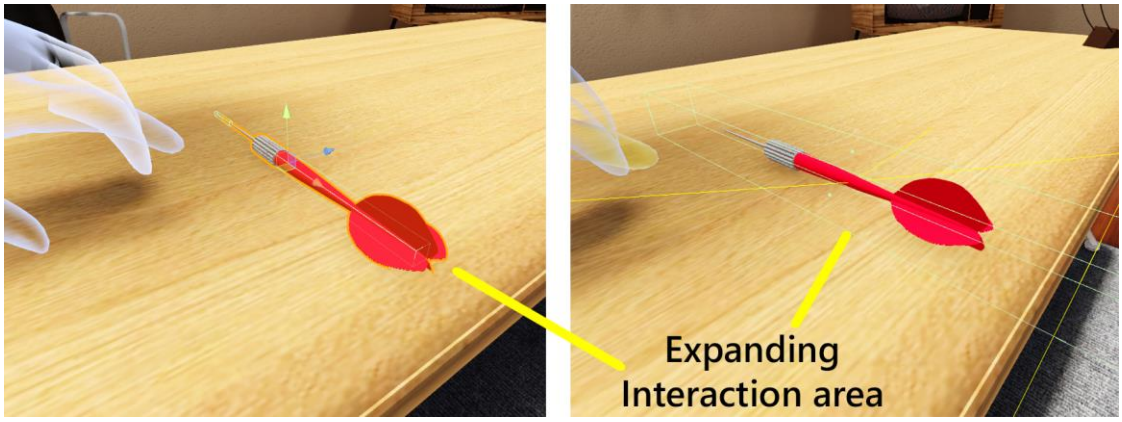


Figure 3.9 Interaction area expands (green grid) as user attempts to grab the object from a greater distance

3.1.2 Scene Interaction

Handling objects is not the only way of interaction. From the time someone wears the headset until he/she enters the virtual world and starts experiencing new things, there is a bridge between these two. This can take the form of a menu or any sort of options available to the user (and so much more, though the spotlight will shine upon the usual suspects). This is going to be referred as the scene interaction.

When a design is well thought-out and implemented there might not be a reason to redesign it when porting a program to another hardware. However, this ideology applies to similarly functioning hardware. For example, there is no need to change the menus when porting a PC game into consoles. Their input is quite similar. That said, when porting it into a mobile device, this type of interaction cannot remain the same. In this case there is not a medium between the user and the application (e.g. a controller) but the user can interact "directly" with the application with the use of touchscreens. The bright side to all this is that no separate implementations (usually) are required.

3.1.2.1 Ray-casting, Explained

Redesigning and implementing something proven, is costly and time consuming though, especially in a field where there is no proven guidelines and principles to follow in the new design. This is the trap where many applications fall when ported into VR. The same type of menu is presented, a 2D panel, in some distance away for everything to be visible. More often than not, this 2D panel will be curved in order to fall inside of the user's "sphere" of influence. Still every option will not be within reach. The solution is to use ray-casting (17). A straight ray comes out of the hand/controller, just like holding a small laser and trying to point at things at a distance. Wherever this ray falls, a button from the menu for example, it gets highlighted and after a visual timer has passed the button is simulated as "pressed". This solution is inexpensive yet effective and something that a person can be acquainted with after some time of use. It is only natural to be selected for the VR ports. The main budget in a VR application (game or not) falls, or should, into the reality aspect. The less interaction with a menu, the better. The logic behind this step becomes even clearer when counting the mobile VR headsets that do not have controllers. Ray casting from the centre of user's view is the one-way route (VR experience without at least one

acceleration-only based controller seems to come to an end as the limit of the experience is great).

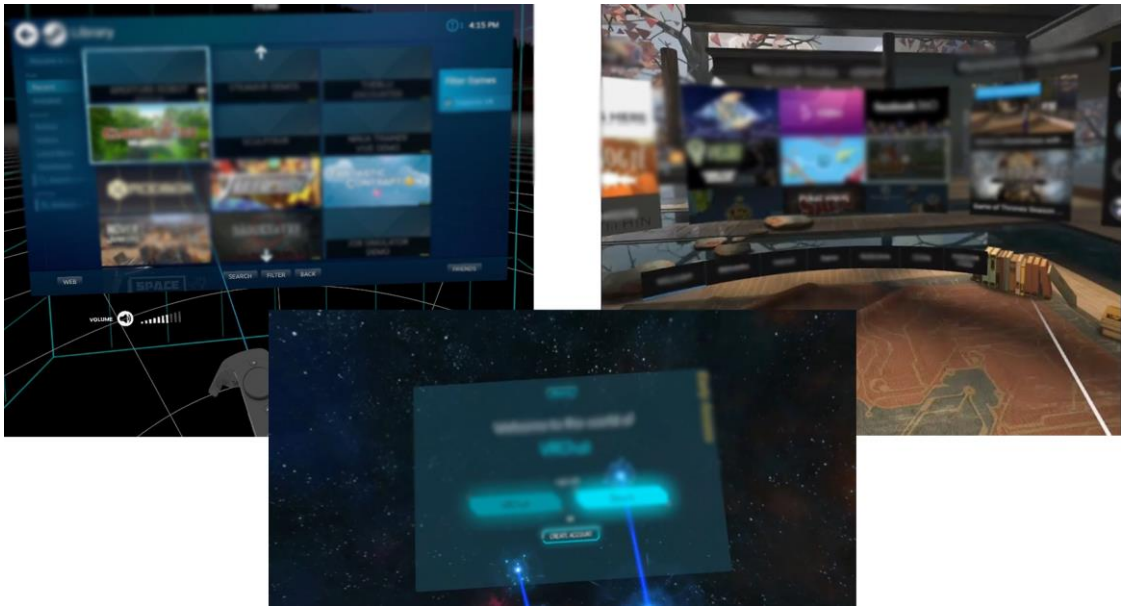


Figure 3.10 VR examples with ray-casting menus

3.1.2.2 Keep Your Interactions Close, Reasons Why

A new way of interacting with the scene inside the virtual environment should be designed. One that can work with desktop and mobile headsets and that compliments the novelty behind this field. A good way to start is by keeping these types of interactions close to the user (and to a bare minimum). Since all the positives of ray-casting are established above why should anyone care enough for a redesign? Virtual / Augmented / Mixed reality is not just a trend. Neither is it an improved nor refined existing technology. It is something new that requires this hard work. One good example is around the 2010s the trend for 3D screens that immerged. This was an old technology (first television introduced in 1935!) way ahead of it's time that was brought back for commercial use. It felt rushed from the companies and it was something that never caught on from the public as it was too expensive, a lot of trouble for not that much of a difference and not enough material taking advantage of this technology (for example different camera angles to enhance the depth effect). The result is that it came and went (18). One last good example is the touchscreen smartphone. Touchscreens are around for many years and there were marvellous touchscreen phones before the first iPhone was introduced. The problem was in their design and the public was not fond of them. It was a computer-oriented design that was implemented straight on a small screen phone with completely different type of input controls. It was not simple to use (not ideal for an everyday device) and the user had to adapt to the way the phone wanted to be handled. When the iPhone came, it didn't change anything to the technology (ok apart from that brilliant multi-touch screen) but it changed completely its design. User friendly, easy to use and most importantly, natural interactions with the device. It felt like there was no bridge between the user and the program the device was running (bridge of course being the UI). And the rest is history.

Figure 3.11 iPhone's friendly & intuitive design VS P800 design

When someone enter these new worlds the very first thing that creates the most important - first- impression is the world itself, the audio-visual stimuli. The second tier is the scene interaction. Imagine having an enjoyer being blown away with the immersive graphics the beautiful acoustics and immediately after having a "TV remote controller" trying to point on a button in the distance. A good design is one that falls into every category as the iPhone's design did. There is not a straightforward solution to this. What anyone should keep from this section is what the title suggests, keep everything close to the user. VR is a new paradigm and it should be treated differently. This ideology is where the focus of this thesis interface lies on.

3.2 User Interface

A VR application needs too its own user interface. It can be out of the VR experience, requiring user to remove the headset (as some applications have), include the interface inside VR or both. There can be a single design but the means of interaction with the interface are not universal. Numerous headsets do not have 6 DoF (Degrees of Freedom) controllers for convenient interactions (tracking rotation and position), but only 3 DoF, as their input is only an accelerometer, and others do not have any. The implementation for the interaction should be input independent (same design and same ease-of-use at least). The design can follow two routes, the traditional route with 2D implementations and the more modern, VR route, where the UI is integrated to the virtual world in form of 3D objects (as explained in State of the Art). This thesis UI is focused on the traditional way with updates to the design to fit this new world of media.

3.2.1 The Traditional Way

The user interface is important to conventional applications as it can heavily affect the experience itself. It is the bridge between the user and the system. Responsible of user's involvement extension. It is a field that has a wide range of applications and it is researched and implemented for many years. The traditional 2D UI can be used by VR applications but with an explicit reminder. This type of user interface does not belong inside the virtual world, especially when the desired results are presence and immersion (when was the last time one walked out to see floating panels) and it should be kept to a minimum.

3.2.1.1 Design Philosophy

A commendably designed user interface can offer a lot to the overall experience inside the VR application. The difference with the more conventional ones is that UI's role loses its place under the spotlight and fades to the background. It has a supplementary purpose (for example a

menu) and despite the importance of its design to fit the rest of the world, there is still a limit in resources and amount of effort required to develop a proper UI. If indeed the interface has a supplementary role, then its use from the enjoyer will be from a conscious decision that all by itself will be a halt from the undergoing immersion (if someone is immersed to a VR stage and wants to exit to go to another one or possibly lower the volume, and so on, it is by then a conscious decision to stop the game and open a menu). The traditional 2D user interface, that is well known, is still an excellent design choice, if done right.

The interface does not have a fitting place inside the virtual world. It should not be present to the user and its lifetime should be as short as possible. The reason for these elements to be used should only be because of user external needs (e.g. pause the game), and not related to the game's experience.

This is the philosophy all UI elements follow in this project. Limited lifetime and most importantly, limited number of elements presented at the user at the same time. Most common elements inside the application usually are the interactive types (buttons in an option panel) and the notification types (system interruptions, warnings, user assistance). The interactive elements can be triggered from the system but mostly from the user whereas the notification ones are system triggered. Thus, the latter has higher hierarchical order. A user should have only one element at a time presented to keep the information to a minimum (and a cluster of different visuals at the same time). A hierarchy for each new element is required.

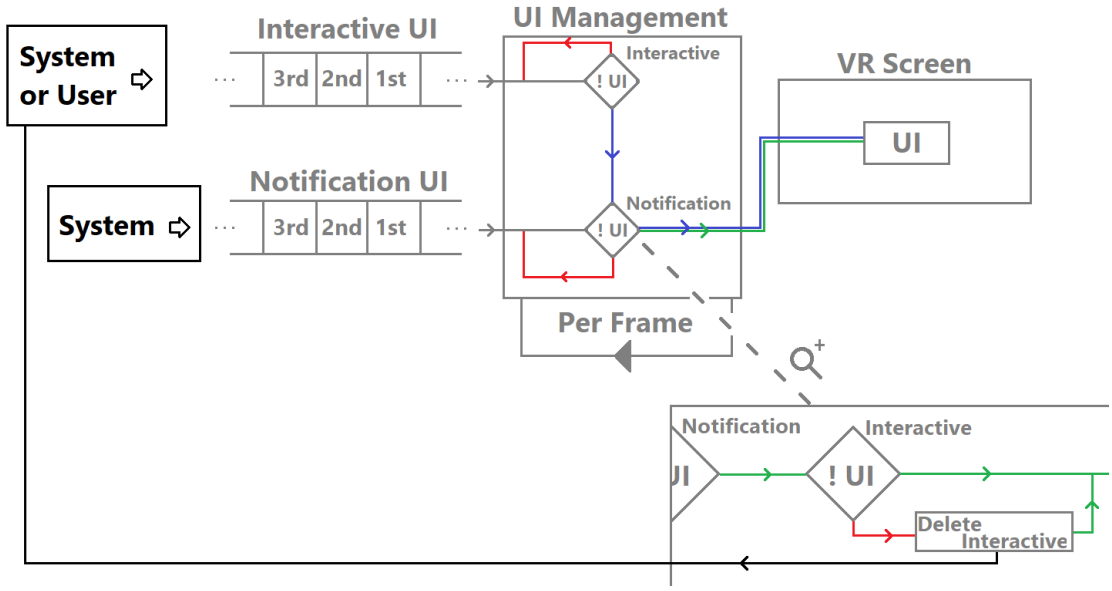


Figure 3.12 UI Management's workflow

Each element when called to be spawned, gets stored in a waiting queue (Figure 3.12, Interactive UI & Notification UI) where each one is later being spawned with the same order when entered the queue (a.k.a. FIFO). There is a separate queue for the interactive and the notification types. Top priority (as it is only system interrupted) have the notifications and then the interactive types. This means that when an interactive element is first on the list, it waits to be spawned until no other interactive or notification element is currently displayed (Figure 3.12, UI Management). On the other hand, the notification types, before instantiating, only wait till there is not another notification element on the screen. When it is time to be displayed if another interactive element is on screen, it gets deleted (Figure 3.12, zoomed-in Notification workflow) and re-inserted to the interactive waiting queue (Figure 3.12, Interactive UI). At the same time at the bottom of the notification, the name of the deleted interactive UI is displayed with the message "will appear shortly" for the user to understand what is going on at the moment. It is a simple method that keeps all elements ordered, holding these visual "interruptions" to a minimum.

3.2.1.2 Detailed Principles

User's interface design philosophy for virtual reality can follow, as told, the "traditional" implementation but with new details that when implemented, they take the role of the design's foundations.

Smoothness is key (19).All element movements should be smooth, predictable and have a purpose. If a new element is about to appear, a smooth fade in and a fade out is much preferred from a sudden pop in. Apart from leaving the impression of a polished result (20), it feels more natural as for example behaviours can be more predictable (1 p. 129). Furthermore, one must keep in mind that with a good designed animation a lot more information can be achieved (21), without the need for further visual elements to explain them (e.g. opening a panel from an options bar; the panel, when opening, flies up from the bar itself. With this animation it is easy for the user to understand that the panel came from the options bar without additional information).

Interface simplicity. The goal is always to make the interface as easy to understand as possible without the need for users to think what could possibly mean in what they are seeing. Same goes for VR. When an element is presented it should be clear, and if possible, with little to no text as possible (all VR forums have at least a section of text readability being arduous). Text is difficult to be done right, mostly due to resolution problems (even this is starting to be fixed with very high res displays in headsets) and should be used only in cases that there is no alternative. Designing a visual element with a small range of colours and a clear sketch can be enough to be almost instantly recognisable. To this, a lot of help comes from the digital age humanity is going through right now where all people are in constant interaction with technology, thus, many designs are becoming natural to all humans. To anyone that has used a smartphone, a gear icon is associated with application's options menu (it is used from all phones, and not only. The repetition of information is associated with learning (22)).

Similar interactions. Despite the elements being two-dimensional, their way of interaction should not differ from the three-dimensional objects inside the scene. The user should reach for the button and try to grab it as in all other objects in scene, as in real life. Since the third dimension is missing, it can be effortful for the user to land his/her hand on the element itself mostly because of miscalculated depth. One solution can be the 2.5D implementation. 2D objects that can move about in all three dimensions and appear that have depth. For the user to know that the hand is hovering on the element, a button for example, that element should move backwards as it would have done if it was a real-life button pushed by someone's hand. Upon pressing the button on the controller, the hand closes and the element is interacted. Same type of interaction for everything facilitates to the application's apparent unification.

Understand spacing. Developers should have in mind when designing any UI elements, the space between the element's sub parts and the space between the element and user. Starting with the first one, the element's borders should be clear to the user and distant enough to avoid accidental triggers. This of course is a rule that applies too outside of VR (mobile phone design). The rule that applies only to this special case is the latter, spacing between user and UI. Since the interaction is similar for the 3D objects, the UI should be in close proximity to user. Distant enough to be easily observed (issues with close elements to the eyes – especially text), but close enough to be within reach. Important role has also the 2,5D behaviour of the element's interactive parts (pushed back when hovered).

3.3 Notifications

Every application or game has its own type of notifications triggered either by the system or the user. They usually contain useful information but for a specific time during the application hence their sudden appearance with short lifespan. They are not part of the permanent HUD. This feature is carried on to VR applications but still it is not well thought out. Again, the importance of this aspect is in development, is what affects how it's treated.

