Project Report : GHOST-HUNTER

Andreas CHARALAMBIDES
Carl ABOU SAADA NUJAIM
Abel HENRY-LAPASSAT

I) Introduction

'Ghost Hunter' emerges as a nearly innovative gaming application, uniquely designed for both wearable devices and smartphones. This immersive game plunges players into an unseen environment filled with roaming ghosts, challenging them to capture these entities within a critical three-minute timeframe.

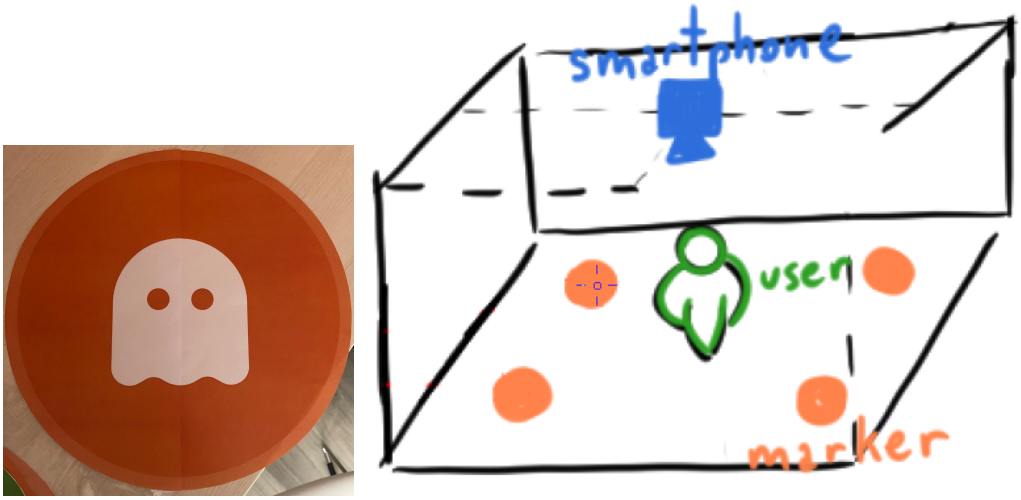

Fig. 1: photos of a markers (left), sketch of scene disposition (right)]

I) Objective and Gameplay

The game's primary goal is for users to navigate an invisible, obstacle-ridden environment in pursuit of ghosts. Players must catch all the apparitions before time runs out, using only their smart device's feedback for navigation. As the players move, they encounter invisible barriers that impede their path. The device alerts them to these obstacles through escalating vibrations: a soft pulsation when they are near, intensifying as they get closer. Players must attempt to 'break' these barriers by interacting with them; success deactivates the vibration, while failure or bypassing results in the loss of a life. The game adopts similar mechanics for the play area boundaries, with the crucial exception that these perimeters are unbreakable, necessitating player avoidance.

Catching the ghosts involves a synergy of UI prompts, 3D audio cues, and vibration alerts. When a ghost is within reach, the device notifies the player, who must then quickly press a designated 'capture' button.

II) Innovative Gaming Experience

Distinct from traditional video games that depend heavily on visual feedback, 'Ghost Hunter' invites players into a novel experience. Here, the game world is navigated and interpreted through essential sensory feedback relayed via the player's device. The initial setup involves the player laying markers on the ground, utilizing their smartphone as an overhead room camera (see Fig. 1). This preparation transports players into an enthralling world where they must capture ghosts. Players need to be acutely aware of their virtual surroundings, which include obstacles and room boundaries. Additionally, players are required to wear a green marker to ensure accurate tracking throughout the game.

III) Technical Foundation and Development Insights

This project employs the Unity Game Engine primarily but is enhanced with a variety of technologies, including Computer Vision. In this report, we will delve into the details of this project, from its conception and rules to its functionality and technical complexity. We will also discuss the limitations and challenges faced during development, along with potential future improvements.

II) Project Overview

A. Game Rules

The rules for Ghost-Hunter are simple, yet it’s necessary to detail them and explain the functioning of the program (which will be elaborated in the ‘Code Explanation’ section of the document).

Players are equipped with a predefined number of lives and they have to capture a specific quota of ghosts within a limited time frame. Throughout the

game, players will have to rely mostly on their non-visual senses to acknowledge the virtual environment, notably by listening to sounds emitted by ghosts when detected by the player, or the vibrations produced by the wearable device whenever the player gets close to or collides with an obstacle (Fig. 2).

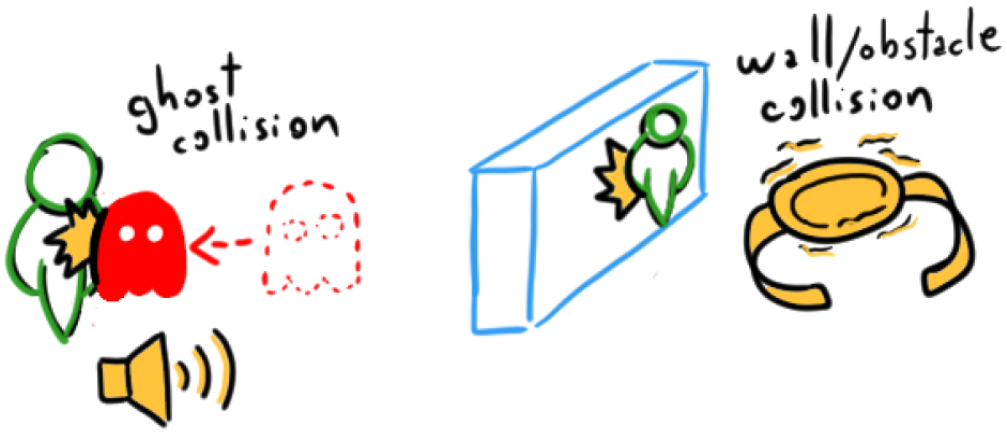

Fig. 2: Sketches of sound & vibrating cases

The only visual feedback is the interface, which lights up whenever a ghost passes through the player's flashlight, and displays only the basic information (time left, lives left and ghosts caught) (Fig. 3).

