AR Vision: Immersive Entertainment through Virtual Devices

Ching-Hsiang Wu[1], Jung-Hao Kao[2], and Ju-Hsuan Weng[2]

[1] Dept. of Electrical Engineering
[2] Dept. of Computer Science and Information Engineering
National Taiwan University, Taipei 106319, Taiwan

Abstract. Our project aims to simulate the Apple Vision Pro, allowing the player to play games in reality. We utilize an ArUco Marker and a cellphone for camera localization. Users can easily perform actions through their gestures.

Keywords: Augmented Reality · Image warping · Gesture recognition

1 Motivation

All eyes are on the upcoming launch of the Apple Vision Pro in the next year (2024). This powerful gadget allows users to play games or interact with 3D objects in reality, controlled by hand gestures or even voice commands.

However, it comes with a hefty price tag of $3,499, making it unaffordable for many people. Therefore, our goal is to develop a cost-effective yet efficient system that resembles Apple Vision Pro.

Fig. 1: Apple Vision Pro

2 Problem Definition

To develop a system resembling the Apple Vision Pro, we use a cellphone to simulate AR glasses and a computer to show what the user sees. In addition to presenting the real scene, the computer will also show the game board. The game board is fixed in a virtual location, so its position on the screen varies as the player moves.

We implement the Minesweeper as the game board in our project. All actions can be performed through gestures. Our system supports two modes: the game mode, where the player can play the game, and the setting mode, where he or she can adjust the scale or the position of the board.

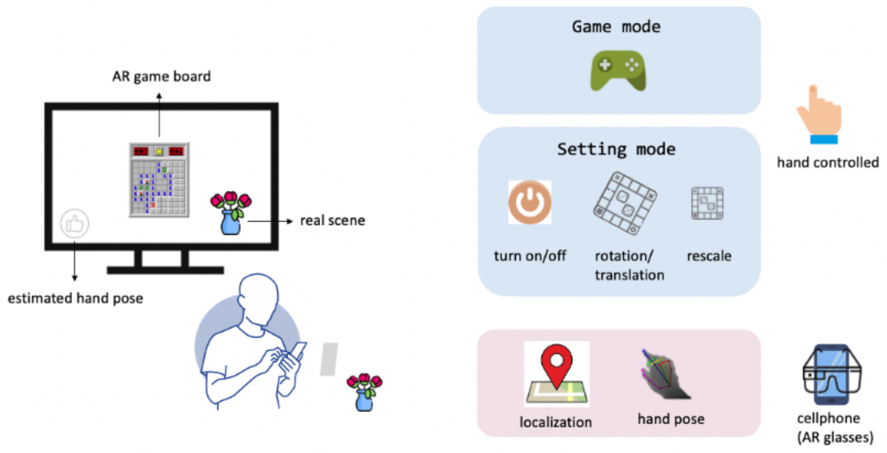


Fig. 2: Problem definition

3 Methodology

The system's pipeline is illustrated in Fig. 3. The current mode can be controlled through hand gestures. The core concept of our design involves representing the game board as an image. In the setting mode, we can easily apply homography transformations, such as rotation and translation, to this image. In game mode, the game board is updated by modifying the image.

Once the image is transformed, we project it onto the screen using image warping techniques.

3.1 Gesture Recognition

In this project, we classify gestures into two main categories: static and dynamic gestures. Static gesture recognition involves determining gestures from a single frame, while dynamic gestures require analysis across multiple continuous frames.

Our gesture recognition relies on the MediaPipe Hand Landmarker task, developed by Google Research, which allows us to detect landmarks of the hands

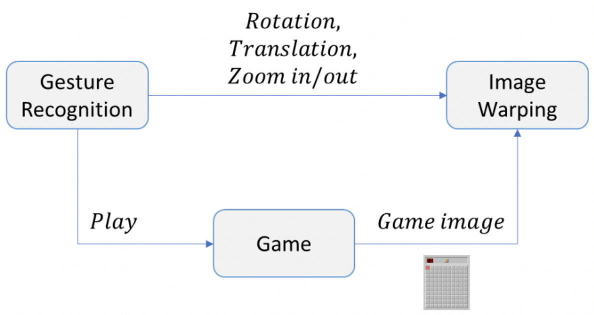


Fig. 3: Overall pipeline

(Fig. 4) in an image. This task enables the identification of key hand points, facilitating the application of visual effects. It operates on image data using a machine learning model as a continuous stream, producing hand landmarks in both image and world coordinates, along with the handedness (left/right hand) of multiple detected hands.