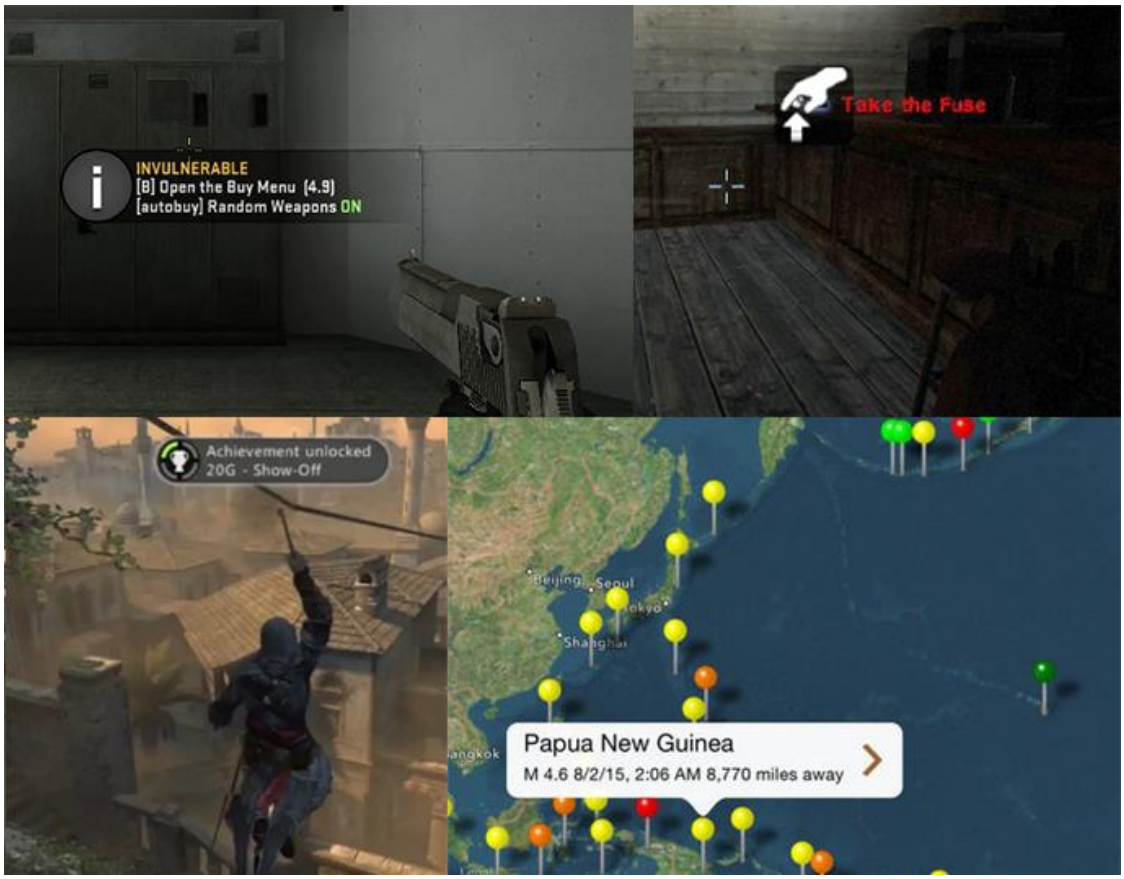


Figure 3.13 Examples of pop-up notifications in games and application

3.3.1 Implementation

The classic notification type is the pop-up. A small UI element containing a brief text and a visual element or a border to make the text more visible. This practice falls also under the direct ports from a desktop to a VR application, where it should not. These elements exist on screen space. This means that they are not part of the 3D virtual world, instead after everything (on a frame) is calculated and it is ready to be drawn on the screen, right on that moment these elements are drawn directly on the screen. This rule of course applies to every object. In the end everything is just an image on a screen. There is no such thing as a 3D representation as it would mean in the real world. All objects in order to have that 3D effect, mathematical rules around a single origin are applied to them. On each frame calculation, before they are drawn on the screen, their position and rotation (and scale) are calculated in relation to the origin and then their updated image is drawn to the display (Figure 3.14). All objects that are on screen space do not obey to that origin but obey to the screen's origin, where it remains untouched. Thus, at the end of each frame, their position and rotation do not alter as the rest of the virtual world does. On top of that they are rendered last in order, appearing to be on top of everything, losing also the priority that they have in the virtual 3D space (a similar feeling would be wearing eye glasses and having a sticker stuck on them).

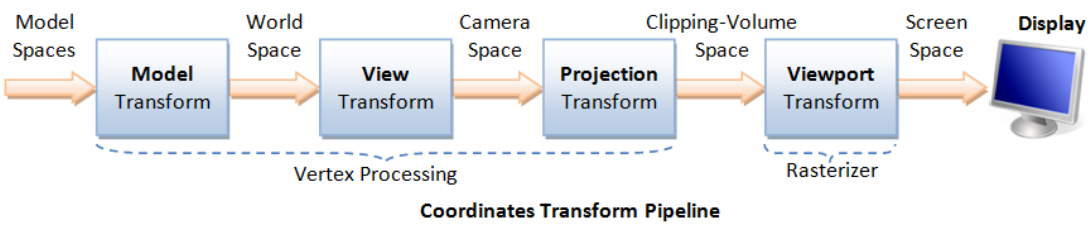


Figure 3.14 Computer graphics pipeline

Everything inside the scene must pass from all the stages of the rendering pipeline. Screen space notifications, don't. They apply immediately on the latest stage. An information like the ones they hold, is supposed to be easily viewed and noticed. There is no reason to be manipulated inside the virtual world as the rest of the assets. All they must do is to appear clearly on the screen. After all the screen is (or at least it should be) at a comfortable distance to be able to read a text. Inside VR, the headset's monitor is no more than an inch away from the user's eyes. Still with the current high pixel density displays and the precise lenses that are on the market, it could not be difficult or tiresome for someone to read a text, after all it works in AR. The problem lies with the use of the display. It is used to represent a whole 3D environment creating the illusion of depth, trying to imitate the real world. As the user is moving around his/her head everything inside the display is behaving accordingly in a manner that represents similar results to what could happen in the real world. Suddenly a notification appears on the screen that does not behave like anything else the user can see. It is stuck on the screen, no depth perception (1 p. 114). Despite being easy to notice, it is not subtle or user friendly. It is sudden and unnatural, something that most probably any user will try to get rid of. That few seconds of potentially mild annoyance can be enough to cause problems.

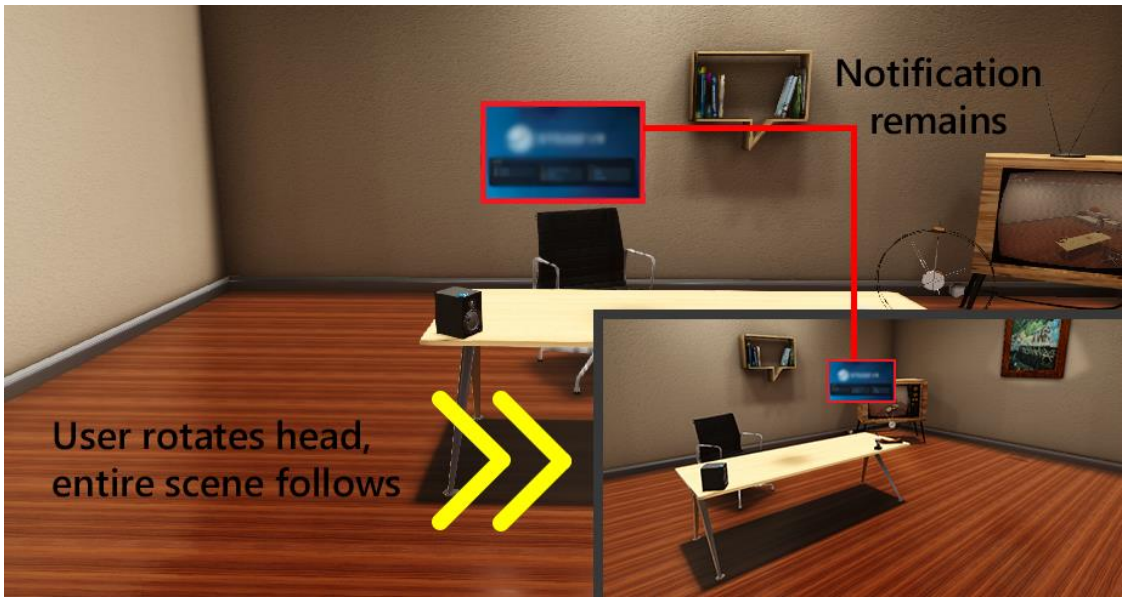


Figure 3.15 Screen space pop-up notifications inside VR

To avoid, at least, the annoyance problem, the ticket is to follow all stages of their pipeline. The UI behaviour does not have to be re-engineered, just to be part of the 3D world (with slightly different treatment). While enjoying the virtual world, a message can be instantly observed from the user but with "kinder" ways. It can spawn in front of the enjoyer with a slight distance offset (reminder that UI is not part of the world), and while he/she is moving around the world, if the message is important it can follow, with a delay, the user (Figure 3.16). In that case it is constantly noticeable, but it's part of the world and has smooth, predictable, animations. The users reactions should be just like when a new text appears on the phone, not as the reactions when the engine warning light (followed by the warning sound) shines from a damaged car (the comparison might seem as an overkill, but from more than two years of demos on VR -starting from 2015 when it was new- that was in general the reactions of many people when an on-screen message suddenly appeared. It felt sudden and unwelcome and users attempted avoidance manoeuvres like having an insect in front of their faces) (11). Once more, a solution to this is straightforward. All these types of UI should appear on a fixed radius from the user, so distance is known, and the only updated variable is the forward direction of the users focus (a.k.a. the VR camera's lookAt vector) which is also known. A delayed lerp function that is triggered after a substantial distance from the UI spawn point, is what remains. Substantial distance because there is no reason for the pop-up window to be moved around the scene constantly, as the users

head is never perfectly steady. Instead of distance measure the function can be triggered when the UI is out of the user's sight (a.k.a. camera field of view).

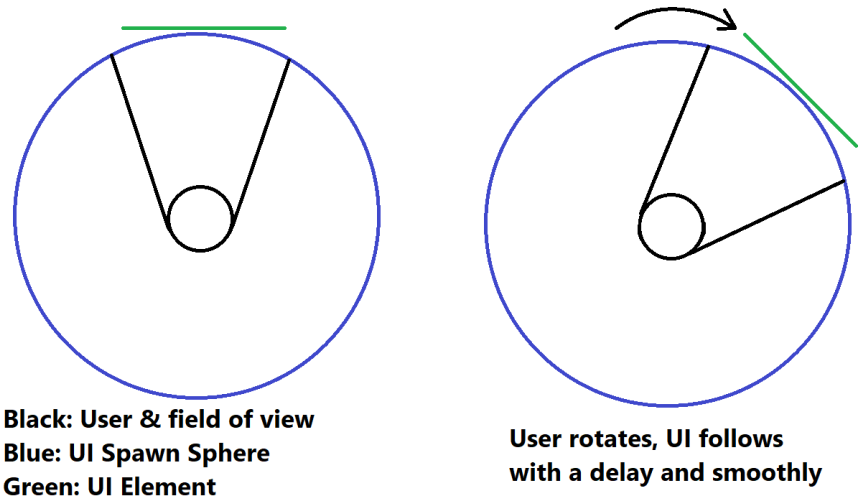


Figure 3.16 UI element behaviour regarding user's movement

Depending on the design of the application, for example how object-dense the world around the user is can affect this implementation's desired results. UI elements will regularly go through objects. The quick-fix is for the UI to be rendered last on the scene, appearing in front of every object (not only in UI is this common but in 3D aspects as well as many First Person Shooter games have the hands and the holding hand to be rendered on top of everything). Still insufficient as inside the virtual world it can be disorienting. It is desirable for everything to appear as natural as possible (yes even now that floating panels are into discussion). If the application scene has many close encounters one suggested solution is for the UI element to understand when a collision occurs and try to avoid it on its own (Figure 3.17). When it detects a collision with another object, its obligation is to avoid trespassing of the other object and still be within reach of the user, thus, the reposition has neither to be far away or too uncomfortably close (do not squeeze into personal space urging the user to take vasive steps – in real life). The way it works is by, invisibly, cloning itself into new instances. These instances then are spread out and form a dome towards the user. The first clone that does not register a collision sends its position to the original element and it moves smoothly (always smooth animations, give the user time to realise) to its new position. This implementation can work for all types of UIs. Keeping the traditional elements, just updated while the design can remain untouched.

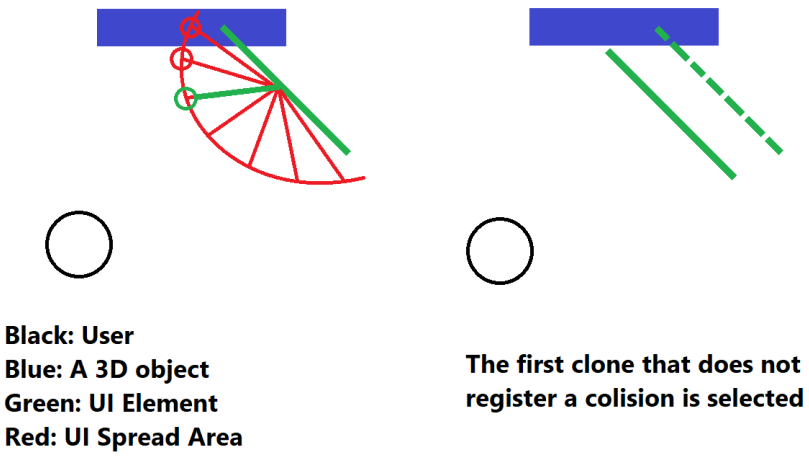


Figure 3.17 UI element behaviour regarding object collision

3.3.2 Designed for Assistance

These types of elements can be used for a variety of applications, many of them rotating around user assistance. However, a notification element is not strictly a small container with a brief information, mostly in a form of a message. It can be anything that alerts the user to achieve a specific, usually short-termed, goal. Notifications that are subtle enough to avoid cracks in the fragile nature of immersion while assisting in avoiding potential annoyance and frustration (1 p. 76) (reaching a place in the storyline where someone does not know what to do).

3.3.2.1 Event Completion

There are not many ways to limit the user's interactions with the visual world nor many approaches to assist one when completing a task, mostly due to the nature of the input (most common case, 6 DoF controllers). For complicated tasks, but not limited to, a real-time progress feedback can smoothen the experience. It can be one more piece to the puzzle to avoid frustrations that have usually do not have to do with the implementation or controls, but with the game's storyline or design.

It can be a simple system that a new element will be instantiated when there is a major update to the current event that occurs. It should have a small subtle animation that both has the role of observation and smoothness. Positioned inside the area where the user-object interaction is occurring but distant enough in the background to avoid stealing the "spotlight" of the current event.

Figure 3.18 Assistive UI element for event completion

3.3.2.2 Object Observation

A well-known and proven to work all these years used, the object notification. Depending on the current event, the correct object that needs to be interacted with can, and should at times, be highlighted. Instantly observed and understood and still be a simple implementation. When all objects that a user can interact with, are all in front, the one that stands out (with a simple outliner glow for example) can be easily spotted. Without any further information it is enough for anyone to engage with the distinct object (23). Designs like this do separate themselves from the real world. Nonetheless they do not alter the user's experience, and mechanisms added to them, despite being unrealistically used, still can be matched with their realistic counterparts. This mechanic can be better exemplified with a tool for a mechanic is required for an event that is located on a table with many other tools. If the developers wanted the tool to be instantly recognizable from the user, then adding a glow would instantaneously make the required tool stand out from the rest. That is the end goal. Despite the tool displaying unrealistic behaviour, glowing out of the blue, that behaviour by itself can be or feel something natural, for example, an extremely hot metallic object that would display similar behaviour (not the most suitable example in this case – enforcing the user to perform a forbidden move in the real world).

Depending on space, object complexity, colours and so on, on this thesis two different implementations are examined. The object can either pulse-flash a color, or its outline to be pulse-highlighted (Figure 3.19). Having a static color is no use as it can be reckoned to be part of the object. In both cases the added color is gently pulsed to attract the attention. It is pulsed a bright yellow as it is a vibrant attention catcher (24) and to further assist one trying to use it, when the virtual hand is within grabbing range, the object with flash (or highlight) green and with a faster pulse rhythm to naturally signal the user to commence the desired interaction. There is just a small caveat with the outline method. It is rendered on top of everything which despite being useful to instantly recognize objects in jam-packed areas, it can lead to user frustration if not implemented correctly due to its lack of depth (1 p. 173).

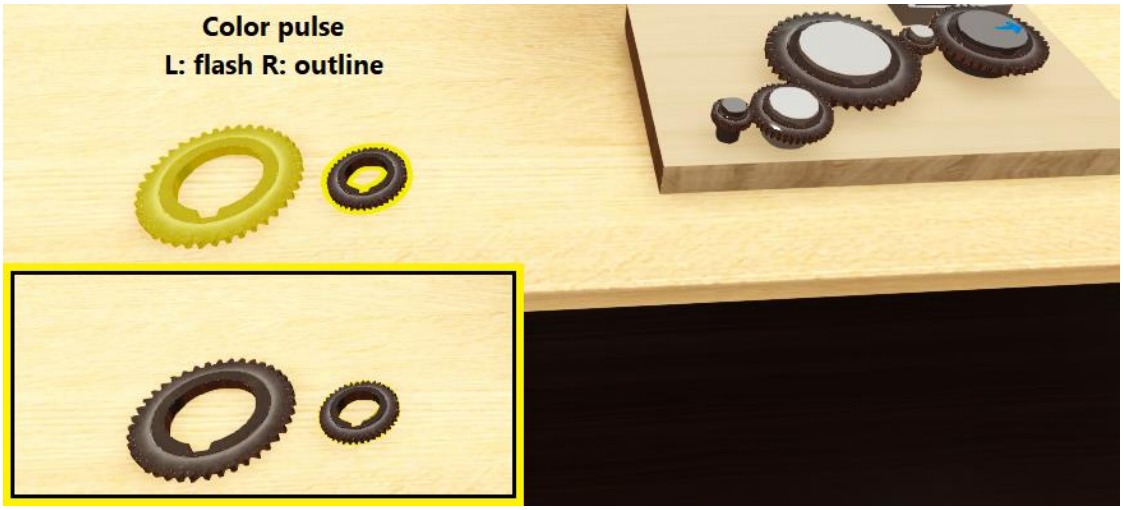


Figure 3.19 Assistive UI element for object observation (above flashing, below outline)

3.3.2.3 Spatial Orientation

On the same page of assisting someone to complete a current situation, lays the spatial orientation. It is effective to highlight an object that is required to be interacted with, but if it is way out of user's sight, the problem persists. For these situations another potential assistance is user orientation. The design should follow the same rules, simple, subtle and smooth. Despite not being part of the real world, the advantages of its use are plentiful, giving a reason not to be ignored (1 p. 242). The element responsible to be instantly recognized as a direction reference can be an arrow. Spawned in front of the user (again with a comfortable -yet easy to notice- distance) with its direction pointing at the object's position. Adding a small animation to make the direction even clearer and design-wise it is completed. Completed, because as an idea it is used a lot in different types of medium and in real life. It is one of the features that a developer can rely on the people using the application to be able to understand.

