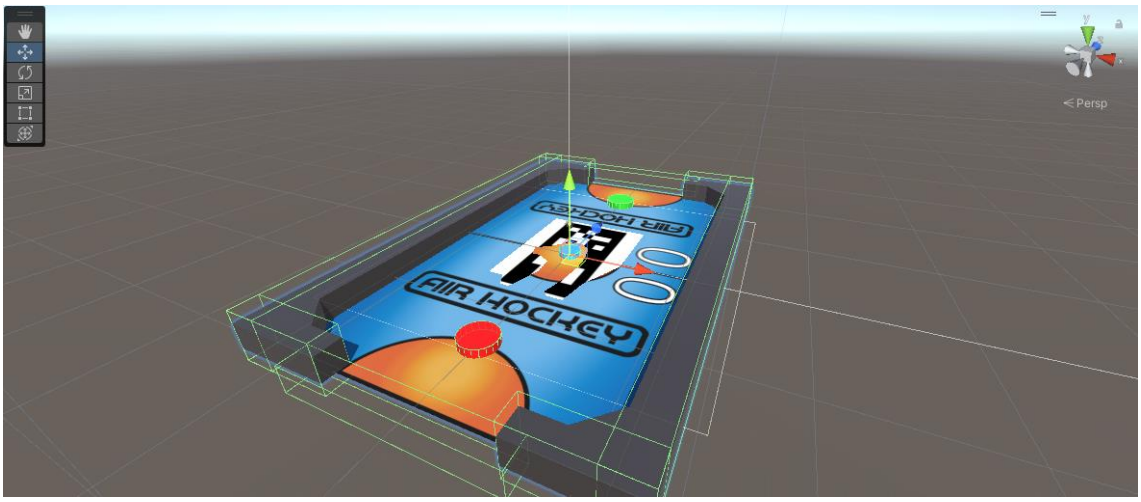
Jaime Cortón
Eduardo Herreros
Sergio Avello

A(I)R HOCKEY



MOVEMENT IMPLEMENTATION

The same approach is used for both singleplayer and multiplayer mode. The idea is sample, the striker moves to the position in which we place our finger on the screen. Whenever we raise the finger, the striker stops. The movement is limited to half court, so we can not cross midcourt to score a goal.

The computation of the velocity of the striker is done through position displacement among frames. Then, each time the "update()" function is called (every frame), we recompute the velocity based on the new and last position of our finger. Is important to point out that we should not pass the position directly, because we would override the position automatically calculated by unity each frame, and then we would encounter problems with the physics of the game.

The implementation can be found in *"touch_screen_singleplayer.cs"* and *"touch_screen.cs"*.

In order to limit the movement of the striker to half court we've placed an invisible wall in midcourt, and we have made used of the different layers, so that the disk does not collide with this wall, but the strikers do. Same goes for the goals.

The velocity of the disk trough the court has been set accordingly so the disk can go fast but still with some deceleration, so the game is fun and easy to play. This has been set altering the properties of the disk's rigidbody and creating a specific physic material.

Obstacles:

First, we considered adding a joystick to the game. This idea was discarded quickly since the game was played at a high pace that could not be achieved through the joystick, thus reducing playability.

Secondly, because of lack of understanding of the unity engine, we committed the mistake enunciated before when implementing the "finger follower" approach. Instead of calculating

the velocity, we modified the position each frame, hence overriding the position calculated by unity. This caused sometimes our player to overlap with the disk, instead of colliding.

GOALS

The goals our simply to invisible blocks that act like triggers. The moment the disk collides with them, a function is triggered to increase the count of the score.

SINGLEPLAYER AI

The AI algorithm for the striker in singlepayer mode is quite simple. Similarly to the movement implementation, we compute the velocity as a position displacement. However, this time the new position is not our finger, but the position of the disk (we target the edge point of the disk, not the centre).

In order to make the AI not perfect we limit the velocity in which the striker approaches the disk. Moreover, we introduce some random x-axis noise, so the hits of the striker vary through the game. This randomness also avoids the disk to get stuck between the wall and the striker.

Obstacles:

In the first drafts, the disk got stuck sometimes within the wall and the AI striker. This was solved adding randomness and not aiming for the centre of the disk.