Fig. 3: Interface changes on ghost detection

While navigating the game, the player will confront various virtual obstacles, which will emit a series of vibrations when the user gets close to them. The closer the user gets, the faster the vibration rate is. These obstacles can be destroyed by executing a punching gesture (more or less permissive), except for the structural walls of the room (Fig. 4). Furthermore, contact with these obstacles will cause the loss of one of the player's lifes (Fig. 5). When they run out of lives, the player loses the game.

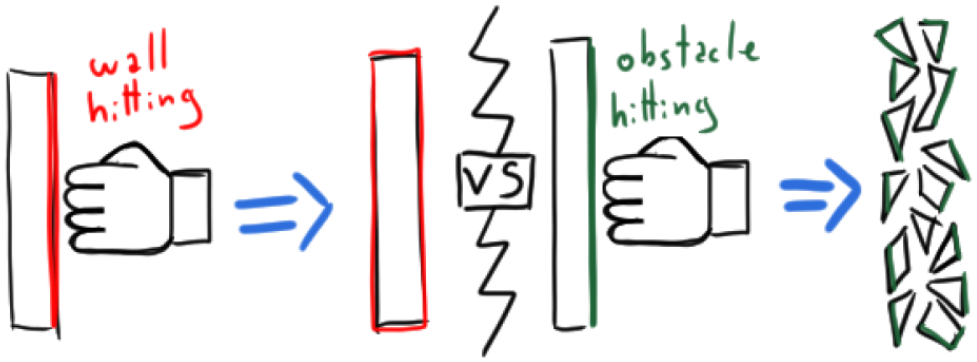

Fig. 4: Punching Wall & Obstacles behavior

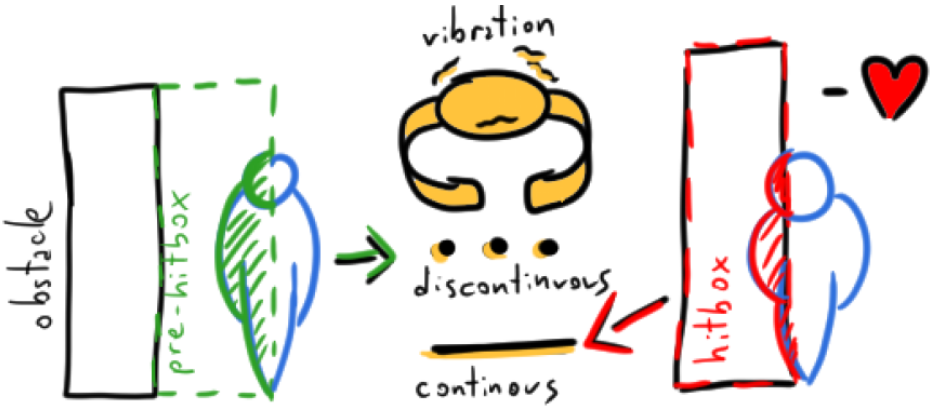

Fig. 5: Wall/Obstacle Collision Behavior

B. Global Functioning

In order to make all of this possible without showing anything to the user, we had to replicate a vibrant virtual world within the Unity application, which serves as the stage for every element of the gameplay. Inside of it, player movement and actions are carefully replicated through multiple scripts and linkings, which make use of all kinds of inputs to produce the corresponding behaviors in either game evolution, and/or haptic/audio/visual output (Fig. 6).

Moreover this virtual environment is used to simulate various elements and behaviors, such as the flashlight following the wearable device's orientation, the ghosts and their behavior (random movement, fleeing away when detected, production of sounds, etc), and the player's actions and their results (hitting obstacles, detecting ghosts, destroying impediments, etc).