Interestingly, there's also the MediaPipe Gesture Recognizer task available, designed for real-time recognition of hand gestures. This task provides recognized hand gesture results alongside the landmarks of the detected hands. However, it's worth noting that this task is currently limited to recognizing only 7 specific types of gestures. Although it supports the creation of custom gesture recognizers, its capability remains constrained to identifying static gestures and doesn't cover dynamic gestures.

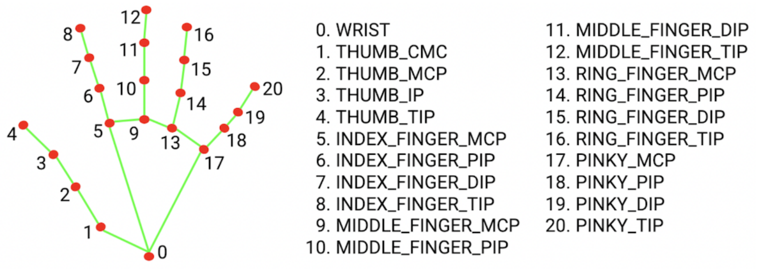


Fig. 4: 21 hand-knuckle

3.1.1 Gesture List

The gestured listed in game mode and setting mode are detailed in Table 1 and Table 2, respectively.

Table 1: Gestures for game mode

Action	Attribute	Figure
Turn to setting mode	Dynamic	Fig. 6a
Click	Static	Fig. 5a
Flag	Dynamic	Fig. 6c
Reset	Dynamic	Fig. 6d
Up, Down, Left, Right	Static	Fig. 5b, 5c, 5d, 5e

Table 2: Gestures for setting mode

Action	Attribute	Figure
Turn to game mode	Dynamic	Fig. 6b
Zoom in, out	Dynamic	Fig. 6e, 6f
Rotation clockwise, counterclockwise	Dynamic	Fig. 6g, 6h
Translation up, down, left, right	Static	Fig. 5b, 5c, 5d, 5e

3.1.2 Static Gestures

The implementation of static gesture recognition relies on the functions detailed below. Figure 5 illustrates the schematic diagram depicting the various types of static gestures that our implementation can recognize.

- `tap()`: Verify if the distance between two knuckles is less than the threshold value.
- `direction()`
  - `get_deg()`: Return the directional angle of the vector formed between two points.
  - `check_dir()`: Verify if the angle is within the acceptable margin of error.

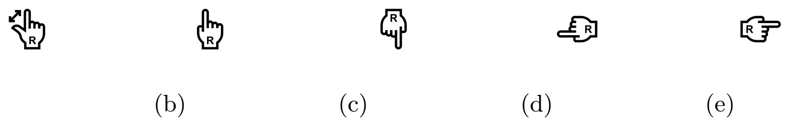

(a) (b) (c) (d) (e)

Fig. 5: Static gestures

3.1.3 Dynamic Gestures

The implementation of dynamic gesture recognition relies on the functions detailed below. Figure 6 illustrates the schematic diagram depicting the various types of dynamic gestures that our implementation can recognize.

- `stable()`: Verify if the coordinates of the trajectory are stable.
- `increase()/decrease()`: Verify if the coordinates of the trajectory are increasing / decreasing and if the distance between the starting and ending points is more than the threshold value.

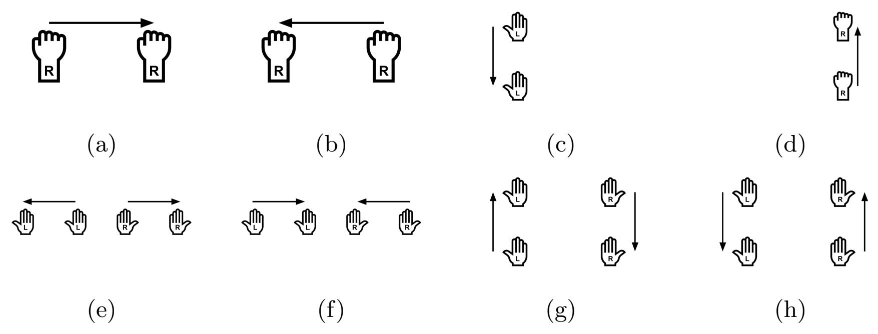


Fig. 6: Dynamic gestures

3.1.4 Corrections and Improvements

Determining handedness can be susceptible to misjudgment

- **Solution**: When a single hand is present on the screen, we maintain a continuous record of its handedness across multiple frames. With each frame, we verify whether the handedness in that specific frame aligns with the recorded handedness. If there's a mismatch, we skip that particular frame. This process ensures consistency by filtering out frames where the detected handedness doesn't match the previously recorded handedness.

