VR RT^2: VR-Integrated Real-Time RaceTrack Simulator

Angelos Mavrogiannis
University of Maryland
College Park, MD, USA
angelosm@cs.umd.edu

Zining Zhang
University of Maryland
College Park, MD, USA
znzhang@cs.umd.edu

Logan Stevens
University of Maryland
College Park, MD, USA
lsteven7@umd.edu

Elliot Huang
University of Maryland
College Park, MD, USA
ehuang12@umd.edu

Hyekang Kevin Joo
University of Maryland
College Park, MD, USA
hkjoo@cs.umd.edu

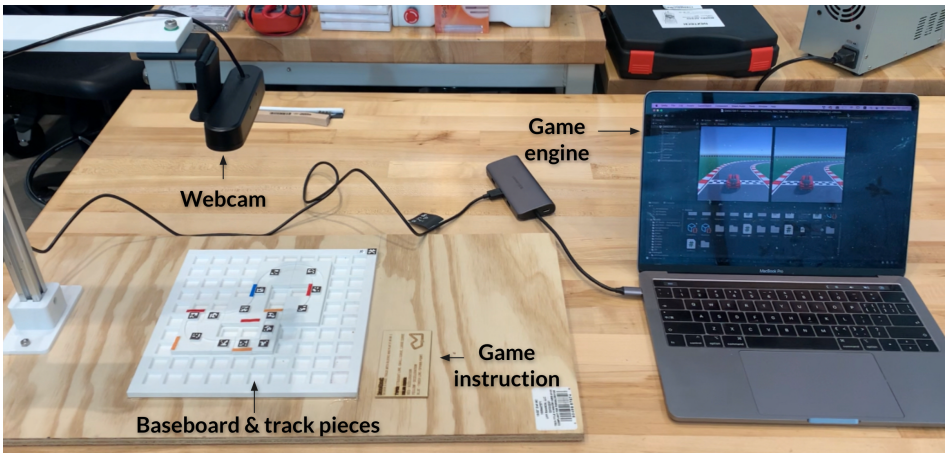

Figure 1: VR RT^2 experiment setup

ABSTRACT

In this paper, we present an interactive system that converts a physical racetrack to a VR-simulated racing environment. The user begins by constructing a physical racetrack using a set of 3D-printed track pieces on a 3D-printed baseboard that resembles a chessboard. A top-down view of the 2D design of the track is captured in real time with an overhead camera. The track pieces are segments of straight or curved lines to model a straight part of a road or a turn, and the user can incorporate additional functionalities upon some of these pieces, triggering different functions in the corresponding parts of the simulated environment (e.g. acceleration or deceleration of the vehicle). Each of the pieces has an ArUco marker glued on its center, which is detected from the captured video stream. By extracting the coordinates and marker id of each of the markers we compute their relative position and orientation and create an occupancy grid representation of the physical map in a matrix form which is saved to a text file at every frame. The text file is then processed and decoded in a real-time Unity environment which renders the virtual racetrack with a game engine.

KEYWORDS

human computer interaction, computer vision, 3d printing, simulation, virtual reality

ACM Reference Format:
Angelos Mavrogiannis, Zining Zhang, Logan Stevens, Elliot Huang, and Hyekang Kevin Joo. 2018. VR RT^2: VR-Integrated Real-Time RaceTrack Simulator. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The main idea of this project is to transfer a physical racetrack to a digital representation in a virtual reality (VR) simulated environment where an ego vehicle can be controlled by the main user. This concept can either be interpreted as a game, where the user navigates through the track with the goal of achieving a minimum lap time or as a platform for testing (autonomous) driving-related algorithms in custom-designed tracks. Our initial plan involved drawing a racetrack on paper or on a whiteboard but we advanced our idea to incorporate some hardware components that we could design and manufacture ourselves. Our approach is presented in the form of a game, where the user constructs a physical map by

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

arranging a set of 3D-printed square pieces that represent road segments on a 3D-printed base that resembles a chessboard. Some of the pieces incorporate additional functionality by triggering special effects (attributes) in the simulation, like accelerating or decelerating the ego vehicle for specific parts of the simulated racetrack. These special pieces are color coded so that the user can distinguish them from the rest of the normal pieces.