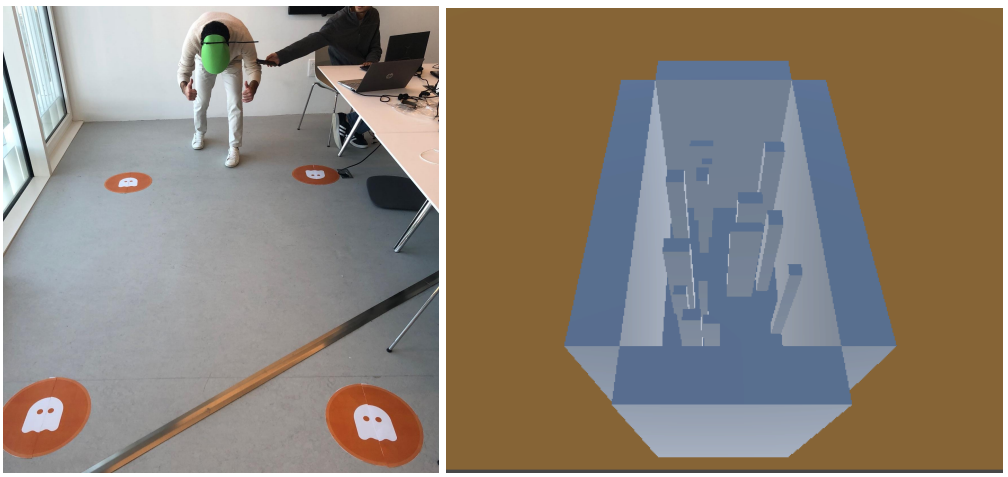

Fig. 6: Real Environment (left) & virtual environment in Unity

To enable player movement, we used computer vision, computed in the overhead camera through python scripts and external software (IVCam), in order to track the player's real-world position and map it to the virtual scene synchronously. Computer vision also helps us detect markers in order to both setup the room, compute its dimensions, and track the player's position regarding their distance from the top-left makrer (Fig. 7). Such technologies helped us give this project an innovative approach for player immersion, blurring boundaries between reality and virtuality.

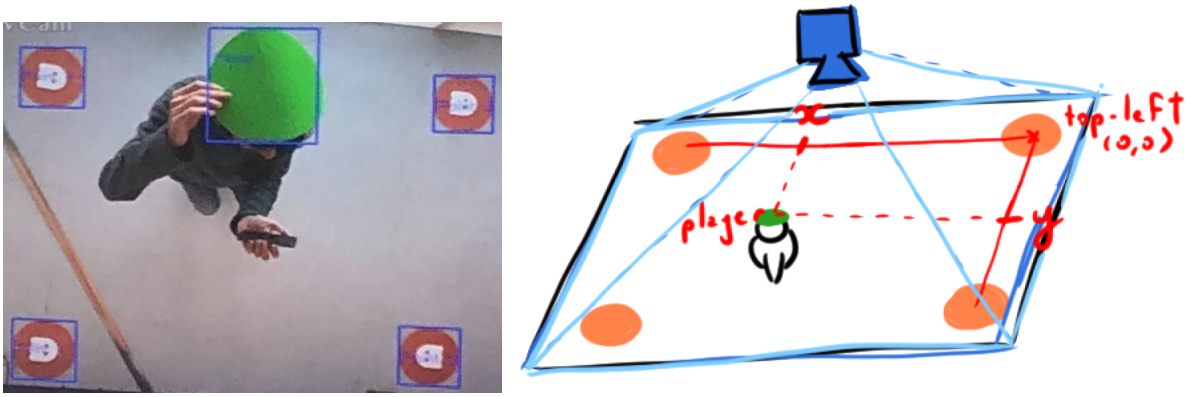

Fig. 7: Marker Recognition & Player Tracking(left), Player positioning in Scene

C. Setup - (from real to virtual environment)

As previously described, configuring the game necessitates the user to position the four provided markers in a rectangular shape, forming the optimum layout for a basic room. Additionally, they must wear a green marker on their head to enable tracking of their position by the overhead camera.

During this setup phase, user can observe the overhead camera's view on a computer screen, allowing for adjustments if necessary. The detected

markers are also displayed on the watch's interface, turning green when recognized (Fig. 8). When the four markers are correctly detected, a signal is sent to the game, which is now ready to start.

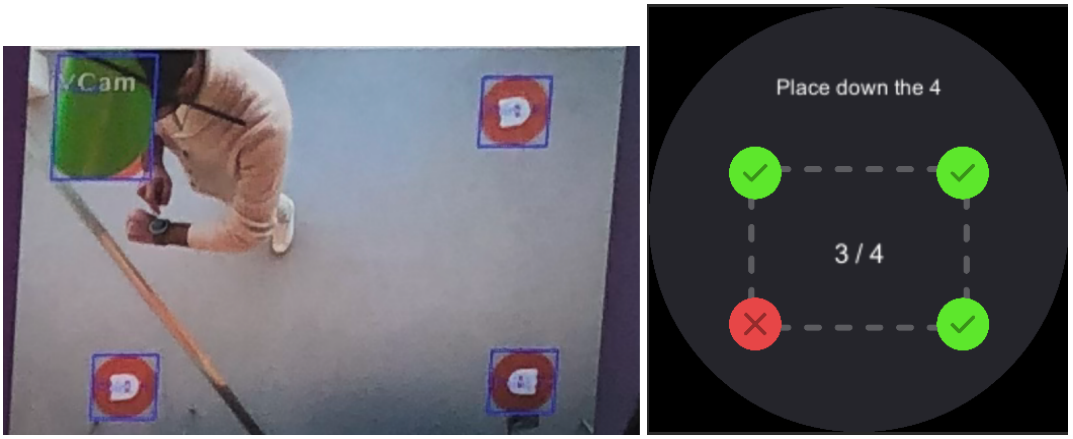

Fig. 8: Marker Recognition during placement (left), Visual feedback on Watch UI

Upon successful marker recognition, the virtual room initializes using the dimensions defined by the markers' positioning. Obstacles are then randomly positioned & sized, taking into account the room's defined size. Then, ghosts are distributed and start their movement, while the player's representation is set up in the virtual world.

