A Method For Active Motion Sickness Reduction Using Predictive Models

---

A Thesis

Presented to the

Faculty of

Wentworth Institute of Technology

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Applied Computer Science

---

by

Jacob L. Ledbetter

Spring 2024

WENTWORTH INSTITUTE OF TECHNOLOGY

The Undersigned Committee Approves the

Thesis of Jacob L. Ledbetter:

A Method For Active Motion Sickness Reduction Using Predictive Models

Yetunde Folajimi, Chair
School of Computing and Data Science

Micah Schuster
School of Computing and Data Science

Lauren Melfi
School of Computing and Data Science

DEDICATION

Dedicated to my grandmother, may she rest in peace.

ABSTRACT OF THE THESIS

A Method For Active Motion Sickness Reduction Using Predictive Models
by
Jacob L. Ledbetter
Master of Science in Applied Computer Science
San Diego State University, 2024

Motion Sickness is a common problem in Virtual Reality (VR). It is often described as discontinuity between seen motion and motion felt by the vestibular system, though multiple other factors play a role. While much research has been done to understand why motion sickness occurs and when it is happening in the user, little has been done to actively prevent motion sickness as or before it occurs. We plan to use a machine learning algorithm to detect when a user is likely experiencing motion sickness. Once motion sickness has been detected, corresponding mitigation features such as vignetting, snap turning, or user warnings are enabled on the user's behalf. We found that this method of active motion sickness mitigation is feasible. We also found that consideration must be taken into the performance impact that this active mitigation system may have on the VR system.

TABLE OF CONTENTS

PAGE

LIST OF TABLES

PAGE

LIST OF FIGURES

PAGE

ACKNOWLEDGMENTS

I would like to thank Dr. Folajimi for helping find articles I could not during my lit review.

I would like to thank Dr. Schuster for guidance, assistance with LaTeX, and being a shoulder to cry on when everything goes wrong.

I would like to thank Dr. Melfi for guidance with some complicated matrix transformations.

Lastly, I would like to thank members of the ChilloutVR Modding Group Discord server for helping with C# issues and pardoning some truly terrible code.

CHAPTER 1
INTRODUCTION

Virtual Reality (VR) is a popular medium for user interaction. Its primary feature is the use of a head mounted display(HMD) that allows the user to see a fully rendered virtual environment. Such a display necessarily blocks the user's vision of the real world. In addition to the HMD, the user typically has controllers allowing them to move through and interact with the environment. The HMD and controllers have their position and rotation tracked in physical space. This tracked data used to create a tracked space containing the tracked devices, which can be used to attach and move objects within an application's virtual space. VR is used in several contexts from telepresence[1], to simulation and training, to entertainment[2].

1.1 Problem Statement

A common problem in VR is that of motion sickness. Motion Sickness includes symptoms like nausea, fatigue, and upset stomach, among others. This most often occurs in VR when the user is moved in the virtual environment while the user is stationary in reality. Examples include moving using the controllers, standing on moving platforms, or riding vehicles.

Many VR applications are aware of these issues and have implemented some features to prevent motion sickness. One common option is to apply a vignette[3] to the user's vision when they are moving. Another is to change how the user moves, either by letting the user select an area or moving an avatar in third person to an area, and then teleporting the user to said specified area[4]. The last common option is to snap the user's rotation a set increment rather than smoothly turn them[5].

No solutions currently exist to dynamically reduce motion sickness for VR users based on the user's level of motion sickness.

1.2 Literature Review

The majority of solutions to reduce motion sickness are either user toggleable features, like those above1.1, or attempts to reduce latency within the VR system[6] (where latency can cause unexpected environment changes and thus sickness). User toggleable features either require the user to toggle them (before or after becoming motion sick), or are on by default, possibly degrading the user's experience where unnecessary. Decreased latency helps in many cases but is rarely the only cause of

sickness. Many studies have been done to ascertain why people become motion sick in VR[7, 8]. Some have found that 360°treadmills can provide relief[9], but they are not economical for most users, and do not solve all potential motion sickness triggers. Some have also found that visual cues can reduce motion sickness[10], but if unsubtle, may degrade the user experience, especially for those who do not need such cues.