2 RELATED WORK

The key inspiration for this work can be attributed to the MazeDraw project [1]. In Maze Draw, the user draws a maze on a whiteboard with a marker and then using an HD camera and OpenCV, the physical drawing is converted to a digital drawing, which is then adapted to a 3D-simulated maze in Unity that the user can navigate through using an Oculus Rift headset. The computer vision part is implemented as a color-based line segmentation, given that the user has access to a colored marker and a whiteboard. Our project is very similar to MazeDraw, however, instead of drawing lines we are using 3D-printed pieces and we attach an ArUco marker on each piece to track each piece in real time. Furthermore, instead of a maze environment, we focus on a more complicated racetrack environment consisting of straight lines and turns of multiple sizes. Mario Kart Live: Home Circuit [3] is another relevant product. This is a Nintendo Switch game where users can set up and create custom environments in their own house by arranging various obstacles or landmarks with computer vision-recognizable patterns on them, and then drive a MarioKart in a simulated environment that is an enhanced version of their house. At the same time, a real kart is also being controlled in the real-world environment (house map). The landmarks have unique functions that trigger special effects (attributes) in the game, just like our acceleration-varying color-coded pieces.

3 SYSTEM OVERVIEW

Figure 2 shows the VR RT^2 system architecture, which is composed of a physical racetrack kit with a map baseboard and multiple track pieces that provide physical interaction for users, an HD webcam that captures real-time racetrack design on the baseboard, and a Unity game engine that can simultaneously render the racetrack design in virtual reality.

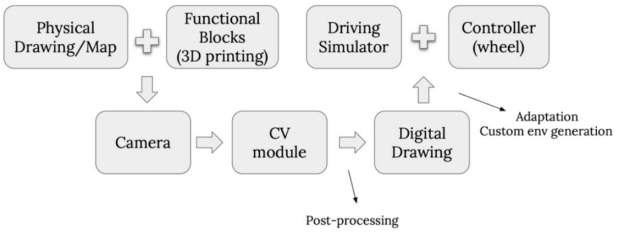

Figure 2: VR RT^2 system architecture

3.1 Hardware

When designing a system of pieces that can form various race tracks, we first started in 2D and then moved to the 3D environment to refine and optimize the models for 3D printing. Our main goal was to use as few unique pieces as possible to create as many different tracks as possible in a fixed-size canvas.

3.1.1 2D design. After plenty of research and multiple iterations of design and prototyping, the final track design has three types of unique pieces (Figure 3) for track building–a one-unit-size straight line piece, a one-unit-size 90° curve piece, and a two-unit-size 90° curve piece. These three unique pieces can enable almost limitless track design, and at the same time, prevent user confusion. The main idea behind our design is modularity; for instance, the user can connect two curve pieces to represent a 180° curve. The track below, on the right, is an example of a race track formed by our three-piece system.

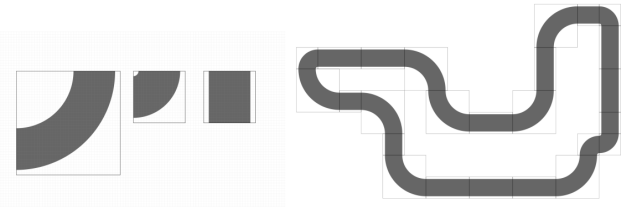

Figure 3: 2D track pieces design

3.1.2 3D design. Based on the Webcam resolution and Ergonomics, the dimension of the one-unit-size piece is designed to be 20 mm x 20 mm, and accordingly, the two-unit-side piece is settled to be 40 mm x 40 mm. To better improve user experience, a grid baseboard for easy alignment is added to the system. In this way, the users no longer need to carefully align and put pieces together; instead, they can just put the pieces on the grid baseboard and the pieces will always align perfectly. Due to the ArUco markers used for piece detection, a placeholder for the ArUco marker of each piece is designed at the center of each 3D model (Figure 4).

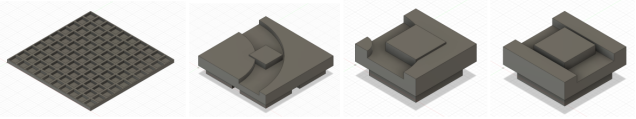

Figure 4: baseboard and track pieces 3D modeling

3.1.3 Fabrication & Assembly. As shown in Figure 5, we used an FDM 3D printer with PLA filament to fast print the baseboard and 3D track piece models. Each ArUco marker is later attached to the placeholder of each piece by using hot glue. Furthermore, To better visually differentiate the functionality of each piece (i.e. none, acceleration, and deceleration), we applied a color-coded method to the physical pieces.