Once this initialization process concludes, the game is set up and ready to begin. In the subsequent section, we will delve into the implemented interactions, enumerate the tools & devices used, and outline project's feature.

III) Project Features

A. Interactions

This project encompasses a diverse range of interactions, all easily performable by the player:

- ***Moving Around:*** While in motion in the real environment, the player can trigger specific events such as detecting a ghost, which results in a haptic and auditory response, or colliding with an obstacle, causing vibrations from the wearable device coupled to an auditory signal if the player is losing a life.
- ***Rotating Watch/Smartphone :*** By rotating their wearable device, the user is rotating the simulation of the flashlight in the virtual scene, which might trigger a ghost capture, lighting up the interface and generating a sound effect.
- ***Clicking :*** User can click on the interface's button when it lightens up in order to effectively capture the detected ghost. The latter will trigger a sound effect and vibration (different if succeeded capture or not).

- ***Gesture Input :*** In order to destroy obstacles, the player needs to perform a punching gesture, which will possibly destroy the wall, modifying the virtual environment, and making 2 noises, one for wall hitting and the other for wall destruction.

B. Devices & Tools

From what has already been discussed, we can already list multiple devices that have been used in this project. First, there is the player's wearable device, a Galaxy Watch5 Pro that has been lent to us by Mehdi & Anastasia. Second, we have the overhead camera, which, in our case, was a phone (could be another device, but this was the most effective and readily-available in this timespan). Third, this project also made use of a computer to link the watch and the phone, and process the phone's captured data (fig10).

The watch is the main (and only) game interface. It is used to manage the player's flashlight orientation, listen for gestural inputs, output corresponding feedback, and display the game interface. To get and use the orientation of the device, we accessed its gyroscope's values and used it as a base for the flashlight's direction in the scene. Also, the watch is the only device that is executing the main code, thanks to a Unity application built from the Game Engine, to make an Android Standalone Software.

The phone is only used as a camera, it is not executing any script. It is linked to the computer using a software named "IVCam," to capture the camera output and send it to the device to perform the necessary computations.

The laptop receives the phone inputs (using IVCam), and executes computer vision scripts to analyse the phone's captured images in real-time and send the information to the watch's Unity application.

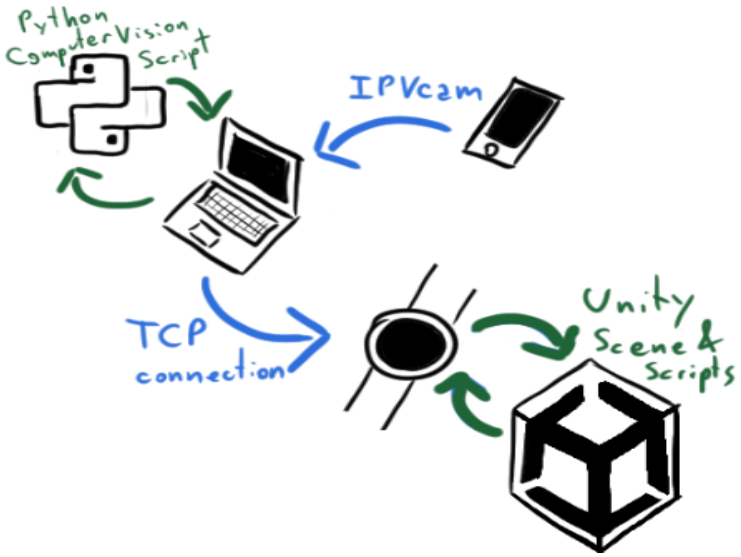

Fig. 10: Project Devices & Technologies Plan

IV) Coding Explanations -

A. Andreas

One of the challenges we faced in our project was tracking the user's position. We initially thought we could achieve this by using the user's smartphone device's accelerometer, gyroscope, and various mathematical and physical theories. However, this method proved to be inefficient and unreliable. Given the limited timeframe, we had to find a more effective approach as the initial one was far too time-consuming.

After a discussion with Mr. Mehdi Chakhchoukh, we decided to adopt Computer Vision to track the user's position. Our refined strategy involved placing a camera in a fixed position above the designated "playground", providing a top-down view. This setup would be instrumental in capturing the entire scene below, which included both markers and the player, thus allowing for comprehensive tracking and interaction within the space.

The logic behind the Computer Vision implementation is as follows: To calculate the user's available playground area, the user must place four physical markers in each corner of the desired play area. These markers are uniform and distinct, easily recognizable by the system. Each marker's position is identified using color detection algorithms, leveraging the HSV color space, which is less sensitive to lighting changes compared to the traditional RGB space.

Once the markers are placed, the user initiates the system, and the camera starts capturing real-time video data. The system processes each frame to locate the markers based on the predetermined color profiles, ignoring irrelevant objects and potential interferences. Advanced image processing techniques, such as Gaussian blurring, are employed to reduce image noise and improve the markers' detection accuracy.

The detection algorithm generates contours around the areas of interest (markers) and calculates their centers. These centers are pivotal as they help determine the corners of the user's playground. By categorizing these centers as 'top-left,' 'top-right,' 'bottom-left,' and 'bottom-right,' the system establishes a virtual playground boundary corresponding to the physical space marked by the user.