3.4 Effective Details

All the design choices up to this point were focused to the user's visual aspect. What the user sees and the way it is delivered is after all what differentiates the virtual reality experience from all the other existing methods. It only takes one human sense to be stimulated. Yet, this experience can be reinforced by stimulating other human senses (that current hardware makes it available), sound and "touch". The methods to stimulate these other two senses are common to the interactive world via artificially created 3D sound effects and vibration from the controllers. One can ask what difference these two implementations will bring to reinforce presence (1 p. 46). This is where the brilliance lies, as the desired results for immersion are not based on the complexity of available resources but on how they are taken advantage of. Virtual reality as a concept by itself is based on known and used technologies for years. But simple yet clever chain of ideas (and of course the rapid advances of technology) has brought us here.

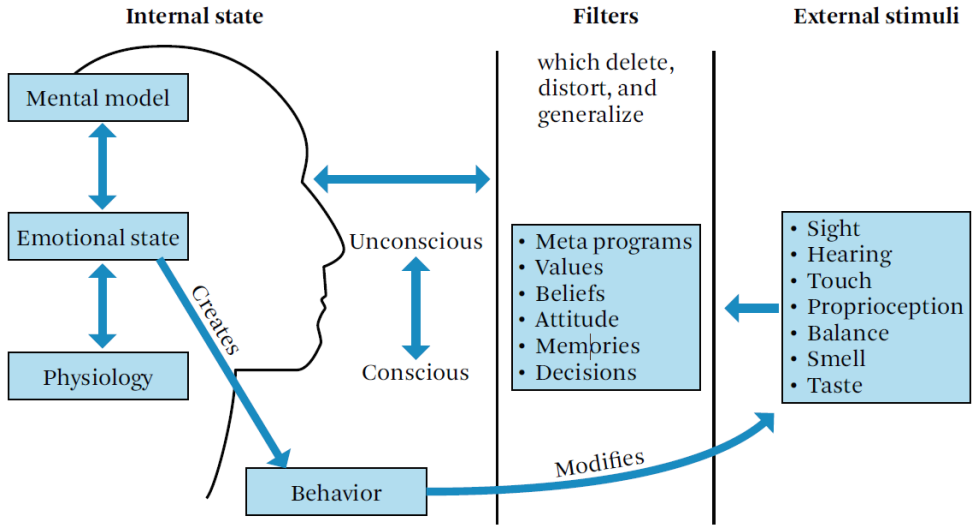


Figure 3.20 How external stimuli affects human behaviour. Image from The VR Book, 2008, author Jerald Jason

3.4.1 Sound

Sound from a technical standpoint has nothing new to bring on this new type of experience. Three-dimensional sound effects are common and truly easy to use. This by itself can be enough to enrich anyone's senses. Improvement will come from the sound design regarding both dynamic and static objects.

From a developer's perspective creating an algorithm that achieves object interaction that has a natural feeling for the user (physical interaction) is something major. But from the user's point of view it still is an incomplete experience. Everything a person can interact with has a different feeling and in this case different and distinct sounds. In most interactive applications (games actually) an object upon interaction creates a specific sound every single time no matter what the circumstances. From a development standpoint this does make sense. There is a balance between spending time and resources and results (e.g. taking tame to altering sounds depending on different criteria for the same object). This applies for virtual reality applications as well, but the scale is different. When hitting something it does produce a sound where its volume is dependent on the item's mass, material type and velocity (and more). The first two can be solved with the creation, as mention, of specific sounds for specific objects, but velocity needs to influence the sounds results. It is something simple as all the parameters are available. The mass of the object, its material, and the range of user's strength (speed or force – there is a limit for humans and the tracking devices that can be pre-calculated). What remains is a generic behaviour that can be attached to all physical objects and upon impact, receiving the force from the physics engine and alongside the rest of the mentioned parameters to manipulate the collision sound's volume (and pitch – what the sound tools can provide). It is far from perfect, but it is a piece in the realism's puzzle without tipping the scale's time and resources end.

```
Function: Object Collision

Precondition: Define object minimum achieved velocity while held by hand (min_Velocity)
              Define object maximum achieved velocity while held by hand (max_Velocity)
              Object contains AudioSource Component

object_Velocity <- Get Object Velocity From Collision()

current_hand <- Get Hand Holding This Object()

clamp_Velocity <- Clamp object_Velocity between min_Velocity & max_Velocity

normalized_Velocity <- normalize clamp_Velocity between 0 - 1

soundClip_Volume <- normalized_Velocity

Play Sound Clip()
```

Figure 3.21 Collision velocity (magnitude) affecting sound strength

Inside Unity the solution (focused on volume) can be abstract enough and stable as this game-engine (nowadays all of them) provides all parameters required. For each object the developer can set up its mass and physical properties (one of them is the sound depending on material) and have a behaviour script attached for the sound changes. Upon impact, if the object is a Rigid-body, the collision is registered inside the physics engine and it can be observed from the developers via code (Figure 3.21). Then the magnitude of the object's (at hand) velocity can be retrieved. Normalizing this value between a predicted minimum and maximum velocity (that will make sense for the user – a 1kg object thrown to the wall with bare hands cannot exceed 100N no matter what the user's strength) and the value produced can be used directly for the sound volume. It is crude, there are more things to consider, but it is a base to start that produces observable results.

It should be a constant reminder during development that the result is not just to create an experience but an "alternate reality". It has to include everything that will make the enjoyer feel like it is real (1 p. 239). One key area is in overall sound design (25).It is common to the gaming world the importance of ambient sounds. In the same spirit, this needs to be amplified for VR applications. The bottom line is that in real life there is never dead silence even if it is perceived that way. In a well-designed and executed game there are constantly ambient sounds playing I the background but depending on user's action there are spots of "zero" decibels (results may vary depending on game). This should be out of the question when coming to VR sound design. There is no need for a new implementation or even further development, just good design. It is subtle but crucial for immersion. Sound achieved immersion is desired and pursued from developers of conventional applications (non-VR), thus making it even more important for VR applications since the foundations for a great immersive experience are already there.

Figure 3.22 Project's VR environment, an office

To bring the idea to reality the sound design can be examined with this storyline's scene. It takes place inside an office, a typically quiet place. To have always at least one sound playing its design should be consisted of the inside and the outside of the observable environment. Inside environment is consisted of objects-producing sounds (e.g. a loud clock) and outside is a selection of produced noises that fit to the environment the user is in, for them to be instantly familiar. In this case, inside the office the permanent (fail-safe) sound is the interactable clock. Visible, familiar and subtle enough to - potentially - not bother during the VR experience. Others include a small speaker and a static noise display in the back. External sounds can be anything that will be familiar with any user, as long as they do not interfere with the experience (annoying sounds).

3.4.2 Touch

The remaining human sense that can boost the results closer to the goal is touch. It is a very limited advantage as it is not fully used as vision and sound. User's hands are reserved by controllers therefore - technically - there is no touch at all (matching real life feel with the virtual objects inside the scene). The solution is to mimic touch with the use of vibration from the controllers. Having a vibration as an extra feedback every time an interaction occurs, does improve the experience, but it may not, at first, seem something that assists with immersion. When holding a controller for anyone it will be an instant observation to the limitations of the system (in this case touch). Implementing a vibration feedback mechanism that suits the environment and the interactions within it, by changing duration and vibration strength, is more than able to assist with user engagement. As described in the sound section, behaviours that mimic real life ones can "trick" the user into absorbing the current situation are real. A vibration to the controller where its strength matches the force of the object upon impact is a correlation what can happen instantaneously inside anyone's mind while encountering it. The human brain is excellent at finding patterns and filling on its own the gaps (26).

The algorithm for the vibrations works the same as in sound mentioned in the previous section. Upon collision, the object's velocity magnitude can be retrieved and normalized (with the range of min and max achievable velocities). Then depending on the hand that holds the object (something simple to find as the interaction algorithm attach to the object provides this information), this normalized value can be fed to the controller for the vibration. This project has a function for the vibration that works with any Steam VR compatible controller.

```
1  Function: Object Collision
2
3  Precondition: Define object minimum achieved velocity while held by hand (min_Velocity)
4                Define object maximum achieved velocity while held by hand (max_Velocity)
5
6  object_Velocity <- Get Object Velocity From Collision()
7
8  current_hand <- Get Hand Holding This Object()
9
10 clamp_Velocity <- Clamp object_Velocity between min_Velocity & max_Velocity
11
12 normalized_Velocity <- normalize clamp_Velocity between 0 - 1
13
14 Trigger Controller Haptic Pulse(current_Hand , normalized_Velocity)
```

Figure 3.23 Collision velocity (magnitude) affecting controller vibration strength

This solution is viable only for a specific type on interaction. It is due to the limitations of hardware. This project is focused on solutions with the minimum commercially available hardware. This is mentioned, as there are methods trying to enforce the touch sense in VR with exoskeleton gloves or robotic arms. Solutions that help significantly more but have their own disadvantages that, at least for this point in technology's progress, will not reachable from the public. Viable solutions are tested (solutions that are available to the market now) to enhance the presence via vibration. Suits that one can wear and have many vibration motors scattered around. Vibrations will be felt through the entire body which can be game changer for specific VR applications. The idea is the same as with the controllers, trying to mimic the real world and have the brain being responsible for the connection of what feels real and what is.

3.4.3 Always Keep a Distance

A design element that does not belong to the VR environment is bringing the foreground of interactions close to the user. Developers must remember to always keep a distance of what is going on from the virtual camera as it is not just a window to all action, but it represents user's eyes. This is relatable to games being ported to VR. A screen, where the game is happening, is away from the enjoyer (or at least it should be). If sudden and/or intense moments occur during the game in front of the virtual camera (e.g. a close-up explosion), this will not affect the user's enjoyment (generally speaking). The same idea cannot be applied to VR, both for physical and mental reasons.

Human eyes can focus only on a small area and the peripheral vision is just a blur. Having a display close to the eye is a problem as all information in the visible space is crystal clear. If an object is close to the virtual camera, and therefore close to the eyes, it will be difficult to see and focus on it as there is no real depth (solutions are on the way with eye-tracking headsets rendering clear images where the eye focuses and blurring the rest) leading to user frustration or worse eye fatigue (27). There not a lot of evidence or a clear study supporting eye restrain inside the VR but in general it is a well-known issue (28). An "off the record" rule is to have a limited amount experiencing the world of VR. Despite the rule making sense, it is not a solution that can support the evolution from VR. The solution is careful design as user frustration (lack of) is tied with presence (1 p. 200).

3.4.4 Confined Play Area

All available virtual devices have confined play area where the user can create the available physical space and then displayed as a grid inside the virtual space. It is not visible, apart from when the detected controllers are near it, then a bright coloured grid appears (depending on system there are variations but with minor changes). It is a good implementation as it shows precisely the play area without obscuring the view of the rest of the game, and it is appeared only when "it is needed". This is an excellent solution to keep the user in that area to avoid contact with the outer world, and it takes off from the developer's mind design restrictions, as everything is done by the device's drivers. Yet, having as goal immersion and presence, this guardian system the devices use is unwelcome (1 p. 213). Apart from the aesthetic of the system being a disadvantage (like a prison cell), it is truly an observable reminder that one is trespassing the bounds. This is pleasant in terms of avoiding injuries but unpleasant for immersion, as it also reminds the nature of the experience (being a video game) and can lead to breaking the involvement (a personal note, I have never seen a VR demo setup in shows or exhibitions with the guardian enabled).

Of course, this system is advantageous, but it should not be regarded as the solution to the play area confinement from the developers to avoid further design, especially when it is solely on user's preference to keep the system running or not, but only as a fail-safe. The restricted area should be visible to anyone inside the VR, but it should be part of the virtual world. Blending in with everything else, but still easy to notice and understand its purpose. The playable area naturally depends on every user and the available space, so it is difficult to have a universal design for every case (standing only or room scale – the latter being a headache on its own). The design could be consisted of two parts. The first one would be a constant visualization of the confined area, subtle but easy to spot, and the second one would have the role of an alert when the user is out of bounds, thus being the replacement of the default guardian.

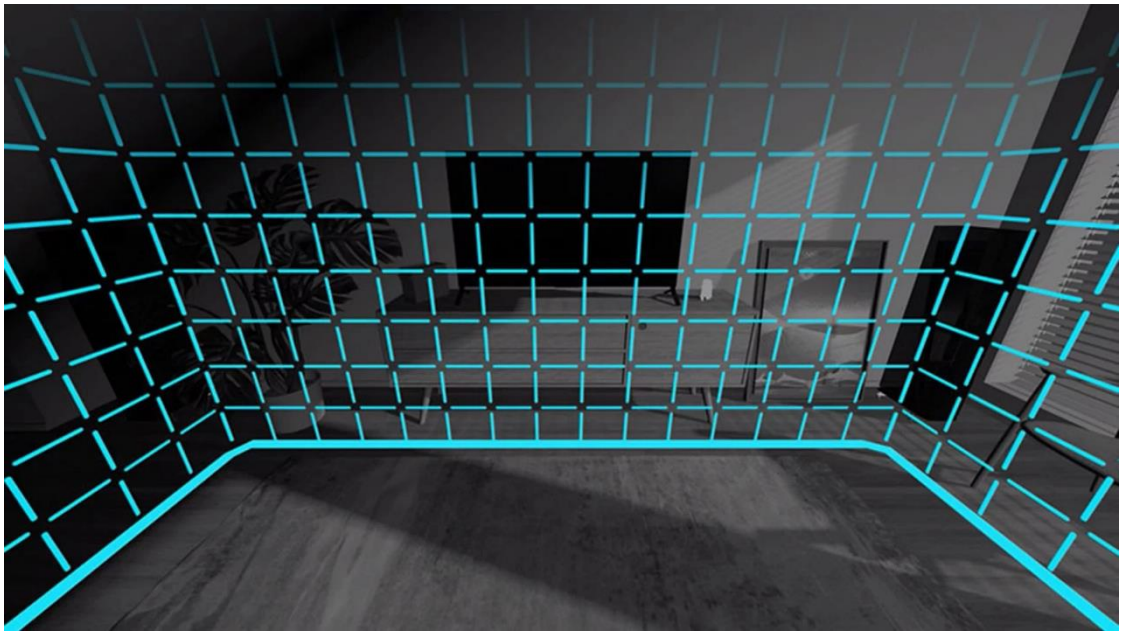


Figure 3.24 VR headset's borders visualised

For the two-part design to become more understood, its ideas will be applied on an example virtual reality application inside a living room (Figure 3.22). The virtual living room has an available space for the user to move about but that does not necessarily match the physical one. The first design choice to constantly and subtly remind the user the physical confined space and blend in with the environment would be a carpet. The carpet will be scaled according to the minimum base area created. It is scalable (up to a point at least), simple to implement and it can be as an intermediate warning before going out of bounds (Figure 3.25). But when that time comes and the user must be aware, the second design that replaces the guardian can come forward. In this scenario there are spotlights in the ceiling that are off. When the user is going out of bounds, a spotlight, that is bright enough, will turn on and illuminate the area, and a message can appear to notify the user that is out of bounds and to explain the connection between going out of the confined area and the spotlights. From the second time on only the spotlights can be turned on or off. An instant warning to avoid injuries, but still part of the world to keep anyone immersed.

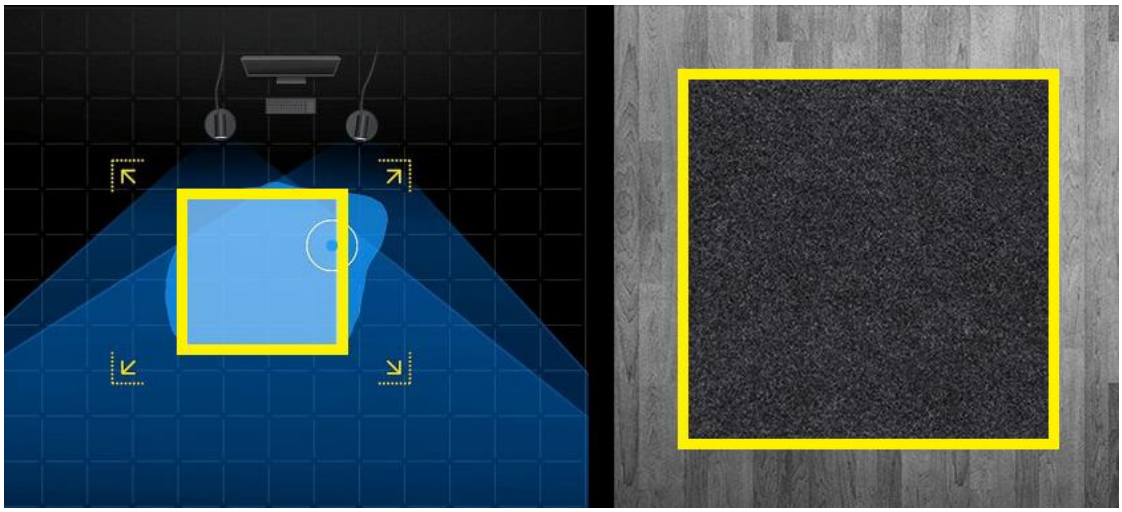


Figure 3.25 User-created play aera size translated visually as a rug (in this example) inside VR's scene

Lastly, there is one last type of bounds the developers must take notice. The ones that are inside the physical available space. Objects big enough that will be in walkable distance for the

developer to enter through them. There are no ways to prevent object intrusion from user. This is also an issue a good design might not be able to prevent. A sufficient implementation is to fade to black vision with a message notifying what is happening. This goes against to what lengths are already taken for the beloved immersion. Nonetheless, going inside object will be most of the time user's curiosity to try to test the game with "unnatural" behaviours (except if the design is not good and users by mistake enter through objects) thus the immersion is faded away by choice.