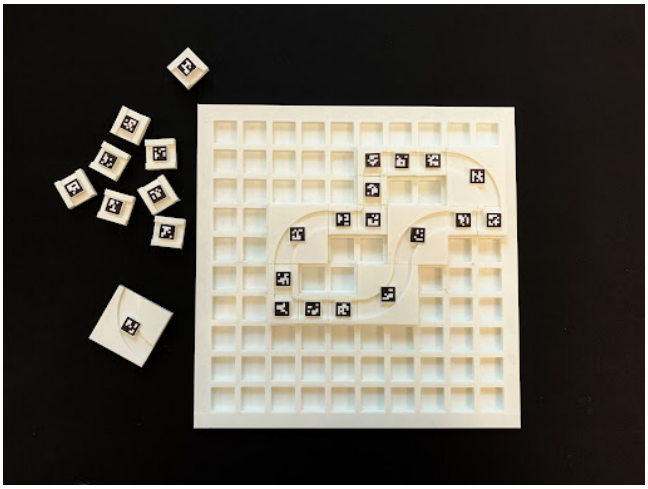

Figure 5: 3D printed baseboard and track pieces

3.2 Computer Vision

3.2.1 Marker Detection & Identification. We generate a total of 26 ArUco markers using the 5×5 ArUco dictionary. The marker with $ID = 25$ is glued at the top left corner of the board and serves as a reference marker, enabling us to compute the relative distances of all the other markers that are placed on the board with respect to the coordinates of this marker. The rest of the markers are divided into straight line segments (*IDs* 1 to 14), small turns (*IDs* 15 to 19), and large turns (*IDs* 20 to 24). We generate and detect the markers using the ArUco module of the OpenCV library and extract the coordinates of the four corners of each marker for every frame of the input video stream. The figure shows the detected markers for a sample configuration, along with their centers and approximated contours.

3.2.2 Orientation. Having acquired the coordinates of the four corners of each marker placed on the board, we can easily compute the coordinates of the point in the center of each marker as well as its orientation. Given the geometry of the board and assuming that i) the camera will always be placed on top of it capturing bird's eye view images, and ii) the board will always be placed in an orientation where the reference marker ($ID = 25$) lies on its top left corner, there are only four possible different orientations for a marker: $\{0°, 90°, 180°, 270°\}$ where each of these numbers represents a potential clockwise rotation of the marker with respect to its original orientation as extracted from the ArUco dictionary. The *detectMarkers* function returns the detected corner coordinates for every marker in an ordered form and we leverage it to easily compute the orientation. For example, as we can see in Figure 7 if the top left horizontal coordinate (TL_x) is larger than the bottom left horizontal coordinate (BL_x), then the piece has an orientation of 90°. If the difference is smaller than a small user-defined margin ϵ number of pixels ($\epsilon = 10$), then we compare a different set of points, as it will lead to an incorrect orientation because of noisy measurements returned from the detection. For example, if we did not consider this margin, and $TL_x = 100$, $BL_x = 99$ yielding $TL_x > BL_x$, but in reality the true orientation was 180°, we would get an incorrect classification as 90°. Finally, we do not glue the markers following their original, unrotated orientation on the pieces, but we set an orientation notation followed by the straight line and turn pieces when saving orientation information to file at the end of the computer vision pipeline. This notation can be seen in Figure 8. Computationally, we convert the orientation of every piece to one of these integer numbers by first computing the difference between the real-time-tracked orientation and the original orientation in the ArUco dictionary.

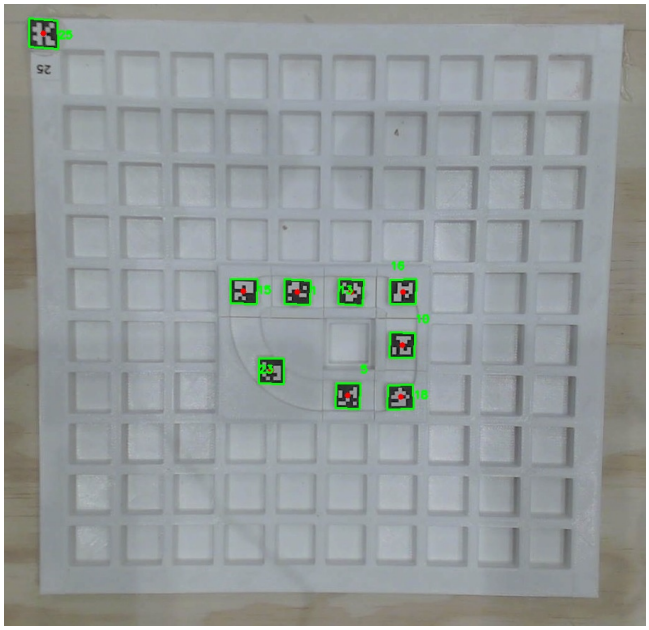

Figure 6: The output of the ArUco marker detection part using an image captured with our mounted camera, showing a sample board configuration.