A variety of models have been made to predict motion sickness in the user. Some use Deep Neural Networks (DNNs)[11], some use Convolutional Neural Networks (CNNs)[12], while others use more classical Machine Learning (ML) approaches[13]. Most models predict using a combination of data from the application as well as user submitted feedback, but others predict using the users themselves, specifically their eye movements[14, 15]. Some tools have even been made to help in the creation of predictive models[16]. However, none of these models have been used to drive the reduction of motion sickness, they solely predict when/if the user is getting motion sick.

1.3 Objectives

The goal is to create a model that can predict a user becoming motion sick and find the most applicable motion sickness reduction feature. It should then activate the motion sickness reduction, and subsequently deactivate it, once the user is no longer at risk of motion sickness. As stated in section 1.2, there are many models present for predicting motion sickness, thus many options to consider for what kind of model to use, though a CNN seems best.

CHAPTER 2

METHODS

This is the methods, IE how I did it(badly).

2.1 Motion Sickness Detection with Convolutional Neural Networks

We believe a CNN would be most useful as it can take data from the viewpoint of the user as well as other data (I.E. controller, motion, and pose data) from the user. A CNN take in a wide array of information at once, a CNN can categorize into multiple motion sickness categories, causes, or ratings beyond a binary "is motion sick" or "is not motion sick" rating. More importantly, CNNs can convolve over a window of time to provide a more accurate assessment of the user's sickness as it changes over time.

2.1.1 Dataset

To train a CNN, we need a dataset of VR use with some sort of scoring of how motion sick the user was. For this, we used the VRNet Motion Sickness dataset[17]. The dataset contains a set of recordings for multiple VR games. Each recording has a set of frames (images from the HMD screens, labeled as `s<frame number>.jpg`), their motion vectors (movements of blobs of pixels, labeled `<frame number>.zfp` in the zfp format) and depth (distance from the virtual camera to a virtual object, per pixel, labeled `d<frame number>.jpg`). Each recording also has csv files with data for the camera, controller inputs, controller pose, in game objects, and lights for each recording. As well, each recording has a "voice" csv file containing the motion sickness ratings given by participants during the recording.

The dataset comes as a large collection of nested zip files, some are unorganized recording folders while others contain folders for each game, the game folders containing multiple csv files for each recording and maybe some recording folders. This meant that once all the zip files were decompressed, files needed to be sorted into their respective recording folder and each recording folder into a respective game folder. A fair amount of time was also spent making sure file names matched a specific uniform pattern (`<game name>/P<participant number>VRLOG-<recording timestamp>/<file name>`).

We did not use the lighting data as this tends to remain static in most scenes. Depth data was left unused to reduce complexity. The camera data is the projection and view transformation matrices of the cameras in the scene. These 4x4 matrices

represent how to transform a vertex position from within the VR environment to the 2d image shown to the user. The controller data shows what values are received from the joysticks on the controllers. The pose data shows the real world position, rotation, velocity, and angular velocity of the HMD and controllers. The motion vector data could not be decoded from their compressed zfp format as file headers were either corrupt or not included, and the compression parameters could not be found.

The dataset images are inconsistent from game to game. Some games have both eyes' perspective, while others only have one eye's perspective. While all games have their depth images flipped upside-down, some additionally have their frame images flipped. THe images are quite large for a CNN with 2016 by 1042 pixels and three color channels. Using Pillow(a python library), we flipped images and resized them to half-resolution(1008 by 521 pixels). We dedicated to copy images with only one eye's perspective to make them appear to have two eyes and fit the full resolution of other images.

The motion sickness ratings for each recording were given as the participants in the VRNet dataset recorded them and were not available for each frame of the recording. To account for this, each rating was attached to the nearest recorded frame by it timestamp. For each recorded frame lacking a rating, a rating was generated by linearly interpolating between the two ratings surrounding it and rounding to the nearest whole number. For recorded frames before the first user rating, they were set to the minimum sickness (1) as the participants in the VRNet dataset started recordings not sick. For recorded frames after the last user rating, the last user rating was used as the value. This interpolation step was preformed for each recording and written to a 'voice_preproc.csv' file per recording.