4 Conclusion

4.1 Summary

Virtual reality is tied with, but not limited to, presence. The scope of this thesis was to search and develop, design and code implementation principles for user experience for this new medium. If this could be summarized in a small sentence, it could be "developing applications for virtual reality starter kit". It contains ideas that surround, design-wise, the user and the virtual environment; ideas, simple yet effective that contribute to this goal. Many of these design standards already existed (especially in the world of gaming), but also a lot are purposely created for VR, with a constant reminder that any of them must be able to apply in any type of VR application. What was introduced can be broken down into:

i) Mechanics for physical user-object interaction,

ii) Assistive but hidden methods for naturally feeling interactions

iii) Scene-oriented interaction design principles

iv) User Interface suited to VR

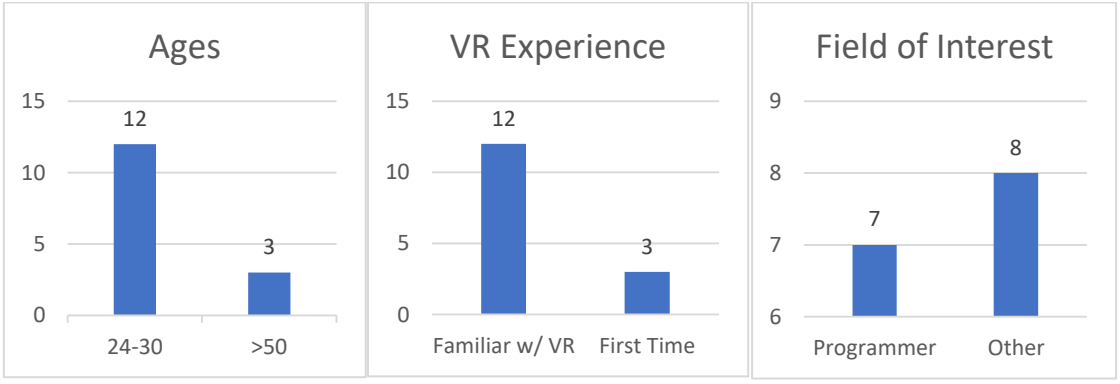
v) Effective details & manipulating human senses to strengthen immersion

4.2 Evaluation

User Experience evaluation is not simple. Users must come from all backgrounds and test an application numerous times to understand what each one experienced. Virtual reality is a very personal experience thus leading to a wide range of results. Furthermore, all the design elements analysed in this thesis cannot be evaluated "on demand". For this evaluation most of these proposed elements are separated into segments and implemented to one VR application inside a virtual office where all users entered fulfilled small tasks. External applications were also used, Steam VR Home to evaluate also ray-casting menus, and environment virtual guards from Oculus Rift application and Steam VR.

The application was the same for all users, but two different HMDs were used with different controllers to isolate hardware experiences from some of these evaluations. Oculus Rift and HTC Vive were used (gen 1 commercial products). The application was tested from 15 users. More specifically 7 were programmers below thirty years old, where six of them had experience with VR. From the remaining 8, five of them were under 30 and had at least once experienced VR and remaining three were above 50 years old with only one having experienced VR.

Tables 1-3 Participant age, participant experience w/ VR, field of interest



This usability evaluation was focused on qualitative data due to the nature of this test and the small number of participants (29). The first step was the creation of a standardized questionnaire for all participants. For each segment a separate question or a satisfaction score and an overall satisfaction for the entirety of the evaluation was given. Each problem every user faced, was

noted to observe similarities. During user evaluation, all were recorded and studied to understand movements and familiarity with the VR simulation. At the end all participants were asked the same follow-up questions and a free self-evaluation summary.

1. The scenario for the evaluation starts with users being inside the virtual office and all of them being left alone without a purpose to get familiarised with the environment. For all of them the default from the headset's environment guardian is enabled (to understand the guardian, all of them before created the virtual space from each headset's app). The only difference is that 8 users are inside an interactive office (where all items can be held, produce sounds, etc.) and the other 7 are in the same office but nothing is interactive.

2. First real task is object manipulation of both large and small items (small equals to controller size or less). First interactions are with large physical objects were the implemented manipulation is with traditional methods such as game-engine provided parenting and scripted parenting without the physical properties being disabled. Then the same objects are manipulated with the implementation of physical interaction and the support of both hands. Once this task is completed it follows to the manipulation of small objects. These small objects all have physical interaction, and half of them have the assistive algorithm for fixed posed at hand and the other half do not. At the end users evaluate ease of use, natural feeling of these interactions, and in the case of small objects if any unnatural behaviour was spotted. The task is consisted of books that need to be placed on a self or thrown away in the bin, and then followed by small darts that users need to grab them (as darts are supposed to be held), and throw them at the dart board.

3. After the establishment of natural interaction, next part is the assistive algorithms for these actions. Again, users have to use small and large items. All items have the algorithm responsible for altering object sound and vibration depending on user's applied force (the stronger one hits something the louder sound it produces and the stronger feel the performer has). Small items require subtle movements and big items require fast and "bulky" actions to test how noticeable the difference in external stimuli is. Small items are gears that need to be inserted in an electric board to fill its mechanism, and large item is a heavy hammer to be used in a whack-a-mole game.

4. Remaining to interactions and their assistance, this task evaluates the UIs role. The same task with the gears above must be repeated. The gear insertion task is by design a complex task, that without any knowledge it is very difficult to be completed (many small almost identical gears, all under the same space). All users (even ones that did not manage to finish the task without the visual aids - 11 out of 15!) were reintroduced to the this same task but with the assists enabled. These were consisted of ghosting gears showing the correct way one could insert them into board (also showing the specific spots), and for the correct gears, a flashing yellow halo around them appeared to separate them from the mis-fitting gears. Times were compared, user evaluation of natural/unnatural characteristics of these aids, how helpful they were, and if they have a role for training purposes.

5. From assistive UIs, the story then goes to UIs in general. By this time the evaluators are in near completion. They are tasked to open the applications UI menu and restart the application. Upon restart they are told to open it again and navigate through all the events they had previously completed till they find themselves to the spot they restarted the application. That completes the application tasks. In the end, the Steam VR Home is used for the final part and all participants are asked to open the home menu and change their avatar head. The reason behind this, is to use Steam VR's ray-casting type of UI elements. Later, the experience between UI interaction types is measured in terms of ease of use, natural feeling and personal preference.

6. In the beginning, where all were just getting to know the virtual environment, it is described that each headsets environment guardian was enabled. By the time participants started the real tasks of the evaluation, those guardian systems were disabled, and the applications embedded environment guardian was enabled (only time where the evaluation was halted to change the state of the guardians). At the end of the demo, everyone was asked to compare both implementations. Questions were asked about the level of understandability each implementation had,

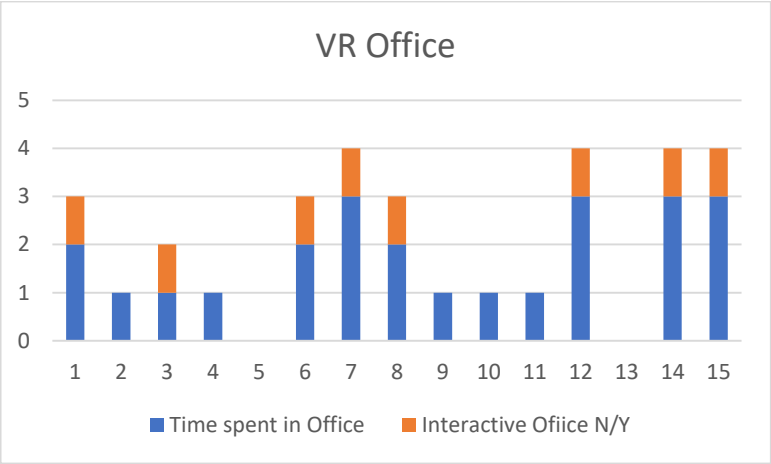
how annoying it was (or lack of), and the preferred method each user would enable for their private VR sessions.

4.3 Results

Results are shown in time and satisfaction charts (with user evaluation from 1 - 10). All these values will be accompanied by user comments grouped down to categories. Both are important to get the overall picture since it is difficult nail down experience satisfaction with just numbers. All charts are explained with each user's evaluation (in X axis). The three elderly people are the last ones in all charts, because in some cases there were interesting differences from the younger audience and lastly, the ones affiliated with VR will be pointed out.

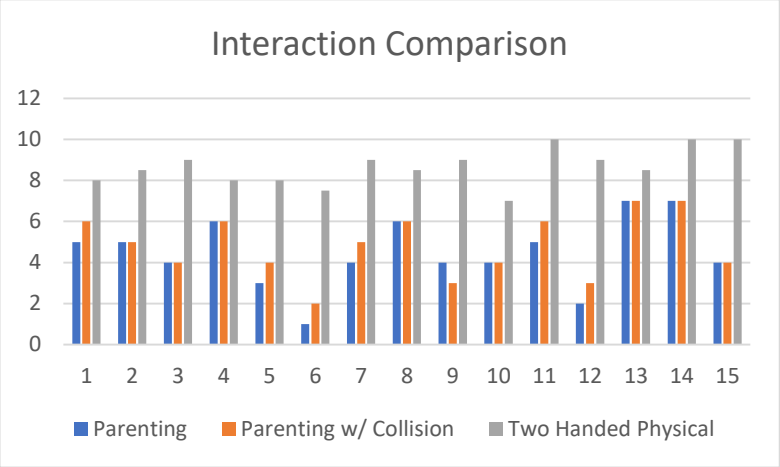
1. Differences between users that were left inside the virtual office, half with all objects being interactable (more than just be able to be held) and the other half with all objects in room being static. All participants immediately after looking around to understand the place they were in, the first movement was to reach one of the desks items (a statue, a Bluetooth speaker, a clock). Users inside the interactive office fiddle around with objects and after some minutes, after having nothing to do asked to begin the tasks. Users without the interactive objects, also spent on average time inside the office; when they tried to grab items there were a lot of attempts till they could figure that the item is not up for grabs, which is what in design a developer should avoid. Interestingly most participants opted to grab first the only object that was producing a sound, the table clock.

Table 2 Time spent in office. In orange are people that were in interactive office.



2. The first real task focused on object interaction. All interactions were the same, reach hand above object, press button to grab, release button to drop. For users, the differences were in physical behaviours (or lack of for the first two types, both involving parenting). Due to this similarity, some found all three implementations comparable (Parenting avg: **4.5**/10, Parenting w/ Collision avg: **4.8**/10, Two handed Physical avg: **8.6**/10). Otherwise the two handed, physical interaction felt, as described, more natural and was a preferable experience. This result came mostly from interacting with static objects, more specifically when the books were hitting the desk or the shelves. First implementation had the ghosting effect and second implementation had a similar effect but with random unnatural collisions. The third one, the velocity-based method, behaved as intended, colliding and interacting in a physical way with the static objects making easier for the participants to place the books on the shelves as they felt as physical-real objects. One note was that only four out of the fifteen participants tried to toss the last physical book inside the bin, the rest just dropped it from above. When asked if this was a move copied from the previous methods (non-physical interactions thus tossing items was unavailable) or it was a deliberate decision many of them were not sure. The ones that were sure (copied movement from previous two implementations), their field of interest was programming.

Table 3 Comparing parenting (w/ & w/o collision) & two-handed physical interaction



In the second part, were all were tasked to grab small items in a specific way in hand, results are more differentiated, because user habit and convenience came into play. The comparison was on fixed posed on hand for half of small items and free (user defined) item pose for the other half. Most users preferred the items with the fixed pose (Table 7, orange lines) on hand and did not really noticed the item itself readjusting at hand (it is a 0.5 sec animation), and the ones that did (6 out of 15) had nothing negative to comment (apart from one user where during the auto adjustment, the item at hand would collide with other items in close range – that's software issue that will be resolved in future update). Regarding the non-self-adjusting items, despite for some taking a significant more time to grab correctly the small items (Table 6 – blue lines), this was more appreciated, and preferred because of the freedom it game them. From familiar with VR participants, something else and interesting was observed. For both free and fixed pose items, before grabbing them, they adjusted their own hand (angle) to match with the pose the item is supposed to be held and them grabbed them. As a result, two of the participants did not observe any difference between the items. Lastly, the grabbing aid (reaching and trying to grab an item while miscalculating the depth, item's colliders expand) was triggers only to one of the participants (one of the elders).

Table 4 Time in seconds to adjust item pose at hand

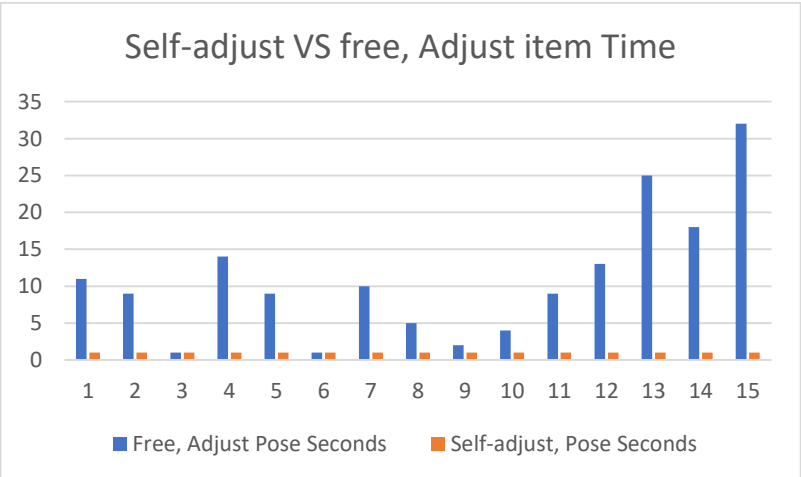
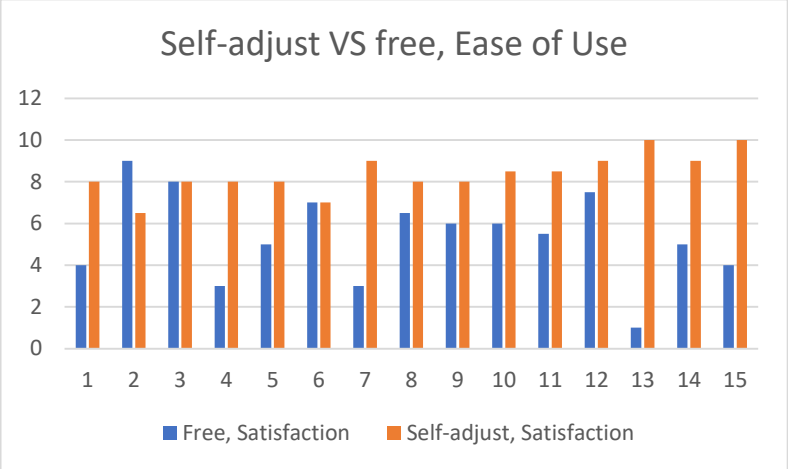


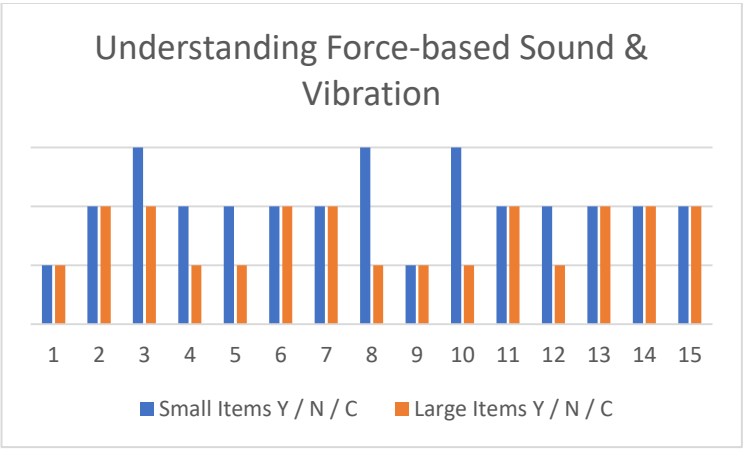
Table 5 Small items, free vs self-adjusting user satisfaction



In those charts (Table 6, 7) it is observed that elder people (13 - 15) were the ones that took a lot of time to manage to adjust the free items at hand and gave the fixed pose items almost a perfect satisfaction score. Also, the people that adjusted the free items with ease gave similar scores to both types of items.