Difficulty of the game. It was perfect at the beginning, which made it impossible to score. Randomness and decreasing the velocity of the striker from the actually needed solved this problem.

Yet, this AI is a quite simple implementation that could be heavily improved. Moreover, it would be valuable to allow the player in the game to choose the difficulty to play.

This implementation can be found in "*AIScript.cs*".

MULTIPLAYER

The multiplayer game is implemented in Photon, with PUN2. This package allows to build multiplayer games easily and with no economic cost.

In order to connect, we first load the players to the same lobby. Once they are in the lobby, one of the players chooses a name and creates the room, acting as the host. The other player would be the client, and must enter the chosen name by the host to enter the same room.

The players and the disk are photon objects, so its position and velocity are shared between host and client.

The pause and gameover scenes are synchronised between the host and the client, meaning that if one of the players presses pause, the same menu pops up to the other player. If one of the players resumes the game, it also resumes to the other player. The home button is not synchronised.

We use RPC (Remote Procedure Calls) that are method-calls on remote clients to handle some events that we want to take part in both host and client or sometimes in just one of them. For instance, the Pause and Gameover menus are set by these calls. Moreover, we take advantage of *"photonView.IsMine"* to modify the goals during the game (otherwise, the score would be increased by the host and the client, and each goal would count as 2).

Obstacles:

Pun is clearly not made for games that demand fast and frequent physical interaction. In our case, we have a disk that it's initially instantiated by the host, whose position and velocity information is shared to the client. This means that we may reach the situation in which the position of the client has been updated and then it receives the information that the disk is in the same place, leading to overlapping. In order to solve this problem we would need to manually code the client movement, such that is the HOST the one in charge of its movement and therefore responsible of sending the information of its position to the client. Hence, the client will send the information of the new desired position to the host, the host would process this information and handle the physics (collision or not with the disk), and depending on this physics the host would send the new position to the client. This would make visible some delay in the client (that could be handled with some interpolation) but would solve any physics problems.

A second approach would be to directly switch to Photon Fusion, which is more complex but more suitable for games with this physical demands. Fusion has been successfully used to develop many multiplayer FPS games. Due to the lack of time and its price however we did not try to dive into this approach.

SCREENS AND MENUS

The game counts with multiple scenes (Home, Lobby, Game…). All this scenes are loaded as a result of a button pressed or something achieved (as last goal scored or lobby/game successfully loaded).

It is possible to go back to the home screen from each of the scenes of the game, so there's no need to close the app under any circumstance.

Jaime Cortón
Eduardo Herreros
Sergio Avello

PAUSE

When the user presses pause, the game freezes. This is achieved by setting the timescale to 0. Then when the game is resumed, the timescale is reset to 1.

Obstacles:

When we set the timescale to 0, the communication between the host and the client is interrupted. Therefore, if we were in the pause menu and one player pressed resume, nothing would change for the other player because the information would never arrive. In order to keep alive the channel while timescale is 0, we need to add the following line "*PhotonNetwork.MinimalTimeScaleToDispatchInFixedUpdate = 0;*".

SOUNDS

The collision of the disk with the walls and the strikers has a sound assigned (collisions handled by tags in the gameobjects), and also with the goals (function called by gameobject behaving as trigger).

COURT THEMES and DISKs

The court counts with 4 different textures that can be changed throughout the game, in the pause menu. The disk gameobject can also be chosen between a blue disk or a donut. The position and velocity of the former disk is preserved.

In multiplayer mode, we need to remove the photon disk that we had previously and instantiate the new one. In the other hand, the courts are independent of the multiplayer connection, which means that each player can load a different theme without worrying about the connection.

Obstacles:

Removing and instantiating the disk photon object every time we want to change the disk in multiplayer is most probably not the most efficient way to do it. However, we cannot just load the texture as we did with the courts because we are dealing with prefabs (the donut has different shape and rigidbody than the blue disk).

Moreover, we needed to be especially careful with the gameobject that we removed before loading the new one. If we deleted the gameobject as usual, the host or the client would end up with two disks, and so on. Therefore, we need to delete the Photon gameobject, as so both, the disk of the host and the client get deleted.

MARKER

The table is loaded with a marker (used Vuforia for this).