We removed many columns from all the csv format files as many were duplicates, only zero, or would have little impact on the model. Some columns contained multiple values per row and were split to a column per value. Data recorded for each device(HMD, left-hand controller, right-hand controller) were recorded as separate rows per frame. To have one row of data equal to one produced frame of application runtime for each file, all device values for a single frame were combined into one row. Camera transformation matrices were listed for all cameras in the scene, so rows not related to the main camera used by the user were removed. The frame images are referred to as "image" data throughout the rest of this paper, while all other data (controller input, controller pose, camera transformation matrices) are referred to as "numeric" data.

2.1.2 Network Architecture

Two models were created, one to handle image data and one to handle numeric data. Both models were made using the TensorFlow and Keras Python libraries for machine learning.

The numeric model is a Convolutional Neural Network that takes 100 instances of numeric data. Each instance consists of 116 values in IEEE 32-bit floating point format. Thirty-two values are dedicated to the main camera's transformation matrices. Thirty values are dedicated to the 5-axis (with two dimensions per axis) input data from the HMD, left-hand controller, and right-hand controller. Lastly, Fifty-Four values are dedicated to HMD, left-hand controller, and right-hand controller's position, rotation, velocity, and angular velocity within tracking space. This input is passed through a convolution over the window of time using Keras's Conv1D layer. The convolution passes 64 filters over the 100 time steps where each filter is three time steps wide. Each Con1D layer is followed by a BatchNormalization layer that enforces that the outputs of the Conv1D layer have a mean of 0 and a standard deviation of 1. This increases network performance as most Neural Networks(NN) expect to work with normally distributed values. Following batch normalization is a layer representing the neuron activation function, of which the Rectified Linear Unit (ReLU) was chosen. ReLU is quite common for NN and was chosen as such. The convolution, batch normalization, and ReLU activation layers are repeated twice for a total of three convolutions. To convert the filters from convolution to a classification, the output values from each filter need to be pooled together. This pooling is provided by a GlobalAveragePooling1D layer. Lastly, there are five densely connected neurons that will output the likelihood the user is a given level of motion sick from one to five, and as such, use the "softmax" function for activation instead of ReLU.

The image model is very similar to the numeric model. Its convolutions occur on two-dimensional images over the time window and as such, use a Conv3D layer. The convolutional layers use 64 filters with each filter spanning three time steps and three pixels on the x-axis and y-axis of the image. The global average pooling layer also becomes a GlobalAveragePooling3D.

Both models, when trained separately, may not perform as well (and did not as seen later in 3). To aid this, both models are combined into an ensemble model. This was done by connecting their output into a Concatenate layer and a few densely connected layers. The Concatenate layer does as it says and puts the output of two layers into a flat list of inputs (being 10 as each model predicted a ranking one through five). The densely connected layers had 32, 16, and 8 layers respectively and were added to let the combined model learn the shortcomings of the separate models and

adjust its prediction to something more accurate. The combined model ends with a final densely connected layer like the final layer of the numeric and image models.

Below is **??**, an image of the model and its various inputs, layers, and output.

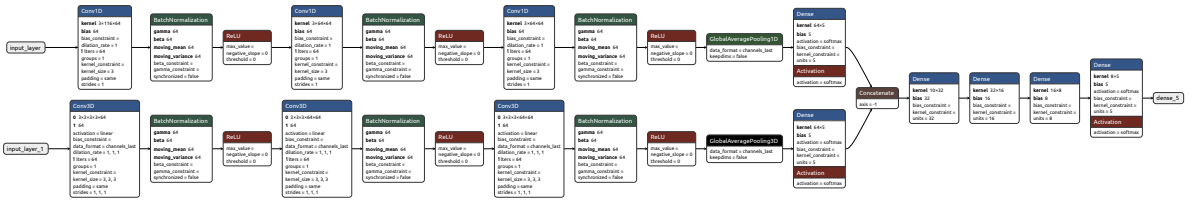

Figure 2.1. Diagram of layers and hyperparameters of the combined motion sickness detection model.

2.1.3 Training

The model was compiled using the Adam algorithm for gradient decent and uses categorical cross-entropy as its loss function. During training, the model was set up to reduce its learning rate (LR) by half when the loss plateaued for 20 epochs to a minimum LR of 0.0001. It was also set up to end training early if 50 epochs passed with no improvement and would log the loss, accuracy, weights, among other things for each training run. Lastly, it would save the weights that had created the lowest loss to a file at the end of each epoch.