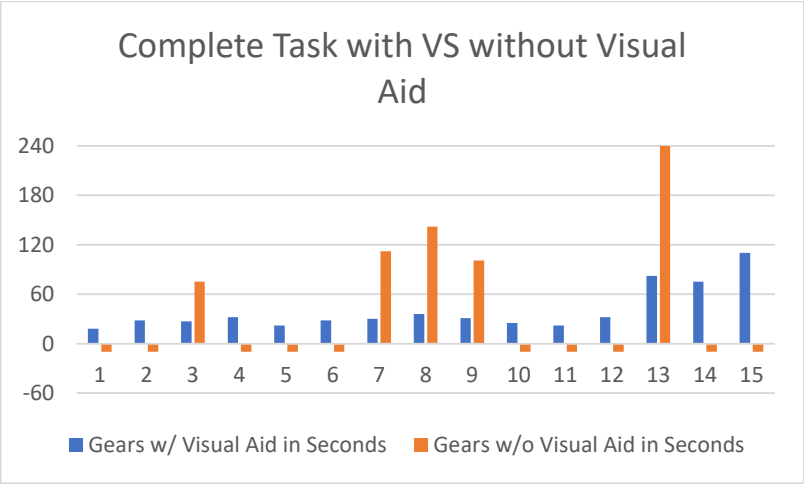
3. Force based strength of sound and vibration tested on small and large items. User were evaluated in if this behaviour was noticed from them and if it added to the overall experience. Graphs have yes, no and confused options. It started as yes VS no but during evaluation for the small objects, some users found to be confused with the vibration mostly and the small items, as they figured that it was a warning from the system (small items were a selection of gears where used needed to find the correct ones and place them to a board; vibration from them was thought to be a system warning for using a wrong gear). All three confused users were familiar with VR or with gaming, which could be a factor to their confusion, or a bad design of the application. Furthermore, since the large item was a big hammer, most participants used it vigorously, and all that did in such a way, did not understand the difference in sound or vibration (a factor could also be that the whack-a-mole game chosen for this, was a bad idea as excitement and short gaming bursts do not go along with experience evaluations). Rest of users did understand the difference and gave a positive feedback to their experience. Despite the mini-game possibly preventing users from using the large item delicately enough to notice any difference, still there a lot more participants understood the difference in sound and/or vibration from the large item rather than the small ones (Table 8, 2 understood with small items, 7 understood with large items). In the graph below the first horizontal represents the users that understood the difference in sound and vibration, the second line is for the ones that did not understand and the third one for the confused (mostly by vibration) participants.

Table 6 Understanding sound & vibration: Yes / No / Confused



4. Next part has small items, again the gears, but redone with visual assistance. Users evaluated the importance, assistance, and blend with the rest of realistic virtual world. Since, by design, the task was difficult to complete, users were given the choice not to complete the task (Table 9, the DNF users are marked with negative values in graph). Three out of four people that completed the task without the visual aid also preferred it in that state because it game the reason to try the whole task. Also reported, from the same people, was the feeling of being "on rails" with the visual assistance (despite being still free to do whatever they wanted, the visual aids where not that simple to be ignored). One of the elders found that flashing gears to be unsettling (as too unnatural) and the rest, had no problems and reported positive feedback in effortlessly completing the task given. One issue that occurred (design issue, will be fixed in future update), what the green flash of items. When the hand is within grabbing distance of the yellow-flashing item, the color changes to green and pulses faster to notify user that it's ready for interaction. But if their hand was closer to another gear then that gear would be held by their hand and not the green flashing one. The green flash was widely understood as that object "was locked" up for interaction with the hand making some of the users "feel" more comfortable and have them reduced their amount of "effort" to interact with that item.

Table 7 Time took users to complete task with and without aid (negative is DNF)



5. Comparison of UI elements, as menus, close hand interaction VS ray-casting. This is the only task that required the use of an external program, for comparison, Steam VR Home. This was the task with the most differentiated results, where satisfaction is not connected with the preferred method. There are no graphs in this scenario as the point was to match the two fields mentioned but results beg to differ. One note for the ray-cast, the elders were not able to finish (all three!) mostly due to visual issues, as was reported later from them. Most of users liked the personal interaction with the UI element (this thesis implementation). As reported, it was faster and felt more real due to similarities in interaction. In visual terms, big buttons with understandable icons was a plus, but the issue was the floating text. It was noted that a permanent text explaining the button's feature would be more helpful. The only scenario where the ray-cast was suggested, was for multiple options that all could be easily viewed from a distance. Again, for ray-casting, despite not being a satisfying experience for almost anyone, for many (reported 9 out of 12) young users it was the preferred method. The reason was that all of them were quite familiar with computers and using mouse as input all those years in their lives. Ray-casting menus experience was similar to them with using a mouse, thus preferring this, not so satisfying but used to, method.

6. All tasks were completed while the embedded environment guardian was enabled, apart from the initial impression of the VR office where HMDs guardian was enabled. As a reminder, the guardian in the virtual office is two spotlights showing user's hands when they are outside of the confined area, and the area itself is marked as a rug (matching is size) under the user. Results are universal. This system was preferred and despite alerting the participants it did not

made them feel uncomfortable or "retrieve" them from the experience. Unfortunately, this design lacked the immediate understanding of its purpose (Table 10, blue lines VS Table 11, blue lines) than the default environment guardian system the HMDs were using. The outcome of the last evaluation is that most people would prefer the embedded guardian (Table 10, orange lines VS Table 11, orange lines) but a better way to instantly understand its purpose requires rethink. Lastly it is observed that participants were split between the two implementations. Being satisfied with one showed in the results that the other was below average in preference (accounting user report not only evaluation score. Embedded guardian avg: **7.1**/10, HMD guardian avg: **4.75**/10 both in satisfaction).

Table 8 Understanding VS satisfaction of embedded environment guardian

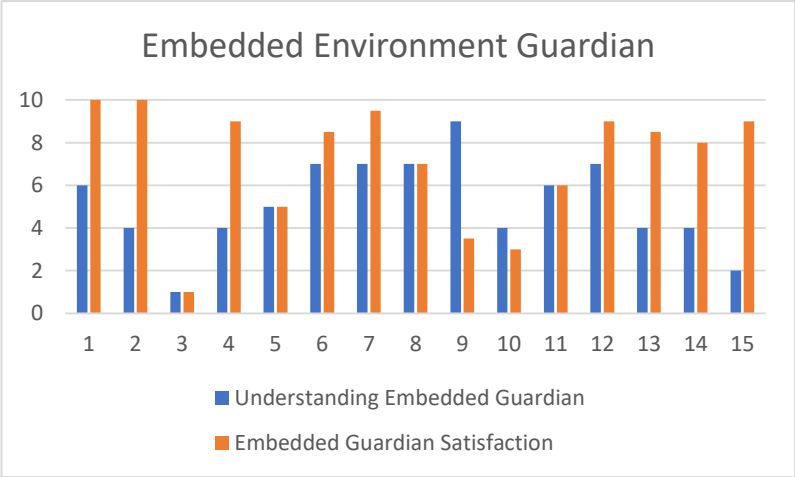
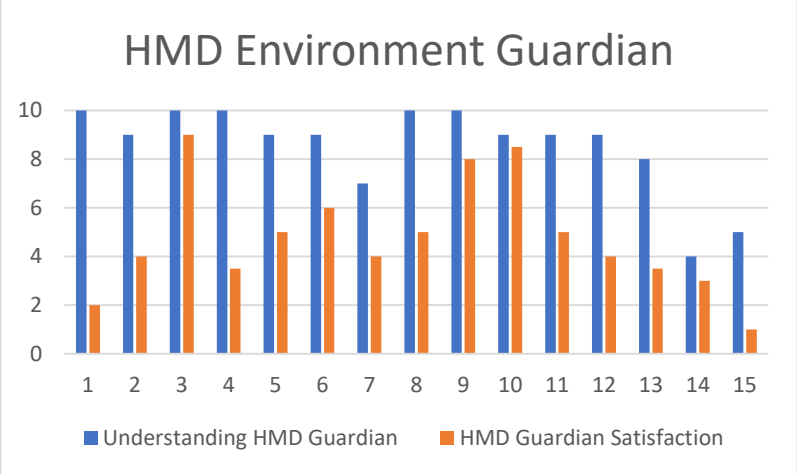


Table 9 Understanding VS satisfaction of HMD environment guardian



4.4 Future Work

Designing the perfect user experience is a concept with no roof. There is always room of improvement, yet there are immediate future plans to further improve on the design. Some involve further iterations and polishing of existing designs and some are entirely new concepts.

Existing Iterations:

- Confined Play Area:

  The ground area of this project which forms the foundation for the guardian system if fixed. Since this thesis can work with any type of headset that is supported from Steam

VR, the immediate goal is to retrieve the space size (play area) the user has created and scale the in-project area (the rug beneath the user) to match the dimensions.

- Fixed Interactions:

  As described for small objects that usually have a specific pose when held, there is an algorithm that, upon grabbing, readjusts the objects to that specific pose (w/ easy to use implementation for the developers). The same algorithm could be used for any type of object, single or double handed, that can be held with specific postures. For this it requires also several different posture from the hand itself (it has only two now pitch and grab). Those can be premade as the existing ones or calculated at runtime depending on the object surface (explained in new concepts below).

- Object Observation / Spatial Orientation:

  At this point these two, as helpful as they can be, they are static. This means that they are immediately shown to user. If the user knows what to do in the next step, they can be annoying till they are turned off (by interacting with object or completing the event). A future adaptation during use of the system can be a solution to this that when specific criteria are met, these visual assistances will take part. Criteria will involve user data from timing, movements and potential interactions.

- Assistive Interaction:

  When there is an effort to grab a small object without success (potential depth underestimation), the invisible collider of the object expands to assist with its reaching capabilities. This was simply implemented to prove its work. The algorithm will change, and it will be adaptive during use, in the same way as described above in Object Observation and Spatial Orientation.

- Sound / Vibration Object Interaction:

  There is a simple algorithm to prove the concept that depending on the force of use (collision hit) of an object, the same multiplication will be applied to the sound and vibration strength that it will produce. The developer must predefine the minimum and maximum force the object can achieve at this point. In future iteration the developer has to do nothing, and the algorithm will automatically calculate the sound / vibration strength at runtime depending on the object's values (Rigidbody properties like mass, drag, etc.) and the external forced applied. Developer assistance and more believable results will be the end goal of this.

New Concepts:

- Full Body Tracking (In Progress):

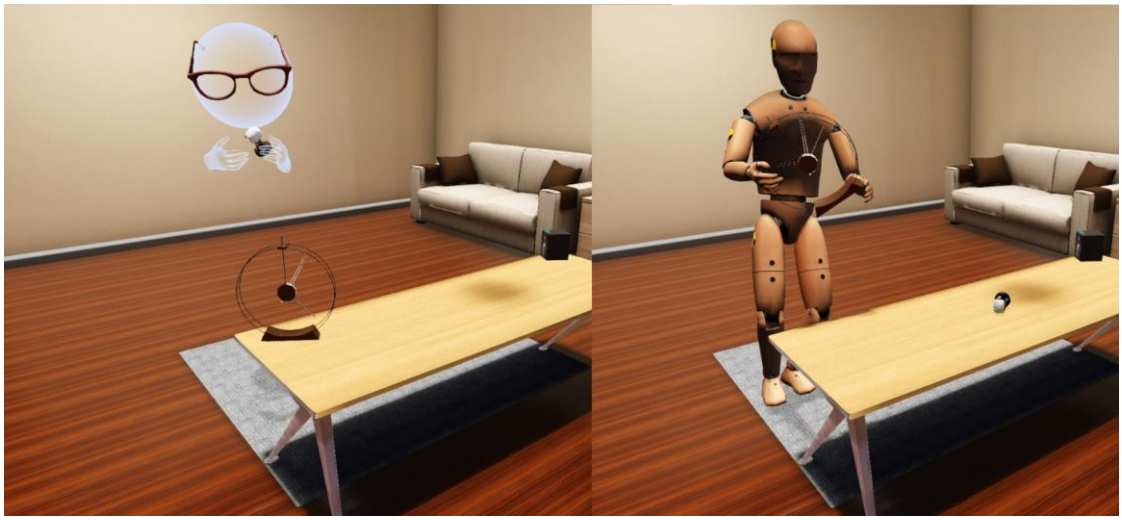


Figure 26 Showcasing first attempt to embodied physical interaction

There is a difference between seeing just the hands and let the user's imagination fill the rest and having a full body simulation. There are algorithms with full body inverse kinematics and predictions to its movement. Even though these algorithms are an efficient solution (user just wears the headset and rest is software), full body is difficult to predict and have realistic results, thus being better for the implementation to be achieved with body trackers. This is the most difficult part of future work, both for implementation and careful design and testing to prove that full body simulation will have a great impact to presence to prove all the extra work (not as much from developer side but from user's side. It stops being a "plug and play" situation having to wear so much gear, same goes for portability, etc.). That said, due to the importance of embodied interaction, this part of the future work is already underway. Currently a full body with IK solvers is implemented to this project with three different targets that are constantly observed to update body's motion. Head and the two hands are the targets being directly connected to VR's hardware (headset and controllers). The next steps include the detailed hand animation (either the existing one with button press or the new implementation of realistic physical hand, see below), also include better hand prediction depending on controller movement. From user side, one important puzzle to figure is the avatar height adjustability depending on each individual's height.

- Realistic Hand Interaction:

  At this point user's virtual hands have three prefixed poses that change depending on button press. A resting pose that matches the pose of the hand holding the VR controller, one pitching pose and one grabbing pose depending on pressing the pitch of grip button respectively. This implementation is standard in most VR applications but it's far from optimal because upon interaction, part of hand many times goes through objects and has always a specific pose. Dialling in that it is user's hands, almost always in close – focus range, it is a field that still is not fully implemented.

  There are a lot of different approaches to solve, but one comes close to realistic hand interaction (30). The way it works is for the hand to be in the resting pose, and when it is within interacting range with the object all finders will automatically start to close and will stop till the tip of each finger "touches" the object. The hand will adjust itself to any surface without the need for premade animations. A small test is already underway (October 2019) where fingertips cast rays and when are close to the object in question, the fingers start to move till the ray hitting the object reaches the minimum distance provided from the developer. Inverse kinematics is applied to fingers. This test is not included in the provided project.

Project:

- Upgrade Unity:

  This project is currently running on Unity 2017.2.4. It is an immediate goal to update to the latest supported Unity version. This project has three assets that might be affected with versions with the most notable being Steam VR. The idea is to convert the whole project to Unity package. Thus, it can be supported from many versions and it will also be easily installed, just with a few clicks.

- Fix Issues:

  From the evaluation some small unknown issues occurred. In the next update most will be sorted out. One important example is the automated pose fix at hand for small items. Auto adjusting at hand means that the items rotate to their end pose. Since this animation lasts only half a second, if their initial and final pose have a big angle difference, during the animation, they will gain a lot of momentum (since they keep their physical parameters). If another item is going to be within range of the self-adjusting item at hand (especially if it is a light one), they will hit with each other and there is a change that the non-held item will "fly" away at distance which is for the users an unnatural behaviour.

- UI Orientation Depending on Head Rotation:

  In the section of UI behaviour, one of them was focused on UI element orientation depending on user's head direction. It is a simple method to explain and despite the straightforward implementation, it is removed from this project because it does not cooperate seamlessly with the UI's collision avoidance algorithm (UI rotates, collides with object, tries to avoid it, loses head direction, re-rotates, and so on).

Evaluation:

- A more detailed future evaluation is under preparation. It will involve more users in a two-step evaluation. It will be split between the current implementation with the psychomotor interaction and with the completed embodied interaction to weigh user immersion in both scenarios (embodied as in having a full body inside VR).

Bibliography

1. Jason, Jerald. The VR Book: Human-centered design for virtual reality. s.l. : Morgan & Claypool, 2015.

2. Cooke, Daniel. When was virtual reality invented? Pebble Studios. 17 08 2017. [Cited: 30 09 2019.] pebblestudios.co.uk/2017/08/17/when-was-virtual-reality-invented/.

3. Pausch, R., Proffitt, D. and Williams, G. Quantifying Immersion in Virtual Reality. s.l. : Proceedings of SIGGRAPH‘97, 13-18, 1997.

4. Steuer, Jonathan. Defining Virtual Reality: Dimensions Determining Telepresence. 1992.

5. Balladares, Alex. Understanding Haptics for VR. Virtual Reality Pop. 03 03 2017. [Cited: 30 09 2019.] virtualrealitypop.com/understanding-haptics-for-vr-2844ed2a1b2f.

6. Caggianese, Giuseppe & Gallo, Luigi & Neroni, Pietro. The Vive Controllers vs. Leap Motion for Interactions in Virtual Environments: A Comparative Evaluation. 2019. 978-3-319-92230-0.

7. Jacob Copenhaver, Filip Krynicki, Dan Medeiros. Ready at Dawn. [Cited: 1 10 2019.] readyatdawn.sharefile.com/share/view/s4565f1ec6e046469.

8. Inverse Kinematics. Wikipedia. [Cited: 24 09 2019.] en.wikipedia.org/wiki/Inverse_kinematics.

9. Parger, M., Mueller, J. H., Schmalstieg, D., & Steinberger. Human upper-body inverse kinematics for increased embodiment in consumer-grade virtual reality, M. s.l. : Proceedings of the 24th ACM Symposium on Virtual Reality Software and Technology. ACM, 2018.

10. Sourabh Purwar, Jagadeesh Kampara. Designing User Experience for Virtual Reality (VR) applications. [Cited: 18 09 2019.] uxplanet.org/designing-user-experience-for-virtual-reality-vr-applications-fc8e4faadd96.