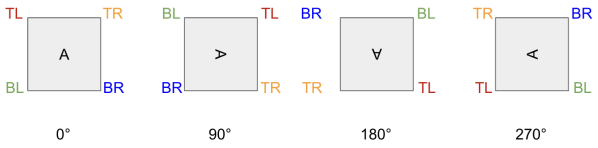

Figure 7: The four different orientations that each square ArUco marker can have. Each successive pose is a clockwise rotation about 90 degrees with respect to the previous (left) pose.

3.3 Occupancy Grid Map

Generating a simulated racetrack in Unity that accurately replicates the physical racetrack on the board heavily relies on the accurate tracking of the markers' relative position and their absolute position on a 10×10 grid. An easy brute-force solution would be to assign grid coordinates to a piece based on its absolute distance to the top left corner of the image or to the reference marker. For example,

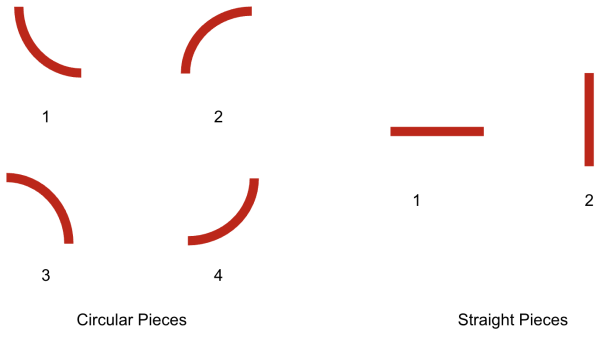

Figure 8: The orientation legend used when saving the orientation information to a text file.

we could measure the size s of a piece in the image space and if e.g. a marker's horizontal coordinate x lies within 0 and s from the reference marker, assign the coordinate 1 to it (first block). However, this approach would not work if we decided to change the distance of the camera to the board, or if a user accidentally moved the camera. To make the system more robust and invariant to camera motion, we compute the pairwise distances between small pieces, between small and large pieces, and between large pieces, when a pair of pieces is placed in adjacent blocks. These pairwise distances are different for different camera poses, but we established an invariant relationship between these pairwise distances and the easily detectable top left marker center, which is a fixed point. More specifically, if d is the pairwise distance between two small pieces, we tested and confirmed that for a detected piece with $ID = i$ the quotient of the division $(x_i - x_{25})/d$ (where x_{25} is the horizontal pixel coordinate of the reference marker with $ID = 25$) yields the exact grid coordinate (starting from 0) of the piece in the x direction (the same is true for the y direction). To extract the pairwise distance between two small pieces, the map needs to have at least two small pieces placed in adjacent blocks. If there are only large pieces or the small pieces are not placed in adjacent blocks, then we compute the large-to-small or large-to-large pairwise distance and compute the small pairwise distance using an invariant ratio connecting the sizes of the pieces, which has been computed offline.

3.3.1 Real-time Output Production. Once we are done with orientation and occupancy grid computation, we save the position and orientation information in two text files respectively, both in an occupancy grid matrix form. The first text file incorporates the location and ID information of the track pieces in a 2D array format, split by space or newline to denote separability. In the other file, we store the information about the orientation of the piece corresponding to the location. The following position (P) and orientation (O) matrices show the exact output produced when we encounter the configuration seen in Figure 6:

$$P = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{15} & \mathbf{1} & \mathbf{13} & \mathbf{16} & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{23} & \mathbf{23} & 0 & \mathbf{10} & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{23} & \mathbf{23} & \mathbf{5} & \mathbf{18} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$O = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{2} & \mathbf{1} & \mathbf{1} & \mathbf{3} & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{1} & \mathbf{1} & 0 & \mathbf{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{4} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

To allow real-time human-computer interaction in the simulated environment, we design the system so that the output is produced at every frame of the video stream to constantly keep the racetrack updated at any time. This allows to make dynamic changes on the physical racetrack which are immediately reflected to the simulated track in Unity. For example, if we remove a piece from the track, then Unity shows a green-colored area in this part of the track which indicates that this spot does not represent a part of the road (the road is represented with gray to model the asphalt).