First, the image and numeric models were trained for 100 epochs maximum to get a clue as to how each separated model performed and to see if they worked properly or needed tweaking. Every 100 rows of the dataset is a time-period, and is given a rating that is the average of all ratings in the window, and represented as a one-hot vector. One hundred rows per time-period was chosen as this represents five seconds of application runtime with the application running at 60 frames per second and only every third frame being recorded in the dataset. The dataset is then split into a training and testing set, where the split is 50%/50%. A split of 80% training data and 20% testing data was originally tried, but was found to cause the individual models to overfit and perform poorly in testing. Time periods were batched together in sets of 2 to make training faster. Typically, with time series data like this dataset, each time period would be somewhat dependent on the past time-period. This is still somewhat true, the past time-periods have fairly little impact on the current period. This means we can shuffle the order of the periods to try and reduce overfitting in the model and help it generalize more effectively. Only the training data was shuffled. Lastly, the resolution of the frame images was cut to a fourth as the dataset was loaded to stay

within the available system memory of the training machine. This leaves the images with a result of 256 by 131 pixels.

Training the combined model was much the same as the separated models. The primary difference is the combine model did not overfit quite as severely and uses the original 80%/20% split for training and testing data. Since the model has two inputs, the numeric data and the image data, both need to be zipped together for each row before being batched into time periods and batches. Additionally, the combined model was set to train for longer with 250 epochs maximum.

2.2 Motion Sickenss Prevention In An Application with Vignetting, Snap Turning, and User Warnings

To apply our model, we created an application that would likely create motion sickness and employed several standard methods for reducing that motion sickness based on the inferences of the model.

2.2.1 The Environment

The application was made using the Unity game engine as well as Unity's XR Interaction Toolkit[18]. It features two roller-coasters the user can ride, one being a simple circle and the other being more complex. The user had both continuous and teleport locomotion methods for moving around the environment. The user enters roller-coasters by pointing at and clicking a button in the environment, after which they will be seated in the roller-coaster's car and will ride along the track. Afterward, they will be dismounted from the ride and able to walk around again as well as select another coaster to ride. The environment contains some hills made using Unity's Terrain Generation tools and realistic texturing received from `https://3dtextures.me/`[19], so the environment resembles those in the dataset instead of a completely flat environment. The environment is shown in **??**. Within this environment, three motion sickness reduction features were implemented.

Figure 2.2. Two images of the virtual environment from our application: one from the ground(left) and the other on the complex roller coaseter(right)

2.2.2 Vignetting

Vignetting is the term for reducing the user's field of view with some sort of visual filter akin to a vignette. This removes visual context suggesting to the user that

they are moving, which their vestibular system would contradict, causing motion sickness. Typically, this is done in games based purely on weather the user is locomoting or not. In this application, the size of the vignette was determined by the rating provided by the model, where the sicker the user was, the larger the vignette and the less of their periphery the user could see. The vignetting was provided by Unity's Tunneling Vignette Shader[3] and examples of our vignetting are shown in **??**.

Figure 2.3. The perspective of the user with a black vignette reducing their field of view.

2.2.3 Snap Turning

Snap Turning refers to larger user rotations that occur instantaneously with a single input. For instance, the user would press right once on the joystick to "snap" 45° instantaneously. This is contrasted by the normal method of holding the joystick right to turn at a constant velocity over time. This reduces motion sickness by reducing the motion seen but not felt by the user as they rotate over large increments instantly and simply rotate their head for smaller increments. By default, the application uses continuous turning, unless the user has a motion sickness rating greater than two from the model, where they then use snap rotation at increments of 45°.

2.2.4 User Warnings

Some applications will warn the user to take a break if they have used the application for a certain amount of time. Our application will display a warning if the model has rated the user as having a motion sickness of five, which is the maximum. The warning (seen **??**) tells the user to take a break and is attached to the user's vision. The warning is removed after five seconds with a lower motion sickness rating from the model. This is meant as a last resort if nothing can reduce the users' detected sickness as to minimize the effects of prolonged sickness. Below is an image of said warning.

Figure 2.4. The perspective of the user with red text near the bottom, stating "You seem very sick. Please take a break soon!"

2.3 Marrying Detection and Prevention

Once the application was built and the model was trained, the two had to be combined. This was not as easy as one would think.