11. André Zenner, Marco Speicher, Sören Klingner, Donald Degraen, Florian Daiber, and Antonio Krüger. Immersive Notification Framework: Adaptive & Plausi-ble Notifications in Virtual Reality. Paper LBW609, 6 pages, New York, NY, USA : In Extended Abstracts of the 2018 CHI Confer-ence on Human Factors in Computing Systems(CHI EA '18), 2018.

12. Coquillart, Sabine, Guido Brunnett, and Greg Welch. Virtual Realities: Dagstuhl Seminar 2008. s.l. : Springer Science & Business Media, 2010.

13. Entity Hierarchy (parenting). Valve Coproration. [Cited: 20 09 2019.] developer.valvesoftware.com/wiki/Entity_Hierarchy_(parenting).

14. Paul Kabbash, William Buxton, and Abigail Sellen. Two-handed input in a compound task. New York, NY, USA : In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '94), Beth Adelson, Susan Dumais, and Judith Olson, 1994.

15. Coroutine. Wikipedia. [Cited: 26 09 2019.] en.wikipedia.org/wiki/Coroutine.

16. Armbrüster, Claudia & Wolter, Marc & Kuhlen, Torsten & Spijkers, W & Fimm, Bruno. Depth Perception in Virtual Reality: Distance Estimations in Peri- and Extrapersonal Space. s.l. : Cyberpsychology & behavior : the impact of the Internet, multimedia and virtual

reality, 2008, Vol. 11.

17. Williams, Glyn. What is raycasting? Quora. [Cited: 25 09 2019.] quora.com/What-is-raycasting.

18. 3D Television. Wikipedia. [Cited: 25 09 2019.] en.wikipedia.org/wiki/3D_television.

19. Thomas, Frank, Ollie Johnston, and Frank Thomas. The Illusion of Life | Principle of Animation | 12 Basic Principle of Animation. Youtube, Entertainment channel, February 2015. [Cited: 10 09 2019.] youtube.com/watch?v=yiGY0qiy8fY.

20. Polishing Your App with Animations and Audio Cues. Microsoft Coproration. [Cited: 20 09 2019.] blogs.windows.com/windowsdeveloper/2016/08/09/polishing-your-app-with-animations-and-audio-cues/.

21. Understanding Motion. Material Design. [Cited: 14 09 2019.] material.io/design/motion/#usage.

22. Robert, Bruner. Repetition is the First Principle of All Learning. 2001.

23. Brefczynski, Julie A., and Edgar A. DeYoe. A physiological correlate of the'spot-light'of visual attention. s.l. : Nature neuroscience 2.4, 1999. 370.

24. Cherry, Kendra. The Color Psychology of Yellow. [Cited: 28 08 2019.] verywellmind.com/the-color-psychology-of-yellow-2795823.

25. Grimshaw, Mark, Craig Lindley, and Lennart Nacke. Sound and immersion in the first-person shooter: mixed measurement of the player's sonic experience. s.l. : Audio Mostly-a conference on interaction with sound. www.audiomostly.com, 2008.

26. Sparkes, J. J. Pattern recognition and a model of the brain. s.l. : International Journal of Man-Machine Studies 1.3, 1969. 263-278.

27. Gwinner, John. How do the eyes focus in VR? Quora. [Cited: 20 09 2019.] quora.com/How-do-the-eyes-focus-in-VR.

28. Rempel, D., Willms, K., Anshel, J., Jaschinski, W., & Sheedy, J. The Effects of Visual Display Distance on Eye Accommodation, Head Posture, and Vision and Neck Symptoms. 5, 2007, Vol. 49.

29. Budiu, Raluca. Quantitative vs. Qualitative Usability Testing.1 10 2017. [Cited: 07 10 2019.] nngroup.com/articles/quant-vs-qual/.

30. Höll, M., Oberweger, M., Arth, C., & Lepetit, V. Efficient physics-based implementation for realistic hand-object interaction in virtual reality. s.l. : IEEE Conference on Virtual Reality and 3D User Interfaces (VR), 2018.

Appendix 1 – Papers Arisen from this Work

1. Steve Kateros, Stylianos Georgiou, Margarita Papaefthymiou, George Papagiannakis, Michalis Tsioumas, **A Comparison of Gamified, Immersive VR Curation Methods for Enhanced Presence and Human-Computer Interaction in Digital Humanities**, 2015
2. Paul Zikas, Vasileios Bachlitzanakis, Margarita Papaefthymiou, Steve Kateros, Stylianos Georgiou, Nikos Lydatakis, George Papagiannakis, **Mixed Reality Serious Games and Gamification for Smart Education**, 2016
3. Margarita Papaefthymiou, Steve Kateros, Stylianos Georgiou, Nikos Lydatakis, Paul Zikas, Vasileios Bachlitzanakis, George Papagiannakis, **Gamified AR/VR Character Rendering and Animation-Enabling Technologies**, 2017
4. George Papagiannakis, Nikos Lydatakis, Steve Kateros, Stylianos Georgiou, Paul Zikas, **Transforming Medical Education and Training with VR Using M.A.G.E.S.**, 2018
5. Efstratios Geronikolakis, Paul Zikas, Steve Kateros, Nikos Lydatakis, Stylianos Georgiou, Mike Kentros, George Papagiannakis, **A True AR Authoring Tool for Interactive Virtual Museums**, 2019

Appendix 2 – ControllerPhysx DLL

As explained in 3.1 section regarding the object interaction, none of the usual implementations can be a viable option. The new method of interacting was mentioned to be the velocity-based interaction. In this thesis this type of interaction was heavily inspired from Newton VR. The object interaction implementation is as important and present in the project as the device manager that has the role of the bridge between the controllers and the implementation. The device manager is linked with Steam VR and thus any type of headset that is supported with Steam VR will be compatible with this project. Both mechanics are stored in the Controller-PhysX.dll which is a separate project.

Starting with the manager, there is a script inside Unity called **SteamVRController** that implements the class (from the dll) **DeviceController**. This script contains the abstract functions of every button a controller can have. Its instance is stored inside the **DeviceControllerClass** script that contains extra useful information for the developer regarding the controllers (a.k.a. the virtual hands). Via this class the developer can access all the functions inside Unity.

To access **DeviceControllerClass:**

using ControllerPhysx.Devices;

DeviceControllerClass.Get.*DesiredFunction()*;

Developer access variables:

- **ControllerTypes**: **enum** - that has all types of compatible headsets
- **controllerType**: **ControllerTypes** – stores the headset that is connected with to the computer.
- **Controllerhand**: **enum** – left of right hand
- **ContollerButtons**: **enum** – VR controller's buttons
- **left/ightController**: **GameObject –** in scene controllers

Developer access functions:

- **GetControllerGrabStrength(Controllerhand _hand)**: **float -** returns normalized value of the grip button of the desired controller (if not analogue – 0 or 1 only)
- **GetHandTag(GameObject child)**: **string -** returns the correct tag of the gameobject given. If the tag is not one of used tags for the hands it returns "UnTagged". For collisions to easily identify if the hand triggered a collision.
- **GetHandTransform(GameObject child)**: **Transform**
- **ControllerhapticPulse(Controllerhand _hand, float _strength)**: **void -** set the vibration strength for the desired hand, normalized value.
- **GetTriggerPressed(Controllerhand _hand)**: **bool -** returns if the trigger button of the desired hand is pressed
- **GetGripPressed(Controllerhand _hand)**: **bool -** returns if the grip button of the desired hand is pressed
- **SetControllerState(Controllerhand _hand, bool _state)**: **void** - set the hand's flashing state. There is a mechanism to flash specific fingers of the hand for tutorial purposes. Calling this function with the state as false will immediately close all flashing renderers. Useful when the developer does not currently know what fingers are flashing.
- **SetButtonFlashing(bool _enabled, Controllerhand _hand, params Controller-Buttons[] buttons)**: **void** – flashes the fingers (by choosing the appropriate button) of

the desired hand. Useful as a tutorial to show the user what button to press in the controller.

Secondly, the other most common script will be the **PhysxInteractableItem**. By attaching this script to any gameobject that has a Rigidbody and colliders, this automatically makes it an interactable - with the user - object that will obey to laws of physics.

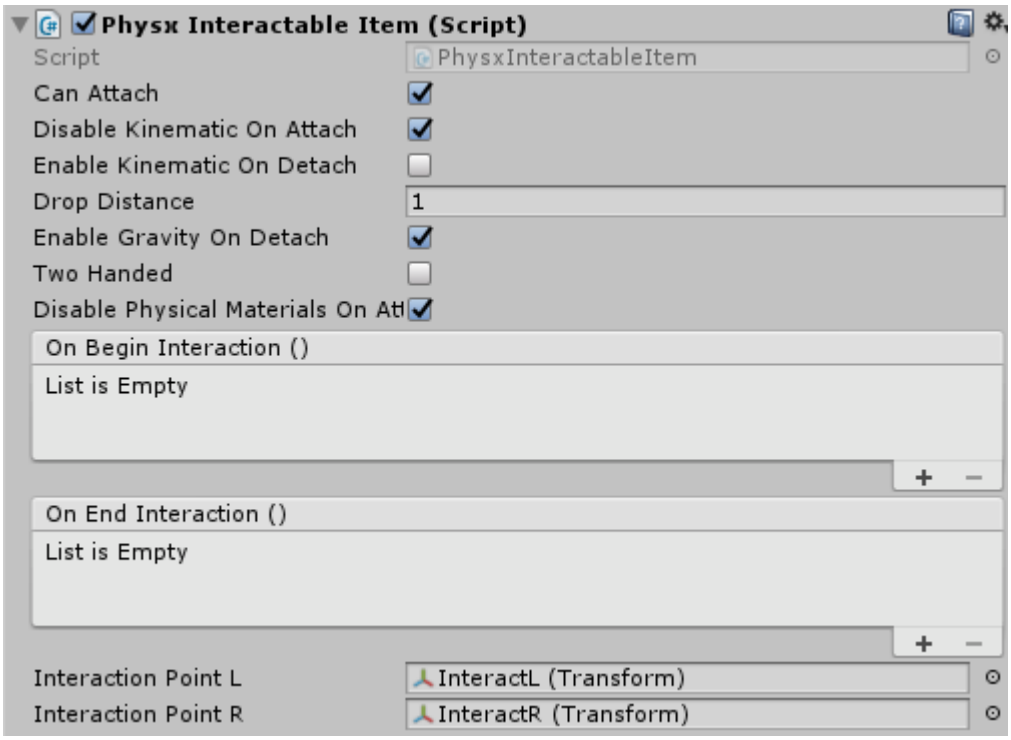


Figure 0.1 Physx Interactable Item script values

- **Can Attach**: enable/disable the object's ability to interact with the user
- **Disable Kinematic On Attach**: when grabbed by the user disable Rigidbody's isKinematic bool
- **Enable Kinematic On Detach**: when object is dropped enable Rigidbody's isKinematic bool
- **Drop Distance**: for Two handed option, select the maximum distance the hands can be apart when holding an object. Above that the interaction comes to an end
- **Enable Gravity On Detach**: when object is dropped enable Rigidbody's useGravity bool
- **Two Handed**: enable the object to be able to be grabbed with both hands from the user
- **Disable Physical Materials On Attach**: if tool has physical materials, disable them when grabbed
- **On Begin Interaction**: Add functions that are going to be called when the users grabs the object. Can also be done via code: GetComponent<PhysxInteractableImet>().OnBeginInteraction.AddListener(*delegate*)
- **On End Interaction**: Add functions that are going to be called when the users releases the object. Can also be done via code: GetComponent<PhysxInteractableImet>().OnEndInteraction.AddListener(*delegate*)

- **Interaction Point L/R**: if the object is desired to be grabbed in a specific pose for each hand, drag and drop the child transform of the object in these two values. If no specific pose is desired leave them empty.

The results of the interaction are based on the pivot of the object. If the pivot is not at the centre of the object it is best advised to attach the script to a parent – empty gameobject and then attach the 3D model underneath it (if a 3D modelling tool is not available). Otherwise depending on the pivots position it might produce weird – unnatural results. In the top two images it is the same interactive object with the pivot centred. Upon hitting on the table, it rotates as it should around it's centre of mass. On the second one it rotates still around the centre of -given- mass producing unnatural results.

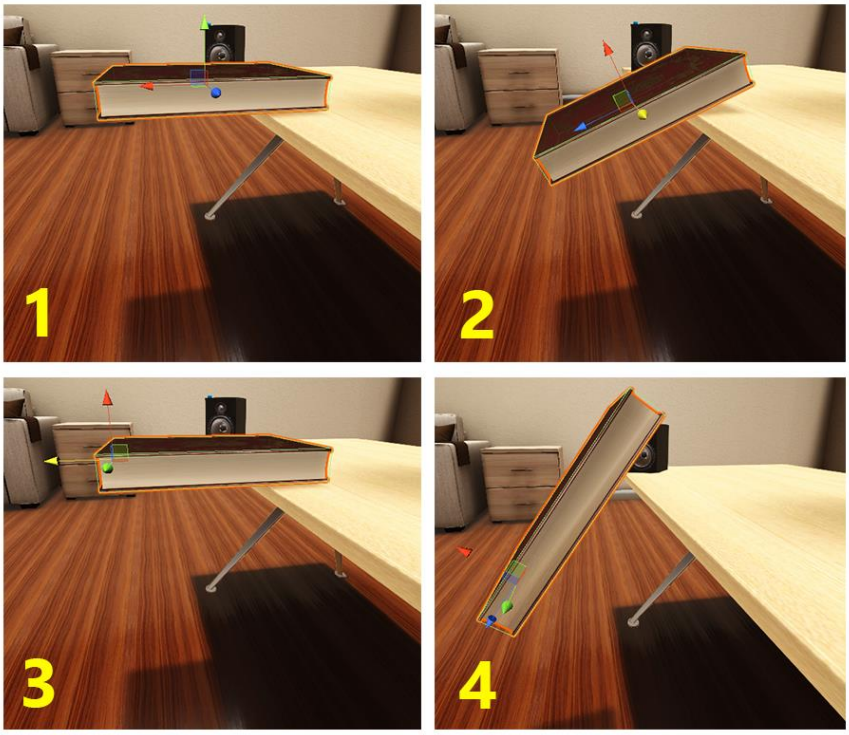


Figure 0.2 1, 2: Centered pivot leads to correct rotation.
3, 4: Non-centered pivot produces unrealistic rotation upon collision

Not all objects should have at the exact centre of the model their pivot for realistic results, but where their centre of mass would be in real life as shown in the large hammer example below.

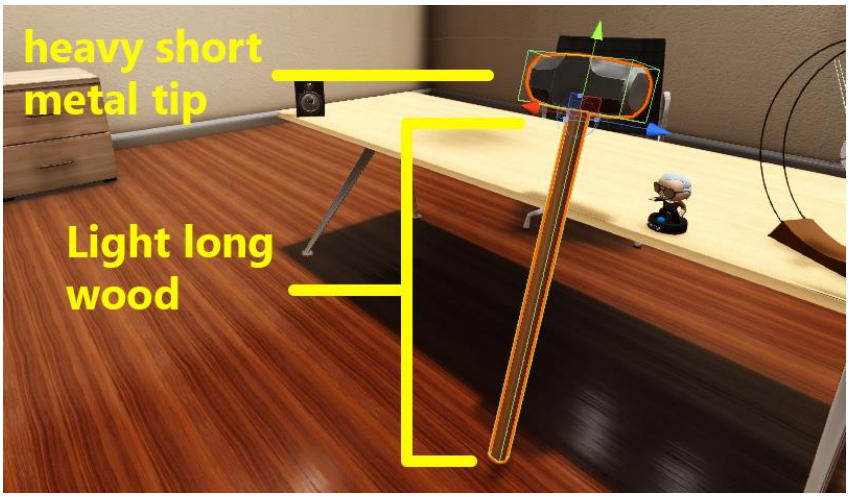


Figure 0.3 Pivot should be at center of mass, example showing hammer and its pivot above

Appendix 3 – Using the Unity project

This Unity project is currently running exclusively on 2017.1.0f3 (it will be updated soon). Only requirement is Steam VR.

Everything mentioned in this thesis is implemented and contained inside a single Unity project. This project contains also many different features, outside of the scope of this thesis, (many used in game development) to assist developers to create a new VR project with all the aforementioned design principles. It is consisted of many managers to assist with the project so the focus can be taken away from development and head towards content. Below everything is explained on how to set up a new project and what managers inside Unity are going to be useful. Some of the implemented pieces of this project that are not crucial to the beginnings of a new project will be left out of this appendix.

1. Event Actions

The storyline of the project is split into small pieces each called **EventActions**. Every action represents a small user's action or movement inside the virtual environment. Every EventAction has it's own script and the role is to spawn all prefabs needed for this event. All EventActions are manipulated from a Handler. When the event is finished the **EventActionHandler** is called and it automatically destroys all current event's prefabs and goes to the next one and spanws the new event's prefabs.

Each event is an empty gameobject inside Unity scene (under EventActions) containing only the corresponding script. The order of the storyline is bound to the order of the events. To Change the order of the storyline, select the event gameobject and drag it above or below the other events.

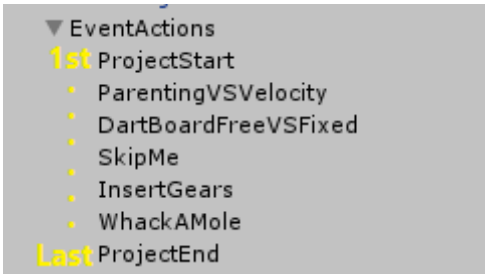


Figure 0.4 Order of Event Actions (as gameobjects) inside Unity scene translates to their execution order when application is running

As mentioned above, the role of every EventAction is to spawn all the prefabs for the corresponding Event. There are two ways the dev can do it. Via drag and drop on the script's gameobject, or inside the script in the override **Init** function call the importer manually.