Accuracy experiences a notable decline when the user's hand is positioned at the screen's edge

- **Solution**: In each frame, we check whether all 21 hand knuckles are visible on the screen. If any of these knuckles are not detected, we skip that frame. This approach ensures that frames lacking the visibility of any of the 21 hand knuckles are excluded from further processing.

3.2 Image Warping

The image warping pipeline is shown in Figure 7, which is described as follows: First, we calibrate our camera using the OpenCV library and obtain the camera matrix and distortion coefficients, then we use the fiducial markers to estimate

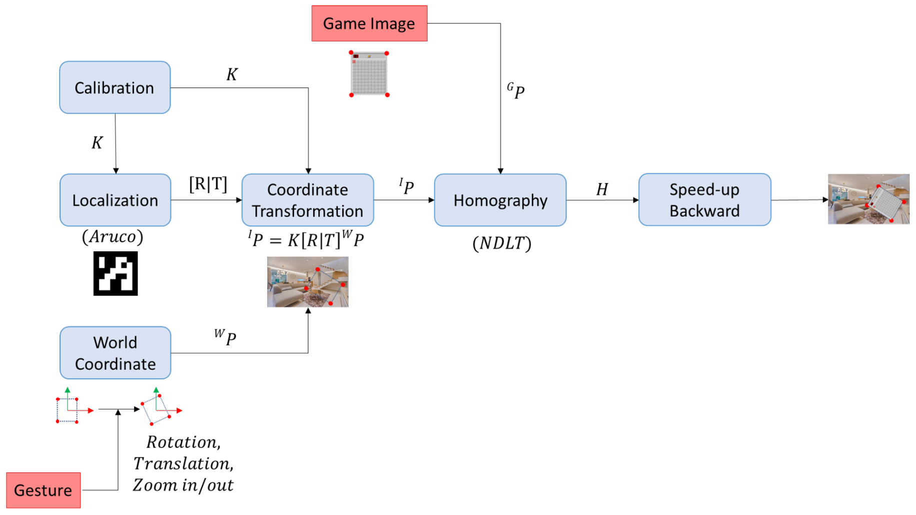


Fig. 7: Pipeline of image warping

the camera pose. Here, we chose the ArUco marker as our fiducial marker. It will return the rotation matrix and translation vector of the camera. On the other hand, we receive the gestures command recognized by the gesture recognition module and perform rotation, translation, and scaling on the world coordinates of the game image. Next, we can use the camera intrinsic matrix and extrinsic matrix to project the world coordinates to the image plane coordinates. Once we have the image plane coordinates, we utilize the four corners and the four corners of the original game image to find the homography matrix. Finally, we use the homography matrix to warp the original game image into the image plane coordinates.

A detailed explanation of the camera localization and back-warping method will be introduced in the next two sections.

3.2.1 Camera Localization

The ArUco marker we use is shown in Figure 8, which is composed of 6x6 grids, and the first and last rows and columns are black. The origin of the world coordinate is the center of the marker and point 0 is the upper left corner of the marker. points 1, 2, and 3 are the upper right, lower right, and lower left corners of the marker, respectively. So far, we have four points in the world coordinate system and four points in the image coordinate system. As a result, we can solve PnP problem to estimate the camera pose.

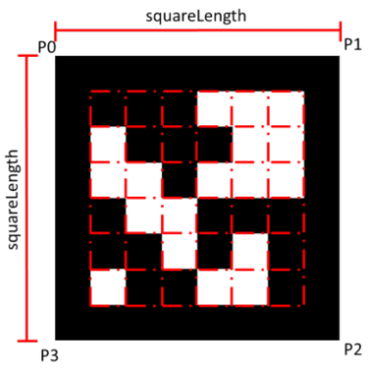


Fig. 8: Aruco marker

3.2.2 Backward warping

The backward warping is pretty time-consuming. we speed up the process by utilizing multiprocessing module in Python. The speed-up backward warping process is shown in Figure 9. The warping process is described as follows:

1. Extract the range of interest(ROI) image.
2. Divide the range of interest image(ROI) into several subimages.
3. For each subimage, executing the backward warping process paralleledly.
4. Merge the subimages into the final image.

We use 8 CPUs and divide the image into 8 subimages. The experiment results show that this method is 10 times faster than the original one.