3.4 Game Engine

3.4.1 Rendering. The VR simulation runs on the Unity game engine and uses publicly available assets and experimental first-party Unity XR software [4] to deliver a simulation that can be used in conjunction with a Google Cardboard-like smartphone-reliant viewing interface and an Xbox game controller. Each grid piece on the baseboard grid is replicated on the surface of the Virtual Environment (VE) via strategically placed custom Unity GameObjects that activate/deactivate and move/rotate according to the output data from the Computer Vision (CV) (See Figure 9). Our custom C# scripts in Unity have a parser that reads from the text files that are output from the CV system in real-time (those text files being the matrices (P) and (O)) [2]. To extract the proper virtual track models for the simulation to render on the virtual baseboard, the only pieces of data needed from the CV system are the ID of each present piece along with their physical grid location and orientation angle. From the ID number of the piece, the track model (e.g., straight, curved, etc.) and its attribute can be extracted. From the location of the ID within the given matrix (P), the Cartesian coordinates of that piece can be extracted. Finally, the data from (O) is extracted and each orientation angle present is applied to each virtual piece existing at the Cartesian coordinates of each extracted angle in (O).

3.4.2 Game and Control Aspects. One of the track pieces that must be somewhere on the board for the player to operate the ego vehicle fully is the finish-line track piece. This is where the user can

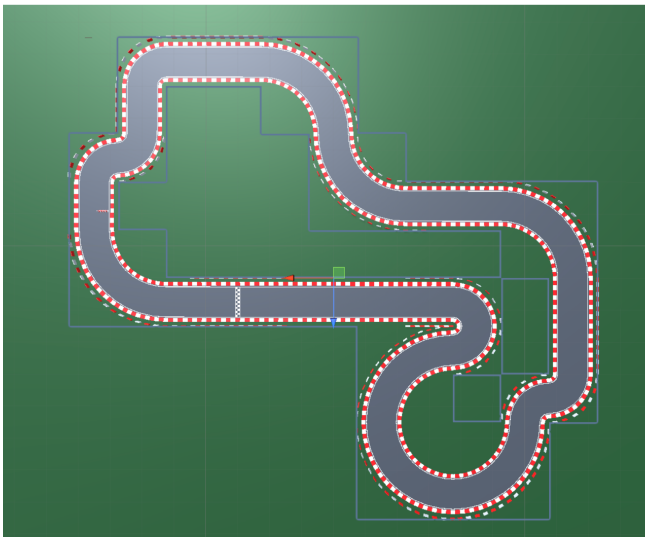

Figure 9: Top-down view of the racetrack as seen in the Unity editor.

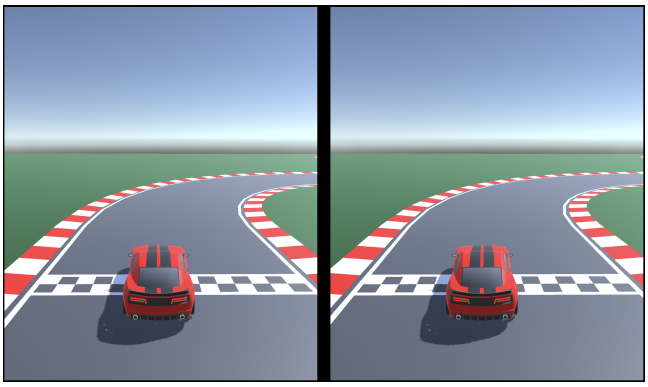

Figure 10: User view of the simulation.

spawn/re-spawn their vehicle to at any given time via a button press. The vehicle model in the simulation is a free vehicle Unity asset [5]. Our VE also uses 3D models from a free modular racetrack Unity asset [6]. Custom control interface scripts were written in the C# programming language to handle user input from either controller or keyboard input [2].

3.4.3 Virtual Reality. VR is achieved through the use of the Unity Mock HMD XR Plugin [4]. This plugin provides stereo rendering support by mimicking the display properties of an HTC Vive Head Mounted Display (HMD) (See Figure 10). While intended to aid in VR development by circumventing the need for a dedicated HMD, this mimicking property of the plugin is an easy way to provide smartphone VR support, which greatly increases the accessibility of this software.