The real challenge emerges in calculating the user's position within this predefined area. The user's coordinates are captured similarly to the markers, but with additional computations. The system continuously captures the user's position,

calculates the distance from each corner, and normalizes these values to a scale of 0 to 1. This normalization process is crucial, as it makes the system adaptable to different playground sizes and camera resolutions.

Moreover, to measure real-world distances between markers and maintain proportionate scaling, the system incorporates a predefined real-world width of the markers. It calculates a scale factor by comparing the real-world width with the perceived width in the camera. This method allows for the accurate determination of distances between various points in the space, essential for maintaining the user's correct positional context within the virtual environment.

To ensure smooth and real-time communication, the system employs TCP networking principles. The processed positional data, including predefined boundary adjustments, are transmitted to the Unity game engine. This transmission allows the virtual environment to adjust and respond to the user's movements accurately, enhancing the immersive experience.

Adopting Computer Vision simplified our process, reduced the potential error margin, and provided a more dynamic and adaptable solution. It allowed for quicker iterations, making it possible to test and refine our approach effectively within the project's constrained timeline.

Useful Resources Used:

- Send Data From Python To Unity TCP Socket - YouTube
- Welcome to OpenCV-Python Tutorials's documentation! — OpenCV-Python Tutorials beta documentation (opencv24-python-tutorials.readthedocs.io)
- OpenCV: OpenCV modules
- How to Normalize NumPy Arrays (Min-Max Scaling, Z-Score, L2) • datagy
- How to detect specific colors from an image using OpenCV? (projectpro.io)
- Color-Based Object Detection with OpenCV and Python | Don't Repeat Yourself (dontrepeatyourself.org)

B. Carl

1. TCP Connection

To make use of the TCP connection implemented by Andreas and process the received data, a delegate was created, which is invoked on every message received.

```
//Delegate to be fired when message received
public delegate void Delegate(string message);
public static Delegate onReceiveMessage = _ => { };
```

Any script that requires an input from the Computer Vision application can thus subscribe a method to the delegate, which will be called every time a new message is received. The corresponding methods can thus each check if the received message concerns them, and process it if it does.

However, as the received message is handled by a separate thread and the delegate updates the UI at times, Andreas created a queue and, every time a message is received, pushed a function that invokes the delegate.

```
// Enqueue the received message to handle in the main thread.
mainThreadActions.Enqueue(() => {
   // This action will be executed in the main thread.
   onReceiveMessage.Invoke(dataReceived);
   // For the purpose of acknowledgment or command, send back a
response
     string response = dataReceived; // This can be anything
relevant to your application.
   byte[] sendBytes = Encoding.UTF8.GetBytes(response);
   nwStream.Write(sendBytes, 0, sendBytes.Length);
});
```

Then, he invoked these pushed methods in Update(), to execute them on the main thread.

```
void Update() {
   while (mainThreadActions.TryDequeue(out var action)) {
       action.Invoke();
   }
}
```

2. UI

For the user interface of the game, Unity's legacy UI was used. It was adapted for a small circular display, and all clickable areas are big enough to prevent misses on the tiny watch screen. The implementation of the UI is quite straightforward, and all it does is receive information from other scripts (or input from the user) and update the watch display accordingly.

3. **Room Generation** (Initialization/RoomGenerator.cs)

When the dimensions are received by the watch, the RoomGenerator.cs script will generate the room and obstacles. It will

start by defining the room dimensions in the Unity scene, and setting up the four walls (one on each side of the room).

After that, the room area will be divided into sections of roughly 1.5m by 1.5m, and the size of each section will be computed. Then, inside each section, a single obstacle will be instantiated with a random size. That way, obstacles will be somewhat evenly distributed in the room. Note that, as the scene will not be visible to the user, the obstacles are represented by cubes.

For more information, check out the script, which is fully commented.

4. **Obstacle collision** (Player/HitTesting/ObstacleCollision.cs)

   There are two different levels of colliding with walls/obstacles: getting close to one, which triggers a vibration, and colliding with one, which will take away one of the player's lives.

   To implement the vibration, I used a script from:
   https://gist.github.com/ruzrobert/d98220a3b7f71ccc90403e041967c46b

   I also created my own vibration handler script that utilizes the above script to provide different methods to generate an appropriate vibration depending on the context.

   When the user gets close to a wall/obstacle, a vibration gets emitted with a repetition rate that reflects the distance between the user and the obstacle. I.e., when the user gets closer, the vibration pattern quickens.

   In order to compute the actual distance between two objects (not the distance between their pivot points), I used the following script:
   https://gist.github.com/andrew-raphael-lukasik/658184336c9799ed66f3a5acfa3e7f9c

   The latter will get the bounding boxes of the two given GameObjects, and compute the smallest distance between those two.

   In order to avoid colliding with obstacles, the user can opt to break them instead. Whenever they are inside the trigger area of an obstacle, the user can punch the air (cause the accelerometer values to peak), the obstacle will be destroyed.

   For more information, check out the script, which is fully commented.

5. **Ghost Detection** (Player/HitTesting/GhostDetection.cs)

   The player model in the scene is equipped with a flashlight (an invisible cone with a trigger on it), which is oriented using the gyroscope values of the device.

   When a ghost enters the flashlight trigger area, its state will change to "Detected," it will stop moving, and wait a second. If the user presses down the vacuum button on their device for a few microseconds, the ghost will be sucked in and captured. Otherwise, if the second passes, the ghost will flee and move away from the player.