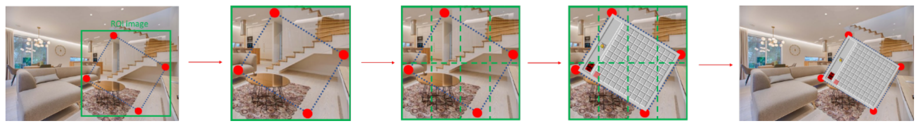


Fig. 9: Speed-up backward warping

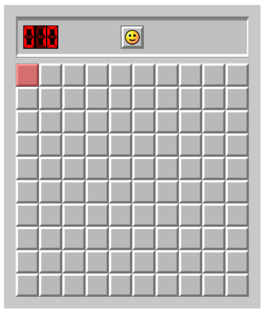
3.3 Game Board

In this project, we decided to implement the Minesweeper. The main idea behind of our design is create the game board as an image and control it using gestures. Consequently, we can update the game board by modifying an image.

3.3.1 Initialization

What we can set

1. The size of a square
2. The number of rows and columns
3. The number of mines.


(a) 10×10 with square size=40

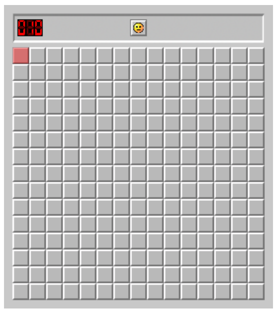

(b) 15×15 with square size=30

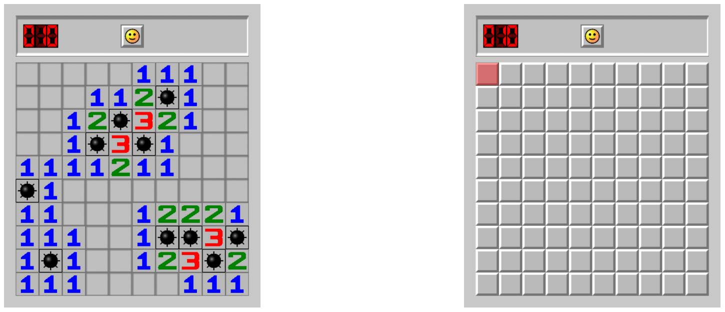
Fig. 10: Different size of game boards

Steps

1. Randomly spread the given number of mines in the board.
2. Calculate the number of adjacent squares containing mines for each square. 8-connectivity rule is adopted in the Minesweeper.
   – The number of a square represents the number of neighboring squares containing mines.
   – A square without a number indicates that there is no mines around it.
3. Answer board (Fig. 11a) is used to record the status of each square. It can not be seen by the player and it is immutable during the game.
4. Displayed board (Fig. 11b) is used to display to the player, so it is mutable during the game.
   – A red square indicates the current position.

3.3.2 Setting Mode

View the game board as an image, and just perform homography transformations, such as translation, rotation and scaling.



(a) Answer board (b) Displayed board

Fig. 11: Answer board and displayed board

3.3.3 Game Mode: Pipeline

The pipeline diagram is illustrated in Figure 12. The steps are listed below.

1. **Input:** a recognized gesture.
2. **Movement:** Use the function `select_cell()` to determine the square that the user chooses to go, the valid movements include up, down, left, right, staying still (none) and reset button.
3. **Action:** Use the function `perform_action()` to recognize the player's action, the possible options include setting a flag, clicking on the square and doing nothing (none).
4. **Output:** Update the game view by modifying the image.

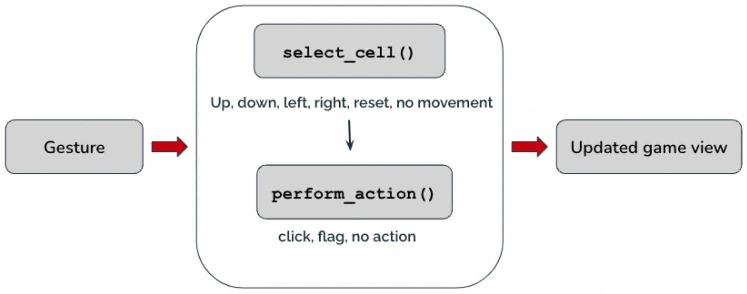
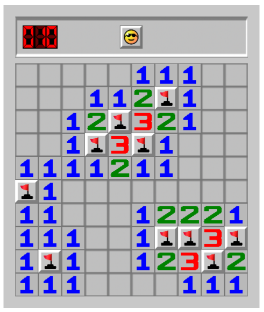