2.3.1 Pulling the necessary data for inference

Pulling controller inputs, and camera matrices were easy as Unity provided functionality to access this data, the device pose was more complicated. The dataset and therefore the model had recorded the position and rotation of each device in a 4x3 transformation matrix. Unity, however, provides the position as a vector and the rotation as a quaternion[20]. Unity provides a function for retrieving a transformation matrix from a translation(position), rotation, and scale(which would remain as $< 1, 1, 1 >$). This function returns a 4x4 matrix, however, where the last row is always $\begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}$. Since the last row is always the same and one row needed to be discarded, the last row was discarded to create the 4x3 matrices needed for the poses.

While originally the numeric and image data from the application would be pulled each frame, this approach introduced massive performance losses. This is because of how the frame image was represented. Two images were pulled from the virtual cameras representing the left and right eye's perspective. They were then attached in one image as a Unity Texture2D. This datatype requires a texture format, and while many are available, none were quite suitable. The model is trained on IEEE floating point numbers, but most formats were either integers, or compressed formats that would have required decompression. Of the floating point formats available, the options were RFloat, RGFloat, or RGBAFloat. The first two formats lack the blue channel the model needs for influence while the last has an additional channel(alpha) that needed to be removed. To remove the alpha value from the image, we iterated over each value in the image copying it to a float array unless it was the alpha value. Since iterating over the image, meant iterating over $256 * 131 * 4$ or 134144 values, this took considerable time, around 60ms. However, frames need to be generated in around 18ms and this code would have otherwise held the main thread, reducing the framerate to around 15 frames per second, which would make the user more motion sick among other things. This required moving the logic into another thread; using Unity's Jobs system. Jobs maintain thread safety by requiring that the user only work on memory explicitly allocated for the job. This meant that before starting a job, all data needed for the job would have to be polled, then copied to the job's own memory; then the job could be run. When the job finished, the data would be copied back from the job's memory back to the main processes' memory.

This culminated in putting together the last one-hundred frames of numeric and image data to be processed in a job. First, the current time-period would be sent to the job with the new numeric and image data from the current frame. Jobs also only work with arrays because of specific memory allocation constraints. Therefore, the job would copy the latest 99 frames from the end of the array containing them to the beginning, and would copy the newest data to the end. For numeric data, this was simply constituted copying from an array containing that data. For the new image, each value was iterated through to only copy the red, green, and blue values, but not the alpha values. Then the job would end and the new data would be copied back. This process alone would take two or three frames worth of time, meaning the time-period only polled every second or third frame, which matched up with what the model was trained on. This, however, was only to ready data to be given to the model.

2.3.2 Loading the Model in Unity

The model was originally planned to be loaded directly from with Unity. This would have been possible as Unity uses the C# programming language, and Microsoft provides a framework for performing ML tasks in their aptly named ML package. The major benefits of loading the model within C# are that it's much more performant. This is because moving data into the model is fast and the ML library can more easily make use of the GPU for influencing than tensorflow from within Windows. Influencing using the GPU is important as, compared to the CPU, it is much better at handling matrix math and manipulations which primarily make up the computation of CNNs. Unlike keras, Microsoft's ML package uses a different model storage format and runtime named Onnx. Converting a keras model to onnx is relatively easy. However, no matter what was tried, the onnx runtime would not function.

2.3.3 Using Inter-Process Communication Instead

Without the Onnx runtime, we had to use a different method to load the model for influencing. Since the model would load in python, we decided to load it in python using tensorflow and keras as we did for training. This necessitated a new process to run the model, separate from the C# process that gathered the data for inferencing. This required the use of interprocess communication, specifically sockets.

Sockets are the same thing used to send data over the internet. Each process creates a socket, one process is a server waiting for clients to connect, clients then connect and send data, the server sends data back. In this case, the python process with the model would be the server and would have to take care to manage the

connection received by the C# process. The python server would wait for a connection, receive the data it needed for inference, get a rating for the user's motion sickness, and send that rating back to the client, and finally disconnect.