6. **Audio**

   To spatialize the audio output (which can be sensed using appropriate headphones/earbuds), on top of making all audio sources in the scene 3D, we used Unity's Native Audio Plugins SDK:
   https://github.com/Unity-Technologies/NativeAudioPlugins
   More precisely, we used the script SpatializerUserParams.cs from the SDK.

C. Abel

1. **Ghost Movement**

   In order to make the game more fun and realistic we had to make the ghosts move, and for that I used a simple 'State Machine' program which I've put in the *Ghosts.cs* script which is attached to the ghost prefab. Plus this is using the 'FIxedUpdate' method from unity because when it comes to movement and physics handling, it is better than the 'Update' method, cause time is linearized in the Fixed one, so there are less errors due to time evolution.

   The state machine logic is quite simple, the ghost is in either [Move], [Turn] or [Wait] state :

   - In [Move] state, the ghost will move forward at a given speed, and that using a counter, that decreases of 1 for each movement call. Once this counter has reached 0, then we randomly assign the state to either [Wait] or [Turn] with a higher probability to be [Turn]. Then if [Turn] state is chosen, we select a random new value for the moving counter. And then we set a random value for the turn angle value.
   - When in the [Turn] state, the ghost will perform a rotation (predefined value) but not at once, I used a counter to try "smoothing" it a bit, but it's not working very well. So basically the ghosts only rotate to the given angle. And right after finishing its turn, the ghost will go back to [Move] state.

- And for [Wait] state, the ghost will simply stops moving and turning, so will do nothing for a given amount of time (randomly choosed). Before going back to [Move] state.

The *Ghost.cs* script also handles the case of a Wall collision, and if so, the ghost will be assigned an opposite direction from the one he actually has, and be forced to enter [Turn] state, and to force him to turn in this direction, we don't set the turn counter so it's triggering the other part of [Turn] state, which is the one telling the ghost to turn in the given 'opposite direction'.

This script also handles the case where Ghosts enters in collision with the player, which triggers the 'ghost sound' to be played. Plus it contains the ghost's detection function which makes him flee away when detected but not captured quickly enough.

2. Game Logic & Linking

Even tho the game logic is kinda implemented in every script, as many scripts are handling their part of the game, the main behaviors are defined in the *GameHandler.cs* script, but also it's linking as much stuff as possible together using centralized functions (not for everything because due to bad group communications we couldn't centralize everything…).

So the main use of this script is to make sure that every component is well set, basically it fetches all of the setup value (ghost amount, time for the game), but also it instantiates the ghosts (creating them, and placing them at the center of the room before telling them to start moving and behaving as the *Ghost.cs* script tells em), and starts the timer using an IEnumerator function which will be running throughout the program runs, and triggering the *EndGame* function which makes the End UI pop and stops the game at the same time.

In order to properly starts the game, I also created the *GameStarter.cs* script, which is just here to store all the setup values we'll need for the game to be instantiated correctly (time, player lives, ghost amount, ghosts to catch amount). Basically I wanted this script to handle more stuff (like the room instantiation too) but again Carl didn't communicate properly on what he had done in his implementation so I couldn't use my script to do so as he'd already implement it.

3. Flashlight Rotation

As the watch is used to scan the room for ghosts, we needed to reproduce the watch's rotation to apply it to the "flashlight" of the player inside of the virtual scene. In order to do so I initially attached *PlayerRotation.cs* script to the player, but this one has been merged with the *PlayerMovement.cs* script.

The main logic of this script lies in the *Update* function, which is executed on every frame. In this, the gyroscope datas are used to update the rotation of the attached game object, applying the raw gyroscope rotation while smoothing it to ensure a fluid movement for the flashlight.

In the *Start* method, I do the initialization, activating the device's gyroscope, setting the target frame to 60, and records the initial Y-Axis angle for reference. Then I create a new "GyroRaw" GameObject to attach the script and after that I initiates the calibration process using the *CalibrateYAngle* Coroutine.

This coroutine serves as said to calibrate the rotation angle on the Y-Axis. This temporarily increases the smoothing factor for the calibration, calculates the calibrated angle, and finally restores the original smoothing factor.

Now that we've initialized the setup, we call inside of the *Update* method the followings functions on each frame :

- *ApplyGyroRota* which handles the 'raw_gyro_rota' value based on the gyroscope datas, also performing some adjustments to ensure that data is correctly oriented.
- *ApplyCalib* simply applies the previously calibrated Y-Axis angle to the 'raw_gyro_rota' variable.

In summary, this script combines gyroscope datas from the device gyroscope sensor with some calibrations and smoothing techniques to control the rotation of the attached GameObject -The Flashlight-.