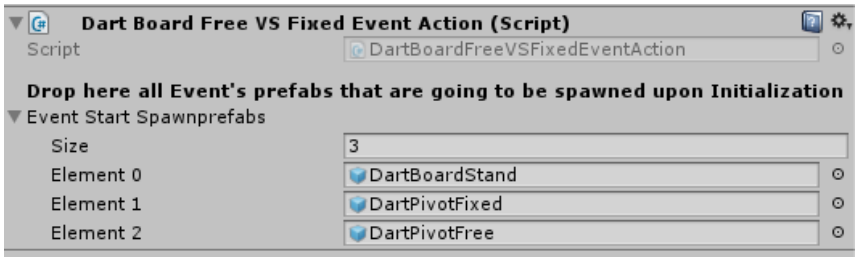


Figure 0.5 Setting objects for instantiation from a specific Event Action inside Unity scene

```
public class ParentingVSVelocityEventAction : EventAction {

    public override void Init() {
        PrefabSpawner.SpawnEventActionPrefab(gameObject.name, "BookVelocity");
        PrefabSpawner.SpawnEventActionPrefab(gameObject.name, "BookFakeParenting");
        PrefabSpawner.SpawnEventActionPrefab(gameObject.name, "BookUnityParenting");
        PrefabSpawner.SpawnEventActionPrefab(gameObject.name, "BooksBin");
        PrefabSpawner.SpawnEventActionPrefab(gameObject.name, "BookDropUI");

        base.Init();
    }
```

Figure 0.6 Setting objects for instantiation from a specific Event Action inside script

Each EventAction is consisted of:

- the gameobject of the event's name inside Unity scene under EventActions.
- a file of the event's name containing all event's scripts under MScFiles/Scripts/EventActions/AllActions.
- a file of the event's name to store all event's prefabs under Resources/Prefabs/EventActionPrefabs.

It is recommended not to alter the paths or the names of the folders otherwise the project will not work properly. Handlers and the importer have access to these folders.

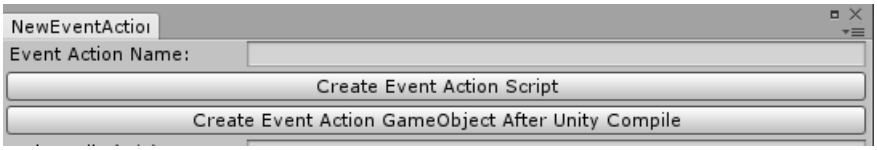


Figure 0.7 Unity editor window responsible for the Event Actions

To create a new EventAction go to Unity's toolbar, KaterosMSc > EventActions. Write the name of the event and press Create Event Action Script. Then Unity is going to build the new script. The script will be named as the name provided from the dev plus the "EventAction" at the end. It will also create the two folders with the name given. Once it's done building press the Create Event Action GameOject After Unity Compile. This will create the gameObject in the scene and attach the script to it.

Each script is a type of EventAction. There are some functions that can be overridden. They are useful for further development in each event.

- **Init**: it is called once from the Handler when the event starts.
- **Finish**: it is called from the Handler when the event is over, before going to the next one.
- **Undo**: it is called from the Handler when the event is over, before going to the previous one.
- **CustomUpdate**: It is called every frame, just like Unity's Update, but from the project's Update Manager. By default, this function is not called. If it is overridden and it is desired to be called, update the Boolean **addCustomUpdate** to true inside the Init function.

Currently this unity project has it's own storyline. To begin a new project, in the same Unity window, press the reset Event Actions button. It will erase all events folders, prefabs, gameobject and scripts.

2. Event Action Handler

For developing reasons each event can be skipped (forwards and backwards in time). Tab goes to next event and LShift + Tab goes to the previous. The change of these events, manually from the developers or from triggers inside the VR, are happening with the help of Event Action Handler and the Event Manager.

During gameplay it is not known what action is currently running. It is the Event Action Handler's role to be the bridge between the events and the rest of the project. It stores in the list all events and it is the one that calls them. It is a singleton and it has 3 functions for the developers to call. These functions are managed via the Event Manager script also for convenience (see Event Manager for details below).

- **EndCurrentEventAction**: Calls the Finish function for the current event.
- **ChangeEventAction(bool _next)**: If Boolean is true it calls the next event's Init function and this event becomes the current one, otherwise if it is false It calls the previous event's Init function and this event becomes the current one.
- **GetCurrentEventActionName**: **string**, returns the name of the current Event Action.

3. Event Manager

This manager is also a singleton and contains a dictionary where a function is bound to a specific string. Its purpose it to connect scripts inside the project. It is mostly used to connect the Event Action Handler with the behaviour-scripts in prefabs. When their purpose is done (for the specific) event, they initiate the next Event Action via the Event Manager.

- **StartListening(string eventName, UnityAction listener)**: binds the listener (containing the function) to the string given. If it already exists, it prints an error to the console.
- **StopListening(string eventName)**: removes from the dictionary the entry with the key given.
- **TriggerEvent(string eventName, float delaySec)**: call the function bound to the key given, add a timer if the function is required to be called with a delay.
- **PrintAllInvokes**: all function invokes are stored in order. Prints that order, for debug puproses.

So, at this point how does the Event manager connects the Event Action handler with the scripts? When a new event is about to start, the handler changes the event and calls the new ones the Init function. The base Init of that script adds its own Finish function to the Event Manager and as key it gives the gameobjects name (remember the gameobject only hosts the event's script and also has as a name the event's name as given from the developer).

```
EventManager.StartListening(gameObject.name, () => { Finish(); });
```

Figure 0.8 How to store a function inside Event Manager

Now if a specific prefab is responsible to finish the current event's turn, it must somehow inform the system that this event is over. The solution is to call the Event Manager with the stored function.

```
EventManager.TriggerEvent(EventActionHandler.GetCurrentEventActionName());
```

Figure 0.9 How to trigger a function from Event Manager

4. Update Manager

Update Manager has a dictionary storing functions bound to a key given. It internally has an **Update** and a **FixedUpdate** function. Every function is called in one of the two functions (specified from the developers) every frame or with a specific timer. Its role is to be a box where all Update functions are encapsulated into one. This helps with debugging as it provides the list of all scripts called in order and other conveniences for the developers such as a limited amount of lifetime.

- **AddCustomUpdate(Action _function, bool _isFixedUpdate, string _gameObject-Name, float _msDelay, float _secLifeTime)**: it adds the function to the Update Manager. Parameters:
  - Action _function: the function that will be called in one of both Updates
  - bool _isFixedUpdate: true for FixedUpdate or false for Update
  - string _gameObjectName: the key the function will be bound to. It is recommended to add the gameobject's name containing this script to avoid preventions of duplicate entries
  - float _msDelay: add a number in milliseconds for the function's call intervals if it's not desired to be called on every frame.
  - float _secLifeTime: add a number in **seconds** if the function is desired to run for a specific time.
- **RemoveCustomUpdate(string _gameObjectName)**: removes the entry from the Update manager.
- **PrintAllUpdatesInOrder**: for **debug** purposes it prints the order of the gameobjects that their functions are called.

5. Prefab Spawner

For convenience purposes there is the prefab Spawner singleton in this project. It searches and spawns (or Instantiates) prefabs inside the Unity scene. It is used purely for all EventAction scripts. This singleton retrieves the paths provided from the developers from the Path Manager (see below for details) and searches for the prefabs given name inside the folder of the events name. If it's found it spawns the prefab and stores internally it's instance. Thus, in every current event, it has stores all its prefabs. When the events Finish or Undo functions are called, all these prefabs are deleted via the Prefab Spawner.

The only time it is recommended to be used from the developers is inside Init functions of every event to spawn their desired prefabs.

- **SpawnEventActionPrefab(string _eventActionName, string _prefab, Transform _parent)**: returns **gameobject**. Searches for the file of the event name given and spawns the prefab given by it's name. If the prefab must be a child of another object, it's transform is required, otherwise null. Just like Unity's Instantiate.
- **SpawnEventActionPrefab(string _eventActionName, GameObject _eventAction-prefab, Transform _parent)**: Same function but spawns the prefab that is given as a gameobject. Mostly used internally for the drag and drop prefabs.
- **DestroySpawnedhologram(string hologram)**: used internally mostly, deletes the hologram given by the name. if null string is given it destroys all spawned holograms.
- **GetCurrentSpawnedEventActionPrefab(string _prefab)** returns **gameobject**: for speed and dev convenience, if an instance of a prefab is not stored anywhere and the dev wants to access it the way to find it is via Unity's GameObject.Find(). This can be depending on scene a costly call. If the prefab is spawned via this Spawner then this function does the same job as Unity's aforementioned function but for a much smaller quantity of prefabs to search (as many as the current event has spawned).
- **GetCurrentSpawnedHologram(string _hologram)**: the same as above but for holograms
- **Reset**: destroys everything that is currently stored

6. Path Manager

For organizational purposes most of the files are contained under specific paths. For example. all event prefabs are stored under Resources in EventActionPrefabs folder. Of course, any of these paths are not "hardwired" into the project. They are stored inside an XML file under Assets (**FilePathConfig**) and are also bound to a specific key-string. The **Path Manager** is also another

dictionary. It has a string value as a parameter which is the key that is bound to a specific path and returns as a string the path itself.

- **GetPath(string _variableName)**: Returns the full path depending on the key given.

7. UIs & UI Management

The User Interface is different segment in this project that works on its own. It is a 2D interface, just like in any other game, that everything is controller from the singleton **UI Management**. To spawn or delete and one of UIs the only access is within this script. Below there is a detailed explanation of the UI set up and how this Manager works.

7.1. UIs

7.1.1. Application Generic

Many of the UIs have the role of templates. For any new project the developers will need to update them. This is true for anything except this type of UIs, the application generic ones. They are split into two different categories, User UI – the interactive ones and the Notification UI – the non-interactive ones.

User UIs are all the options available for the user that are independent of the project's storyline. These are the Start & Exit UI, and the Options UI that allows the user to skip events.

Notification UI is a simple UI canvas that takes as an argument the message that is going to display. Depending on its importance it is split down to three different notifications:

- Notification: Blue canvas border, simple spawn tone, additional information.
- Warning: Yellow canvas border, simple spawn tone, additional warning information.

  Error: Red canvas border, different spawn tone, for user errors.

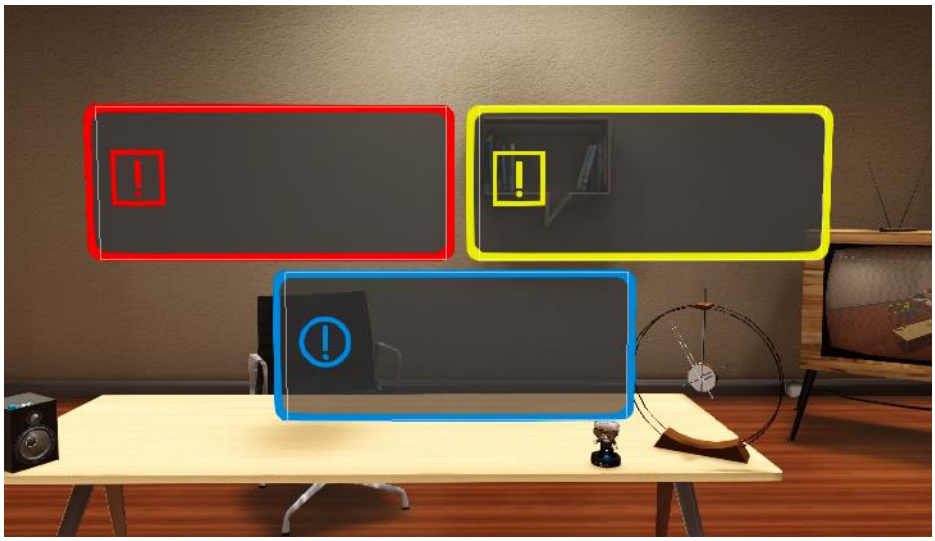


Figure 0.10 All notification UIs, top left: Error, top right: Warning, bottom: Notification

7.1.2. Action Progress

They are spawned via UI Management (more on that below) and they display a string given that will assist the user with the completion of the event. It is suggested to use numbers and percentage as the final progress UI that is called it is prefixed and it contains the message "100%" alongside a correction-tick.

7.1.3. Special Cases

Free for the developers to create any type of UI they want that the UI Management will be able to spawn and destroy.

7.1.4. Aidline

These are the UIs responsible of assisting the user to observe objects being placed out of sight that he/she will be required to use. There is one fully created Aidline UI inside the folder that is used for an action event. The same can be used as a template for the creation of others.

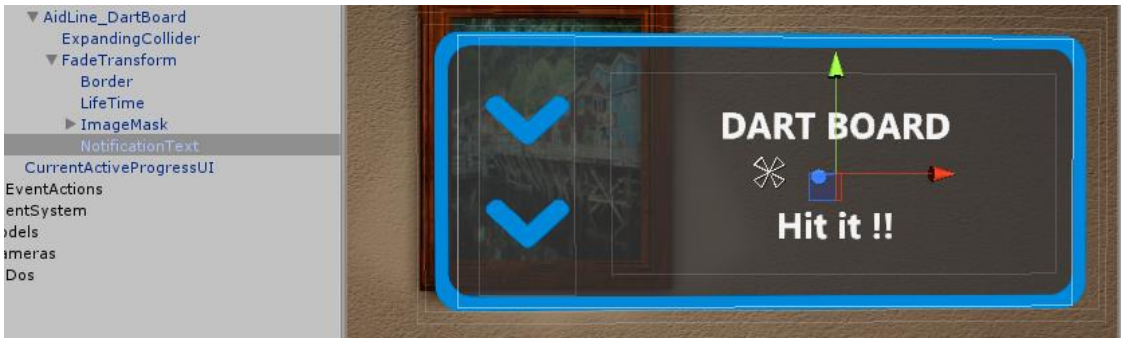


Figure 0.11 Aidline child gameobject containing Text Component

To change the displayed message, go to **NotificationText** child gameobject of the UI and update it. Each new Aidline UI must have a new name.

An Aidline's job as mentioned is to assist the user with object observation around the area. Naturally their role is specific to an event action. Each spawned Aidline is automatically destroyed when the event action, in which there are spawned, is over (they do not have to be spawned from inside the event action script, but anywhere and they are automatically bound to the current event action). There is a mechanic to destroy the Aidline before the event ends (upon interacting with the gameobject it is supposed to show to the user), and a mechanic to spawn a new Aidline after the destruction of the existing one (and this can go on as a chain). On the parent gameobject there is a script **AidlineArrowManagement**. It has an Event List (UnityEvent) where these functions can be selected. Add a new one by clicking '+' drag inside the parent gameobject of the Aidline and select the **UIAidlineFunctions** script. It consists useful trigger functions for the Aidlines, whereas the most important two are:

- **SetGameObjectInteraction** – parameter: **string**: This Aidline will be destroyed when the object given is triggered. Triggered means that the object is interactable (contains script PhysxInteractableItem) and triggers this function upon grabbing the object. Takes as a parameter the name of the object as a string.
- **SetNextAidline** – parameter: **string**: Upon destruction of this Aidline it spawns the next one (internally it calls UI Management). Takes as s parameter the name of the Aidline as a string.

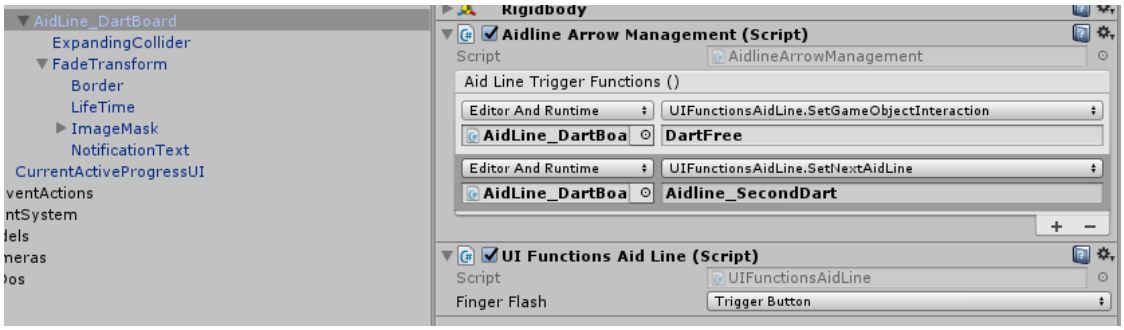


Figure 0.12 Aidline script where dev can select functionality upon UI's triggering

The Aidlines when spawned from the UI Management, also contain a separate element in the (that can be toggled off from the spawn-function called) the arrows. These arrows are helpful to orient the user to the position the Aidline is. After all this is the purpose of Aidlines. To show to the user out of reach objects. The Arrows are spawned in front of the user* and their direction is the position of the Aidline.