One would think to hold open the connection and let both processes talk until either ended. However, because getting GPU inferencing to work in Windows within TensorFlow is rather finicky, CPU inferencing, which is much slower, is needed instead. This slowness, like with the image iterations, would have wreaked havok on performance, so the C# process needed to communicate with the python process through a job on a separate thread. Since jobs have their own segment of memory that cannot be shared with the main process, the connection cant be passed between the main thread and the job. The job also can't hold onto the connection as its memory is deallocated when the job is complete, unless it's copied back to the main process, which couldn't be done. This necessitated creating a new socket connection to the python server for every job (which occurs every 2 to 15 frames), after the processing of the new frame's data was completed. Once the inference was received, the job would terminate and the prediction would be passed back to the main thread to be handed to the C# that handled the motion sickness mitigations.

CHAPTER 3
RESULTS

The model was quite accurate on the testing portion of the dataset. The separate numeric and image models each attained about 60% accuracy. This isn't great but much better what would be 20% if the model were guessing randomly. The combined model, however, got an accuracy score of 85%. This is significantly better than either of the separate models. However, the model's accuracy on the dataset is not the same as its real-world efficacy. Most rows in the dataset are rated as a one of five, meaning a decent accuracy can be achieved by simply assuming 1 if no better rating can apply.

The model's predictions apply decently well to a real-world application. When the user looking around or physically walking around, the model rates them as having no motion sickness (rating of 1). This means the model is not rating the user as sick when the user is moving in a way that does not confuse their vestibular system. In contrast, the model rates them as being slightly sick (rating of 2) when the user moves using their joystick. This is good as the model is not overreacting and can differentiate between motion that the user can physically feel and motion that is only seen. Not only that, but this behavior effectively re-implements locomotion-based vignetting, which is common in most VR applications. When the user is riding a rollercoaster and looking forward, the user is rated as moderately sick (rating of 3). This rating being higher than walking is expected, but the model does not overreact and apply the highest rating which is good. When the user isn't looking forward to the rollercoaster but to the side, the model rates the user as either not sick or low sickness (rating 1–2). This makes sense as the user is perceiving motion for objects in the environment where looking forward the motion could only be perceived as the user's movement.

The performance overhead of this system is significant. With the model and data formatting operations, the application takes 30 to 40ms to produce a frame compared to 16ms without them. This equates to a frame rate of 30 frames per second. This is not unusable but may cause more motion sickness, and for performance critical applications this may just not be feasible to apply. This performance is in part due to bad implementation. Assuming the model was inferencing using the GPU and a better method of loading and passing data to the model were found, and less image processing had to be done, the overhead could be significantly reduced. These could easily be resolved with more time.

CHAPTER 4
CONCLUSION

In conclusion, using a CNN to manage motion sickness mitigation in a VR application is absolutely feasible given that the overhead from the model is taken into account. We used the VRNet motion sickness dataset and created an ensemble model that could predict with an accuracy of 85% on the held-out samples of the dataset. We created an example VR application with a somewhat detailed environment and motion sickness mitigations. When the model was put in control of those mitigations, it made decisions that were fairly logical, though with some performance concerns.

4.1 Future work

The obvious first steps are to make influencing and collection of data from the engine more efficient. Beyond this, the model could be extended to increase accuracy by adding "Long Short-Term Memory" (LSTM) layers so the model can make more improved predictions. Note that this would require not shuffling the trained time periods. There are other motion sickness mitigation features that could be implemented and controlled by the model. A dataset that could classify the user's motion sickness by cause could provide more specific mitigations by reducing the specific cause of motion sickness. A better model and more performant application should be tested on real users to see its real world efficacy. If all these things were done and the improved application was proven to cause less motion sickness, this system could be deployed in many applications to reduce motion sickness at the best time, before it happens.

BIBLIOGRAPHY

[1] S. D. Hoang, S. K. Dey, Z. Tučková, and T. P. Pham, "Harnessing the power of virtual reality: Enhancing telepresence and inspiring sustainable travel intentions in the tourism industry," *Technology in Society*, vol. 75, p. 102378, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0160791X23001835

[2] Apr 2016. [Online]. Available: https://store.steampowered.com/app/457550/Bigscreen_Beta/

[3] U. Techologies, "Tunneling Vignette Controller — XR Interaction Toolkit — 2.1.1 — 2.1," https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@2.1/manual/tunneling-vignette-controller.html, 2023, [Accessed 03-04-2024].

