Mini Golf Game in Unity

Andrei BUNEA, Xiyao LI

March 28, 2023

1 Introduction

Our game is designed for multiple players to build their own track and play golf together via the Ubiq platform. At the start, players join a room using the Ubiq panel to connect with other players on different machines. First, they need to build the golf track and place obstacles on it. Players are provided with one panel of track patches and one panel of obstacles. They can click on a patch on the panel to spawn it and link it to the current track. They can also add obstacles to the track in the same way. After editing the golf track, players can take turns playing golf. We have a scoreboard to keep track of the current putt count. After one round, we reset the count to zero and put the golf ball back at the start of the track. The source code can be found in our GitHub repository [1].

The following tasks have been addressed in our implementation:

- Physics of the ball, club, and obstacles
- Networking items between different players
- Snapping track patches together
- Complete track design for demonstration
- Start and end game management

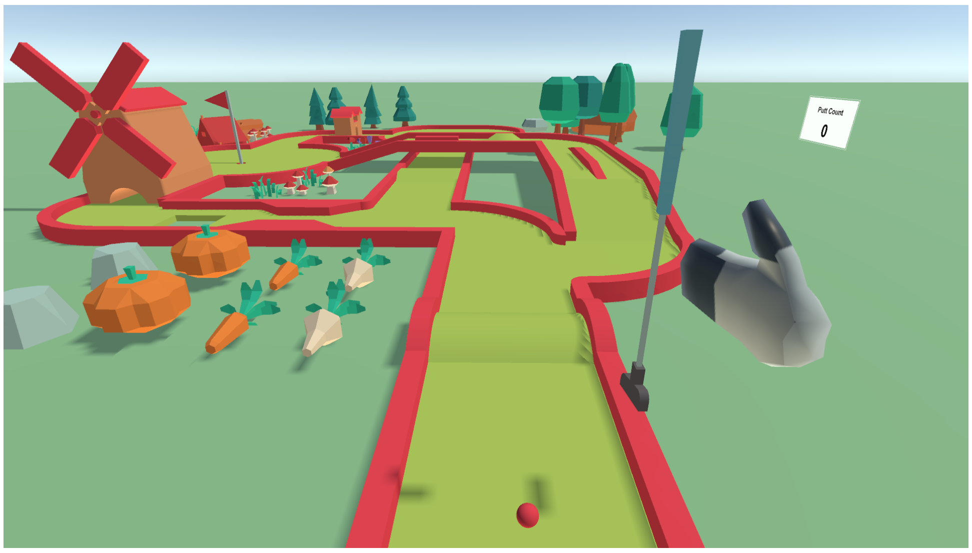

Figure 1: Start scene

[1]https://github.com/Lixiyao-meow/Unity_Mini_Golf_Game

2 Approach to the Problem

2.1 Physics Simulation

Collision Our aim is to model the collision dynamics involving multiple objects, including the ball, club, track, and obstacles. Since Unity has a physics engine designed to simulate real-world physics, our physics interactions are based on it. We attach a collider for collision detection and a rigid body for force simulation. The collider is chosen based on the shape and function of the object. We choose a sphere collider for the ball, a capsule collider for the club, and a mesh collider for the track.

Graspable We make the ball, club, and obstacles graspable by implementing the `GraspBehaviour` class. When the player grabs a graspable object with a `Hand` controller, we enable its `isKinematic` property so that it will not interact with any force but just follow the controller's position and rotation. After being released, we set the object's velocity to the controller's velocity and disable its `isKinematic` property, so that it follows the laws of real-world physics.

One subtle change has been applied to the golf club. When the player grabs the club at the start of the track, the default grab behaviour is always applied to one fixed position on the club with one fixed orientation as shown in the Figure 2. However, holding the club downward while the hand is upward is not a natural posture for playing golf. Therefore, we made two changes to make the graspable behaviour more natural:

1. Rotate the club 180° while grasping, so the player can play with their hand downward as shown in the Figure 3.
   To achieve this, we create an empty object and link a rigid body and the `ClubController` script to it. We then add the club to this empty object for visual effect and collision detection. After rotating the empty object 180° along the x-axis, and rotating the visual club 180° back, the visual doesn't change after these rotations. However, when the player grabs the club, it is oriented upwards following the hand. This allows the player to play with their hand downward, as shown in the Figure 3.

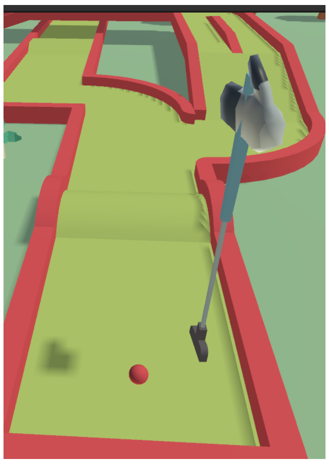
Figure 2: Default grab behaviour

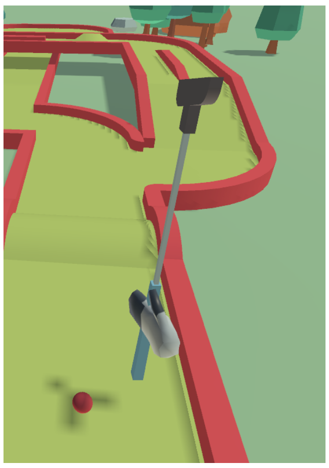
Figure 3: Grab after rotation

2. To ensure accurate and realistic gameplay, we implement grabbing at the contact position between the player's hand and the graspable object