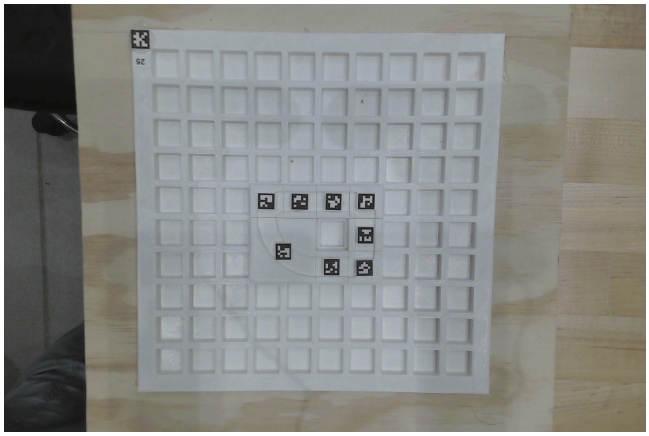

Figure 11: Top-down view of the board used for homography estimation

4 CONCLUSION, LIMITATIONS & FUTURE WORK

We designed and built a 3D-printed board with 3D-printed pieces to represent a physical racetrack and proposed an approach to convert it to a digital racetrack in Unity and control an ego vehicle to navigate in an exact replica of the physical environment. Our approach has some limitations. In terms of the hardware part, there is only one-way communication from the physical map to the virtual environment but the system lack feedback from virtual reality to the physical component. In the future, to achieve two-way communication, a laser projector can be mounted aside from the Webcam for projecting the current location of the virtual vehicle on the map. In terms of the VR simulation aspect, there is no first-person view from a driver's perspective. Implementing the feature necessitates an accessible high-quality Unity-compatible 3D model and additional custom interaction features. In terms of the computer vision part, the board needs to be in a room with adequate light for the detection to work properly. Furthermore, the approach would not currently work if the camera was totally dismounted and placed far away from the board. To make this robust, we would need to estimate the homography between the frame captured from far away from the board and a top-down frame of the board. We would need four well-spread points in the two images for a high-quality homography, so the four corners of the reference marker are not enough, since they are very close to each other. Hence, we tried to approximate the coordinates of the board corners using edge and corner detection, Hough line detection, and color segmentation, but due to the variable light conditions and the background color of the board not being very far in the RGB color scale, the board segmentation did not exhibit consistent behavior. Using a black color for the board would facilitate edge detection and allow us to easily segment the board and compute its corner coordinates for the homography. The currently supported function requires the user to manually select the four corners of the board using the mouse cursor on the new frame. The results for the homography are shown in Figures 11, 12, and 13.

Figure 12: Picture of the board taken from a different distance and angle

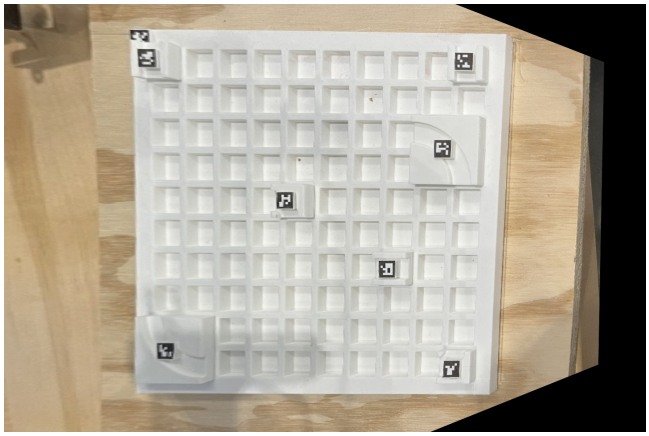

Figure 13: Transformed version of the image shown in Figure 12. This is the perspective transformation returned using the homography matrix computed from image 12 to image 11

REFERENCES

[1] Osterfelt N. Wells W. Boylan-Peck, B. 2018. MazeDraw. https://github.com/NickOsterfelt/MazeDraw.

[2] Logan Stevens. 2022. GitHub Repo for VR-racetrack. https://github.com/loganstevens/racetrack.

[3] Velan Studios. 2020. Mario Kart Live: Home Circuit. https://mklive.nintendo.com. Accessed: 2022-12-18.

[4] Unity. 2019. About the Mock HMD XR Plugin. https://docs.unity3d.com/Packages/com.unity.xr.mock-hmd@1.3/manual/index.html. Accessed: 2022-12-18.

[5] Unity. 2022. Unity | ARCADE: Racer. https://assetstore.unity.com/packages/3d/vehicles/land/arcade-free-racing-car-161085. Accessed: 2022-12-18.

[6] Unity. 2022. Unity | Modular Lowpoly Track Roads. https://assetstore.unity.com/packages/3d/environments/roadways/modular-lowpoly-track-roads-free-205188. Accessed: 2022-12-18.