[4] U. Technologies, "Teleportation Provider — XR Interaction Toolkit — 2.1.1 — 2.1," https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@2.1/manual/teleportation-provider.html?q=teleport, 2023, [Accessed 03-04-2024].

[5] ——, "Snap Turn Provider (Action-based) — XR Interaction Toolkit — 2.1.1 — 2.1," https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@2.1/manual/snap-turn-provider-action-based.html?q=snap, 2023, [Accessed 03-04-2024].

[6] R. K. Kundu, A. Rahman, and S. Paul, "A study on sensor system latency in vr motion sickness," *Journal of Sensor and Actuator Networks*, vol. 10, no. 3, p. 53, 2021.

[7] C. Zhang, "Investigation on motion sickness in virtual reality environment from the perspective of user experience," in *2020 IEEE 3rd International Conference on Information Systems and Computer Aided Education (ICISCAE)*, 2020, pp. 393–396.

[8] U. A. Chattha, U. I. Janjua, F. Anwar, T. M. Madni, M. F. Cheema, and S. I. Janjua, "Motion sickness in virtual reality: An empirical evaluation," *IEEE Access*, vol. 8, pp. 130 486–130 499, 2020.

[9] X. Gong, "Research on minimizing VR motion sickness in VRChat," in *International Conference on Artificial Intelligence, Virtual Reality, and Visualization (AIVRV 2021)*, S. Chen, T. Li, D. Wu, and G. Gao, Eds., vol. 12153, International Society for Optics and Photonics. SPIE, 2021, p. 121530L. [Online]. Available: https://doi.org/10.1117/12.2626662

[10] Z. Qiu, M. McGill, K. M. T. Pöhlmann, and S. A. Brewster, "Display rotation for reducing motion sickness caused by using vr in vehicles," in *Adjunct Proceedings of the 14th International Conference on Automotive User Interfaces and Interactive Vehicular Applications*, ser. AutomotiveUI '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 25–29. [Online]. Available: https://doi.org/10.1145/3544999.3552489

[11] S. Hell and V. Argyriou, "Machine learning architectures to predict motion sickness using a virtual reality rollercoaster simulation tool," in *2018 IEEE International Conference on Artificial Intelligence and Virtual Reality (AIVR)*, 2018, pp. 153–156.

[12] T. M. Lee, J.-C. Yoon, and I.-K. Lee, "Motion sickness prediction in stereoscopic videos using 3d convolutional neural networks," *IEEE Transactions on Visualization and Computer Graphics*, vol. 25, no. 5, pp. 1919–1927, 2019.

[13] N. Padmanaban, T. Ruban, V. Sitzmann, A. M. Norcia, and G. Wetzstein, "Towards a machine-learning approach for sickness prediction in 360° stereoscopic videos," *IEEE Transactions on Visualization and Computer Graphics*, vol. 24, no. 4, pp. 1594–1603, 2018.

[14] J. Kim, H. Oh, W. Kim, S. Choi, W. Son, and S. Lee, "A deep motion sickness predictor induced by visual stimuli in virtual reality," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 2, pp. 554–566, 2022.

[15] H. Gao and E. Kasneci, "Eye-tracking-based prediction of user experience in vr locomotion using machine learning," *Computer Graphics Forum*, vol. 41, no. 7, pp. 589–599, 2022. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14703

[16] E. Wen, T. I. Kaluarachchi, S. Siriwardhana, V. Tang, M. Billinghurst, R. W. Lindeman, R. Yao, J. Lin, and S. Nanayakkara, "Vrhook: A data collection tool for vr motion sickness research," in *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3526113.3545656

[17] E. Wen, C. Gupta, P. Sasikumar, M. Billinghurst, J. Wilmott, E. Skow, A. Dey, and S. Nanayakkara, "Vr. net: A real-world dataset for virtual reality motion sickness research," *arXiv preprint arXiv:2306.03381*, 2023.

[18] U. Technologies, "XR Interaction Toolkit — XR Interaction Toolkit — 2.0.4 — 2.0," https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@2.0/manual/index.html, 2023, [Accessed 04-04-2024].

[19] J. Paulo, "3D Textures," https://3dtextures.me/, 2024, [Accessed 04-04-2024].

[20] J. Voight, *Quaternion algebras*. Springer Nature, 2021.