**in front of the user: for many UIs their spawned position is in front of the VR camera, with an offset, with their direction being towards the camera. This position and rotation of the UI elements in copied from a gameobject's transform: Cameras/VR Camera/[CameraRig]/Camera (head)/Camera (eye)/UserUISpawnPoint.*

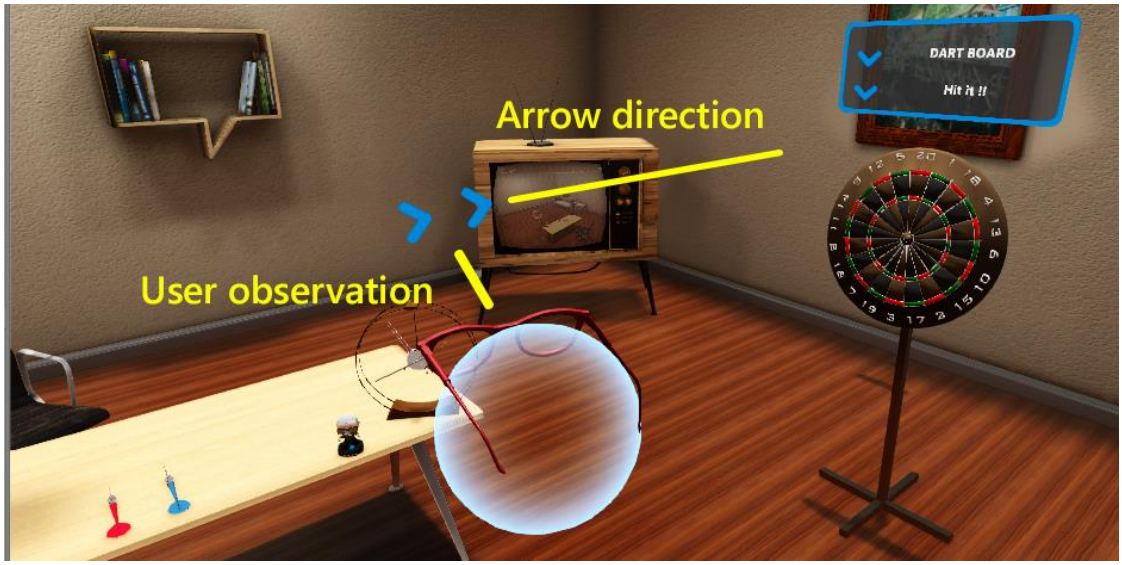


Figure 0.13 How Aidline's arrows work (spatial orientation assistance)

7.2. UI Management

This script contains all functions and variables that are accessible and useful to the developer to use all types of UIs. Also, inside this script contains the behavioural mechanism of the UIs (exists also as header comment inside the script).

UI Management is responsible for both the generic type of UIs and the developer / project specific. Since it is not a good design choice to bombard the user with UIs the generic ones at least that are the most commonly used obey to some rules. The generic ones are the User UIs that are interactive with the user and the Notification UIs. These two, apart from an exception, cannot co-exist inside the user view.

Notifications have the top priority. If a User UI is already instantiated and a new Notification is called to appear, the first will be closed automatically for the notification to be displayed. After the notification is done and destroyed, the User UI will come back (respawn).

In a different scenario, if a notification is about to be spawned, and another one is already active, it goes on a queue and waits for its turn till the other notifications are done. Same goes for User UI. If one is about to be instantiated but another one already active, it will be stored on a queue to be activated once the previous one is over. The way this system works is to minimize the UI in front of the user and to avoid two different types of UIs being hit with each other (since most of them spawn in front of the user).

Aidlines and Progress UIs are action specific and do not have to obey to the rules above. They can only be spawned one at a time. Lastly Special Cases contain all different UIs that are not generic and have a specific, developer created, purpose. There are no rules for this type, not even how many can be instantiated at the same time.

Accessible variables:

- **enum UISounds**: All sounds the developers can access for the UIs the order of the sounds that will be dropped inside UI Management must match the enum's order.

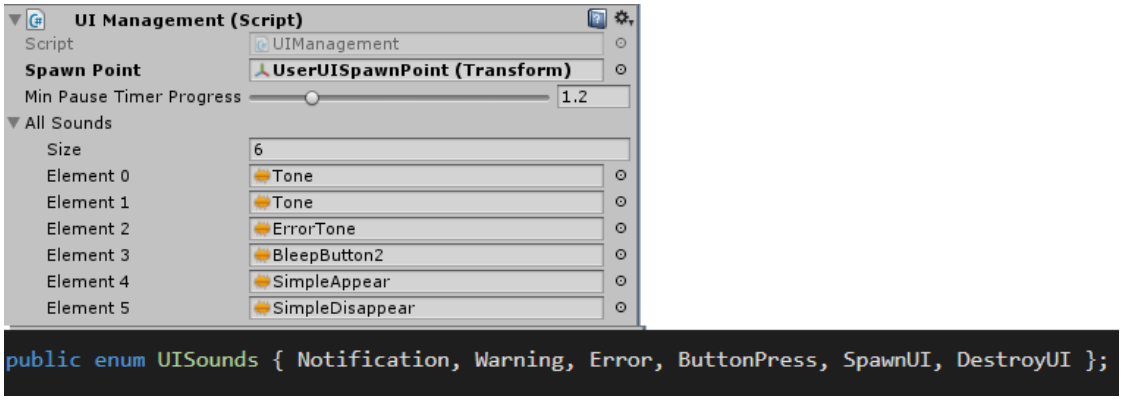


Figure 0.14 Sound order must match the order of enum

- **enum NotificationUITypes**: All types of notification the developer can select to spawn, Notification, Warning, Error.
- **enum UserUITypes**: All types of generic and interactive UIs the developer can select to spawn.
- **enum SpecialCaseUITypes**: The same thing as above but for the developer created custom UIs. When a new UI is created, its name must be updated also here.
- **enum ProgressUITypes**: The UIs responsible for notifying the user of the percentage completion of the event action. Progress UI if it's not finished and ProgressUI100 for when the event is finished.
- **Transform spawnPoint**: Most UIs are spawned in front of the user with an added distance. This is achieved by attaching to the camera an empty gameobject at the desired distance. Then all spawned UIs that are required to be in front of user's face, copy the Transform values of this gameobject. The one used is located at: Cameras/VR Camera/[CameraRig]/Camera (head)/Camera (eye)/UserUISpawnPoint
- **float minPauseTimerProgressUI**: set a timer between each spawn of the Progress UI.
- **List AudioClip allSounds**: the list with all UI sounds. The order must match the enum as mentioned above.

Accessible functions:

- **PlaySound(UISounds _sound)**: Plays the sound for the UI. Only UI Management has an AudioSource.
- **SpawnUserUI(UserUITypes _type, bool _spawnAsDynamic = true)**: Spawns the UserUI given by the name. By default, it will spawn in front of the User. If this is not desired set _spawnAsDynamic to false.
- **SpawnNotificationUI(NotificationUITypes _type, string _displayMessage, float _lifeTime, string _uniqueID = "", bool _spawnAsDynamic = true)**: Spawns the desired notification UI. Select one of the three types of UI, the message that is going to be displayed and how long it will last (the lifetime of the notification is observable from the border color. It decreases linearly as the UIs lifetime is coming to an end). Because the same Notification prefabs are used for any message, it can be difficult to distinguish with notification is spawned (as their only difference is the displayed message). To separate each notification that gets instantiated the dev can add to it a unique tag as a string to the parameter **_uniqueID**.

- **SpawnSpecialCases(SpecialCaseUITypes _type, Transform _parent = null, bool _spawnAsDynamic = true)**: Spawns the developer specific UI given. These have the option to be attached to another gameobject. The second parameter is the Transform of the object this UI might be desired to be attached to.**:**
- **SpawnAidline(string _aidlinename, bool _spawnArrow = true)**: Spawns the Aidline given by name. Upon spawning the Aidline it also spawns a set of arrows showing where the Aidline is. The Aidline is spawned where the prefab was created and the arrows are spawned in front of the user. This is useful if the Aidline is out of user's sight. If the arrows are not desired, they will not be spawned by setting the parameter _spawnArrow to false.
- **SpawnProgressUI(string _message, bool _isEventActionCompleted = false)**: Spawns the spall UI progress with the string given as the first parameter. If the action is completed and the 100% progress UI is required to pe instantiated, set the isEventActionCompleted boolean to true. Then the message is ignored and spawns the ProgressUI100.
- **GetCurrentActiveUserUI()**: returns the gameobject instance of the User UI, else null.
- **GetCurrentNotificationUI()**: returns the gameobject instance of the Notification UI, else null.
- **GetIsDynamicCurrentNotificationUI()**: returns Boolean of the Notification is Dynamic. As mentioned above of how the UI Management works ONLY the dynamic notifications destroy and respawn the User UIs after their lifetime is over. This might be needed from the developers to know if the notification is static or not.
- **GetAllCurrentSpecialCaseUI()**: returns a list of all gameobjects that represent the special case UIs. As mentioned above of how the UI Management works, special cases obey to no rules.
- **GetCurrentAidlineUI()**: returns the gameobject instance of the Aidline, else null.
- **GetCurrentUIName()**: returns string either the notification uniqueID stored, and if not found it returns the name of the User UI gameobject name. Else null. (more explanation of why this is useful inside code).
- **DestroyCurrentUserUI(bool _destroyImmediate = false)**: destroys the currently spawned UserUI. When UIs are called to be destroyed, they play first a fade our animation that can last up to a second. If they are desired to be destroyed immediately, set the _destroyImmediate Boolean to true.
- **DestroyCurrentNotificationUI(bool _destroyImmediate = false)**: destroys the currently spawned NotificationUI. When UIs are called to be destroyed, they play first a fade our animation that can last up to a second. If they are desired to be destroyed immediately, set the _destroyImmediate Boolean to true.
- **DestroyAllCurrentSpecialCaseUI()**: destroys all currently spawned special case UIs.
- **DestoryCurrentAidline()**: destroys currently spawned Aidline.
- **ResetUIManagement()**: destroys everything and clears all values.
- **ClearWaitingQueues()**: clears internal priority-queues for the upcoming instantiation of User UIs and Notification UIs.

8. Rest of developer-useful scripts

8.1. CameraRigInputController

Attached to the camera. Requires Unity's Character Controller. Responsible for camera movement and controller buttons. Left controller button press enables movement (left

joystick/trackpad translation, right joystick/trackpad height), and right controller opens/closes the Options menu (for Vive, Mixed Reality headsets the menu button, for Rift press joystick inside).

8.2. ControllerButtonTutorial

It is the script responsible of storing each fingers material and upon call to assign new materials for flashing. Useful to notify the user which button to press by flashing the finger that is resting on the desired button. It can be complicated to use this script, thus for developer convenience use the provided functions from **Device Controller** (inside Unity it is the **SteamVR Controller** script) See Appendix 1 for more info on these functions (SetControllerState and SetButtonFlashing).

8.3. FakeParenting

Works like Unity's Parenting but without the object being a child to the other object that is following. Attach it to the desired "child" object and drop the desired "parent" gameobject in the **parent Transform** variable. Then this object will follow "the parent" just like as Unity's parenting works but inside Unity the gameobject will have a null parent. Useful when Unity's parenting functionality in terms of transform is wished without many of the problems that can surface (e.g. the pain of incorrect scales).

8.4. PrefabPlacementAutoAdjust

If an object is required to be placed in a specific state, this script has the role of an assistant for the user. An object cannot be perfectly placed inside the VR due to lack of materialistic resistance and other reasons. This script helps by setting the correct angle and place an object needs to be inserted. It contains an acceptance offset of the required placement (e.g. placing object up to 20 degrees off from all three axes combined) and when the object is partially placed, it "takes control" of the object and lerps it into its final position. This is useful because many objects, especially small or complicated ones, will never be correctly inserted from the user. This script allows seamless interaction with these objects and in the meanwhile, it corrects them in ways that may not be observed from the user.

To use this correctly it requires some steps from the developer. The desired interactable object must be duplicated (for the script to compare same Transforms), removed from all behavioural scripts and have this script only attached to it. This object will be now the final position of its interactable brother-object. The developer must take this object and place it in its final position. Then the dev must fiddle with the parameters mentioned below to his/hers desired specifications:

- **Perform Event: bool** select if upon insertion the Event Action will be over.

- **Max Angle Degree Diff: float** select the acceptance insertion offset of the desired angle.

- **Lerp Seconds Play** / **Translate Speed Mul** / **Rotate Speed Mul**: float select the lepr -insert animation values of the object (this animation is the auto-adjustment from the script when the user has the object within the permitted offset but has not precisely to the desired angle).

- **Interactable Prefabs**: select what prefabs this object will observe for insertion. Requires name of prefabs as string.

- **Holograms**: write the name of the hologram(s) that this script will destroy upon user's object insertion.

- **After Placement Trigger**: Add extra functions that this script is going to trigger upon user's object insertion.

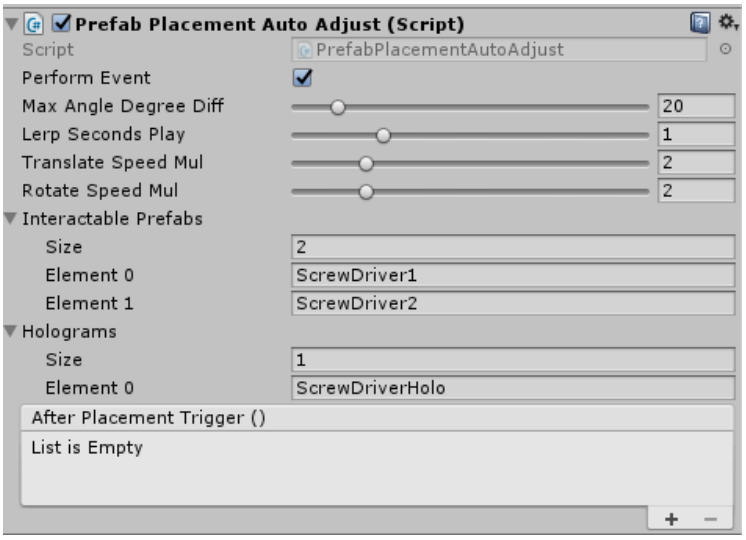


Figure 0.15 Setting up parameters of Prefab Placement script

The developer's work here is done. When this object gets spawned, its renderers will be automatically disabled. When the interactable brother-object is within range and angle offset, this script will detach the object from user's hand and auto adjust it into the correct position. Once it is precisely on the correct position and rotation, the interactable object is destroyed, this object's renderers are enabled, and the script is self-destroyed. If **Perform Event** Boolean is set to true, the Event Action is also completed.

8.5. PrefabSpawnNotifier

This script is used to make the user notice the object that must be interacted. It is simple in use. It must be attached to the object and the object will automatically flash. When interacted (grabbed) with the user the renderers will stop flashing and the script will self-destruct. There are some options available for the user:

- **Notifier Type**: **enum**, select how the object is going to be notified. Select Material Flash and the whole object will glow the color selected from the next parameter. Select New Material Outline and the object will have an outline glow of the color selected from the next parameter.

- **Highlight Color:** select the color the object is going to flash. Caution! When the user's hand is hovering the object the selected color will change and will start flashing green. This helps the user understand when the hand is within reach of the object. It is advised to select any color except green for the highlight.

- **Outline Material**: ONLY if the outline material is selected as a flashing method, it requires an additional shader. Drop here the material with the shader attached. This project provides an outline shader, material name: OutlineShader.

- **Start Notification Timer**: select after how many seconds the object will start flashing.

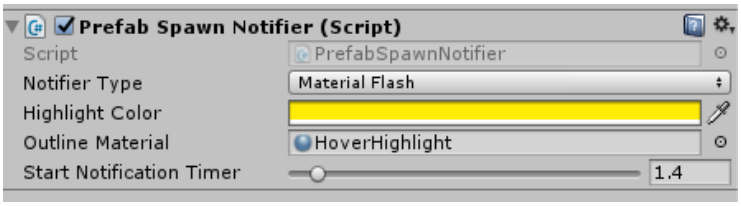


Figure 0.16 Setting up parameters of Prefab Spawn Notifier script

8.6. ObjectFeelInteraction

A simple script to prove that variable strengths of vibration alongside with sound (for object impacts) is an important design to further immerse the user. Currently it works

with a manual adjustment of the maximum velocity the object at hand can reach (while held by user). Developer has to set the min and max values of the object velocity that will play the role for the vibration and sound strength (normalized). Attach the script to the interactable object. It works only with non-triggered colliders.

8.7. TransformInteractable
It is the FakeParenting script but retouched to work for user's hands. Attach this script to the object that needs to follow the hand when grabbed with the "fake parenting" algorithm. This script is created for evaluation purposes. It is recommended to use Physx Interactable Item instead.

8.8. UIMissingFontEditor
Unity Editor script. During the development of this project I noticed that Unity kept missing the added fonts I had. This script searches all prefabs within a path given that have a Text Component and places the font selected (font is given by name as string parameter).

9. Rest of developer-useful assets:

9.1. VR Camera
VRCamera works with any type of commercially available headset that is supported from SteamVR. Also, inside the controllers are two virtual hands. The contain resting, grab and pitch custom animations. Each finger has a separate material for flashing purposes (assist user what button to press at given time).

9.2. VR Guardian
It is the in-house borders of play area. The rug represents the play area and when user is of limits two spotlights appears for each virtual hand. At the moment the rug scale is fixed (to the play area size created for the evaluation), but in future development rug's scale will be matched to size of user created play area.

9.3. IK Mode Manager:
One of the future work list items is the full body IK. It is a concept that is already started in this project. This script is attached to **Scene Management** (inside Unity Scene). There is one option for the developers:

- **Full IK Mode: Boolean**, if disabled the VR hands (with button pressing animations) will be available upon use. If true these hands are disabled, and the full IK dummy will be rendered. For the current version it is advised to keep this option disabled.