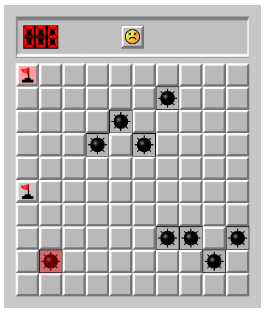
Fig. 12: The pipeline in game mode

3.3.4 Game Logic

Objective

1. Win the game by clicking on all the squares without mines. (Fig. 13a)
   - A flag indicates that there is a mine.
   - It is noted that "putting a flag" does not imply "clicking."
2. Lose the game when clicking on a square containing a mine. (Fig. 13b)
   - A red flag represents that the user has wrongly put a flag on a square without a mine.
   - A flag represents that there is indeed a mine.


(a) Win the game


(b) Lose the game

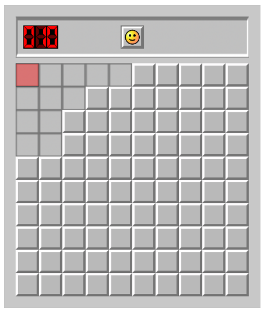

(c) Connected squares

Fig. 13: Game views

Click

We can only click on an unopened square. When clicking on a square.

1. **With a number**: Just open it.
2. **Without a number**: Open its connected squares without numbers as well. (Fig. 13c)
   - Implement by running BFS (Breadth First Search).
3. **Containing a mine**: Lose the game. (Fig. 13b)

Flag

We can only put a flag on an unopened square without a flag. A flag can help us keep notes.

1. **Remove a flag**: Just perform the action "put a flag" on the square again.
2. **The number of flags that we have used**: We can derive the value from the score board located in the top bar.
   - (It equals the number of mines – the number showed in the score board).

Reset

Move to the face in the top bar to reset the displayed game board; however the answer board remains the same.

4 Result

Fig. 14 depicts the scenario captured in the demo video. For additional demo footage, please follow the link provided below.

- **Setting mode**: We showcase the performance of image warping alongside the gesture recognition functionality within the setting mode. Please take a look at the video available at `https://youtu.be/bCG_e6DM8Q0` for reference.
- **Game mode**: We showcase the performance of the game board alongside the gesture recognition functionality within the game mode.
  - **Win**: We demo the game logic required to win the game. Please take a look at the video available at `https://youtu.be/cHwvG6RrSas` for reference.
  - **Lose**: We demo the game logic required to lose the game. Please take a look at the video available at `https://youtu.be/scPI9iFOk0s` for reference.

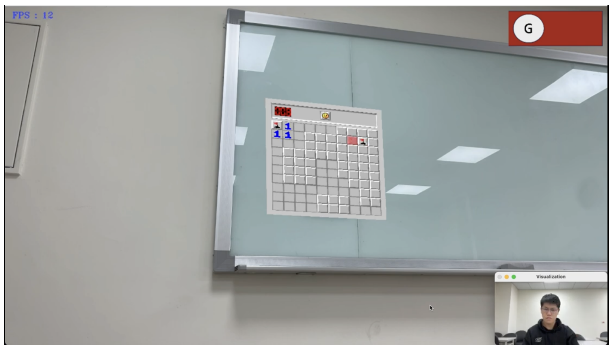


Fig. 14: Demo

5 Conclusion

In conclusion, our project presents an innovative emulation of the Apple Vision Pro's functionality, achieved at a significantly lower cost. Through the clever integration of commonplace devices—a cellphone for AR glasses simulation and a computer for visual display—our team successfully implemented sophisticated features including gesture recognition, image warping, and an engaging Minesweeper game. Our demonstration underscores a more accessible avenue for immersive entertainment, showcasing a cost-effective alternative to the Apple Vision Pro. This amalgamation of gesture recognition, image manipulation, and a fully functional gaming interface not only highlights the potential for affordable AR experiences but also addresses the growing demand for accessible and engaging entertainment mediums. It stands as a testament to the prospects of bridging the divide for users seeking immersive engagements in a cost-efficient manner.

6 Work Contributions

- **Camera localization:** Ching-Hsiang Wu
- **Gesture recognition**: Jung-Hao Kao
- **Game board implementation**: Ju-Hsuan Weng

References

1. Introducing Apple Vision Pro - YouTube, `https://youtu.be/TX9qSaGXFyg`. Last accessed 24 Dec 2023
2. Hand landmarks detection guide — MediaPipe — Google for Developers, `https://developers.google.com/mediapipe/solutions/vision/hand_landmarker`. Last accessed 24 Dec 2023