4. Player & Ghost Prefabs

As we've developed this game mainly under Unity, we used a well-known feature of this Engine, the *Prefabs*, in order to declare customed GameObjects that we would use later on, allowing us to declare their behaviors, components as wanted. All prefabs are available in the *Assets/Resources/Prefabs* folder

The ones I created are the *Ghost* & *Player* prefabs :

- *GHOST* - Body is represented by a Sphere, on which I attached a collider & a rigidbody, which are both Unity components designed to allow physical interaction (mainly collisions with other game objects) involving the given GameObject. And obviously the *Ghost.cs* script to make this GameObejct behave as explained earlier, as a ghost. Plus there is an AudioSource that is triggered from the script, with the "GhostWoosh" sound attached to it.
- *PLAYER* - Body is here represented by a tall cube, on which are also attached a RigidBody and a Collider (for the same reasons as the ghost prefab ones). Also the "RotationFlashlight"object is attached to it, in order to make it follows the user's position and

rotations. Moreover, all script that describes and rules the player's behavior are linked to the parent GameObject in order to be sure they are applied to every child object of it (both body and flashlight), and in these we have *PlayerMovement, ObstacleCollision, GhostDetection* scripts.

5. Additional personal works

Throughout the project, we've wanted to use stuff that we've realized zqs either not relevant, not working, or even not performable. Here are the ones I tried :

- GPS tracking for movement ->

```
public class GPS : MonoBehaviour {
public static GPS Instance { get; set; }

   public float latitude;
   public float longitude;

   private void Start(){
       Instance = this;
       DontDestroyOnLoad(gameObject);
       StartCoroutine(StartLocationService());
   }

   private IEnumerator StartLocationService(){
       if(!Input.location.isEnabledByUser){
           Debug.Log("User hasn't enabled GPS");
           yield break;
       }

       Input.location.Start();
       int wait = 20;

       while(Input.location.status ==
LocationServiceStatus.Initializing && wait > 0){
           yield return new WaitForSeconds(1);
           wait--;
       }

       if(wait <= 0){
           Debug.Log("Timed out");
           yield break;
       }

       if(Input.location.status ==
LocationServiceStatus.Failed){
```

```
            Debug.Log("Unable to determine device location");
            yield break;
        }

        latitude = Input.location.lastData.latitude;
        longitude = Input.location.lastData.longitude;
    }

    private IEnumerator UpdateLocationCoroutine(){
        if(!Input.location.isEnabledByUser){
            Debug.Log("User hasn't enabled GPS");
            yield break;
        }

        Input.location.Start();
        int wait = 20;

        while(Input.location.status ==
LocationServiceStatus.Initializing && wait > 0){
            yield return new WaitForSeconds(1);
            wait--;
        }

        if(wait <= 0){
            Debug.Log("Timed out");
            yield break;
        }

        if(Input.location.status ==
LocationServiceStatus.Failed){
            Debug.Log("Unable to determine device location");
            yield break;
        }

        latitude = Input.location.lastData.latitude;
        longitude = Input.location.lastData.longitude;
    }

    public void UpdateLocation(){
        StartCoroutine(UpdateLocationCoroutine());
    }
}
```

(I won't explain this code because it is not inside the final project, but it was meant to catch the device's GPS values and

translate it to the scene, but I figured out by trying it that is wasn't precise enough…)

- Photon Networking ->

I don't have all the codes for this because it was more of a project organization than a single script (RPC functions all over the existing scripts, use of PhotonViews attributes …)

```
public class NetworkHandler : MonoBehaviourPunCallbacks {
      //photon room
   private RoomOptions room_opt;
   private byte max_in_room = 3; //watch app, phone-cam app, +1 to avoid errors
   private int current_in;

   //Custom prefabs
   public GameObject ui_prefab;
   public GameObject cam_prefab;

   //some variables
   private int player_cnt = 2; //watch + phone
   private bool cam_connected = false;
   private PhotonView pv;


   //Unity Methods
   public void Start(){
#if UNITY_STANDALONE_OSX
      Screen.fullScreen = false;
      Screen.SetResolution(800, 600, false);
#endif
      pv = GetComponent<PhotonView>();
      Connect();
   }

   //Callbacks
   public override void OnConnectedToMaster(){
      base.OnConnectedToMaster();
      current_in = 0;
      room_opt = new RoomOptions{
          MaxPlayers=max_in_room,
          IsVisible=true,
          IsOpen=true
      };
      PhotonNetwork.JoinOrCreateRoom("Room", room_opt, TypedLobby.Default);
   }
```

```
   public override void OnJoinedRoom(){
       base.OnJoinedRoom();
       if(PhotonNetwork.IsMasterClient){
           Debug.Log("OnJoinedRoom -> being MASTER, from :
"+PhotonNetwork.LocalPlayer.ActorNumber);
           PhotonNetwork.SetMasterClient(PhotonNetwork.LocalPlayer); //not rly
necessary but a good security
           Debug.Log("Waiting for PARTICIPANT to join");
       } else {
           Debug.Log("OnJoinedRoom -> being PARTICIPANT, from :
"+PhotonNetwork.LocalPlayer.ActorNumber);
       }
   }

   public override void OnPlayerEnteredRoom(Photon.Realtime.Player newPlayer){
       base.OnPlayerEnteredRoom(newPlayer);
       if(PhotonNetwork.IsMasterClient){
           if(PhotonNetwork.CurrentRoom.PlayerCount==player_cnt){
               pv.RPC("LaunchBothRPC", RpcTarget.AllBuffered);
           }
       }
   }

   public void Connect(){
       Debug.LogError("Connecting to server ...");

       PhotonNetwork.NickName = System.DateTime.Now.Ticks.ToString();
       PhotonNetwork.ConnectUsingSettings();
   }

   [PunRPC]
   public void LaunchBothRPC(){
       Debug.Log("Entering LaunchBoth RPC method");
       if(PhotonNetwork.IsMasterClient){
           //here we wanna launch the watch's part (ui then starting game)
           Debug.Log("launching for master -> Watch");
           ui_prefab = Instantiate(ui_prefab, transform.position,
transform.rotation);
       } else {
           //and then we wanna start the camera's functionning (setup &
communication)
           Debug.LogError("Launching for part -> Camera");
           /*Here we wanna set the SETUP RECOGNITION*/
```

```
            cam_prefab = Instantiate(cam_prefab, transform.position,
transform.rotation);
            Debug.LogError("NetworkHandler -> Starting the communication");
            cam_prefab.GetComponent<RoomCamera>().GetMarkers();
        }
    }
}
```

So this scripts uses Photon Unity network library to handle connections between the game, the camera, the setup, and to synchronize all the instantiations & stuff.

Would definitely be irrelevant to comment as we are not using neither this script nor this logic, but I wanted to point out that even tho it seems like I did less than my colleagues, I worked on lot of stuff beside what is currently implemented, and I couldn't finalize lots of ideas because of the poor group communication we had, but even tho I am proud of what I did, and I wanted to show what I developed for this project anyway.

V) Project Discussions

A. Encountered Issues -

a. User's Tracking Techniques

Throughout this project's conception and development, we've come across multiple problems, and we've also realized that our project was encountering some limitations in its actual state.

The first problem we've encountered was about the player's movement. Indeed, the initial idea we had to track the player's movement was to use the accelerometer and gyroscope built in the watch, but it turned out to be infeasible as these tools are not precise enough to generate an at-least-somewhat accurate tracking of the player. Matter of fact, we couldn't even find any source of help on the internet. Here are the different techniques that we tried throughout the game's development:

- GPS, to track the user's position (Fig 11). The latter wasn't possible because the accuracy of the GPS wasn't high enough for the scale we planned to use. Also, adapting our project to this much bigger scale meant too much efforts for the user. In addition, we would have needed an empty room of at least 20 squared meters, which was infeasible.
- Smartphone (or wearable device)'s accelerometer coupled to its gyroscope to detect the relative change in position of the user, as well as their orientation in the real world (Fig 12). However, as we tried several times with various techniques, we realized that it wasn't

possible in this short timespan to implement something good enough and precise to be trustfully used in our game. Nevertheless, the code for detecting the player's orientation was still used in our project.

- The last technique that we tried (and this time succeeded with) was to completely change the tracking idea, and use Computer vision instead to track them from above, thanks to an overhead camera. The player, wearing a green marker, will be recognized by the overhead camera, and the marker's position will be mapped from the image to the unity scene.

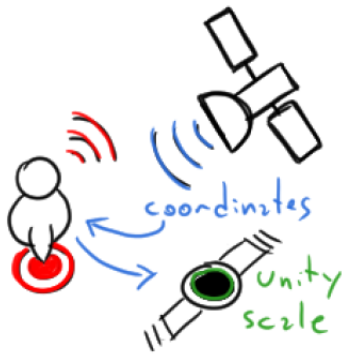

Fig. 11: GPS Tracking Illustration

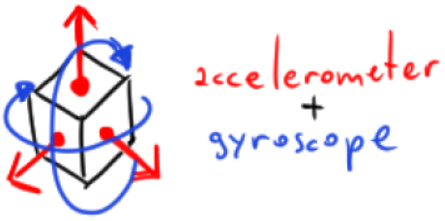

Fig. 12: Sensor Tracking Illustration

b. Project Limitations

Even though our project is running and everything we've implemented is working, it still suffers from some technical limitations. First, and surely the most important one, the size of the playable area (approximately 2mx3m), turned out to be too small. Ideally, we would need an area twice as big for an optimal experience, because, in it's actual state, every game element is a bit too close to others, which lead in too many sounds and events at the same time (ghostsdetected + obstacles triggering vibrations + trying to move), and it can be overwhelming for the user. While the gaming experience isn't the best, all of our implemented features worked as expected.

Another problem would be the setup,which is too complicated. As explained earlier, our setup involves a camera connected to a computer, both linked to the watch, which is executing the main program. Ideally, we would only have two devices, the watch and the overhead camera, but computer vision requires some processing power to be realtime, and the phone's is not enough.

B. Possible Improvements

Obviously, the first improvement we would make involves addressing the limitations, potentially through the use of more sophisticated technologies and devices. We also considered implementing computer vision within Unity, as

this would allow us to rely solely on an overhead camera and a wearable device, eliminating the need for any additional software or devices. However, we were unsure of how to proceed and could not dedicate more time to acquiring these complex skills.

Another enhancement we could develop is the behavior of the ghost. We could employ Machine Learning to teach them to act more naturally or utilize tools such as Unity's NPC Path Finding. However, the latter option might not result in sufficiently random behavior for our ghosts.

Additionally, we could diversify the game features, particularly the variety of obstacles. Currently, obstacles are merely walls of different sizes, placed around the virtual room. Imaginably, we could introduce items like tables and false ceilings, requiring the player to adapt more to the virtual environment and enhancing their immersion.

This innovation leads to another requirement: the ability to track the player's height during the game. This would necessitate refining our computer vision script or exploring other methods, such as using trackers within watches if feasible (perhaps through y-axis accelerometers).