To achieve this, we add a positional transform to the original `Grasp` function, as shown in the code below. When a controller is detected, we compute the relative position between the controller and the visual club position, and translate the club to the contact point. As shown in the Figure 4, the club will remain at the point of hand contact.

```
public Transform visual;

public override void Grasp(Hand controller){
    base.Grasp(controller);
    Vector3 relativePosition = transform.position
                             - controller.transform.position;
    visual.position += relativePosition;
}
```

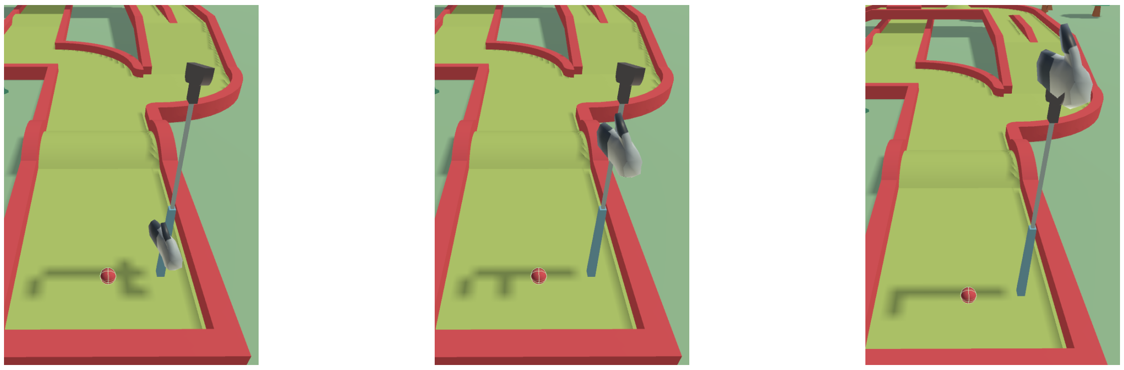
Figure 4: Grabbing at the contact position

2.2 Track Editing

The game allows users to customise their track through an interface that offers two panels containing objects to be spawned. One panel is designed to present and spawn a variety of obstacles that users can place on their tracks, while the other panel generates patches of the track that can be placed to form a functional golf course by snapping the patches together like puzzle pieces.
When we developed these two functionalities, we decided that for demonstration purposes it would be better to have a pre-existing golf course in the game. As a result, we created two different scenes: one where users can only add obstacles (the Minigolf scene) and one where users can build their own track by snapping patches together (the Snapping scene).

How are the objects spawned? To handle object spawning, both scenes contain a `Spawner` component with a `Spawner` script. The script contains the types of prefabs that can be generated, and the method `Spawn()` handles the generation of new prefabs given an ID as a parameter. Each panel has clickable images for every object, and each image has an `On Click()` method in the Inspector that calls the `Spawner.Spawn()` method with the unique ID of the object to be spawned. Every time a new object is created, it spawns at a fixed position.

Particular differences in Snapping scene First, the grasping behaviour is slightly different. The patches generated using the panel are assigned the `PatchGraspBehaviour` script,

which differs from the `GraspBehaviour` script in that the kinematics do not change when grasping and releasing.
The second difference is the `SnappingScript`. It is added to all the track patches made from prefabs. To help the user create their own track, a snapping, magnetic-like behaviour was implemented since these objects are very sensitive to any rotation or movement of the controller. At every frame, all pairs of two `SnappingScript` objects are observed to determine if they are in snapping distance. This operation is handled by an empty component called `SnapManager`. If two objects are within snapping distance, one of the pieces is rotated and repositioned so that it matches the other component.

Very important It is crucial that the patches are generated one by one. Only one player should generate a single patch using the panel, which is then fixed in the track. After that, a new patch can be generated. This behaviour is necessary because once another patch is generated, the previous patch loses its snappability. It would be difficult and sensitive to place the previous patch perfectly again. Snappability is lost because only one component should be attracted to others at any time. If more than one is attracted to others, all objects within snapping distance of a moving patch would follow it and create a domino effect, ruining the whole track. Therefore, when grabbing a patch and moving it alongside the track to snap it in the desired location, some patches will try to snap to the grabbed patch and create disorganised behaviour.

2.3 Networking

The main components that need to be added to the network are obstacles, patches, ball, club, panels, and scoreboard. Networking is handled by adding the `GraspBehaviour` component to graspable objects, except for the patches that use the `PatchGraspBehaviour` script. The panels and scoreboard have their networking components inside their own scripts: `ScoreBoard` and `Spawner`.
For graspable objects, every time a user grabs an object, they become the owner of that object. Ownership does not change when releasing the object, but only when another player grabs the respective object. In the networking process, only the owner of the object sends the new location to avoid an oscillating motion caused by updating the position from more than one user.
A particular case is the ownership of the ball. To emulate the behaviour of golf, the owner of the ball is always the same user who is also the owner of the club. In addition, the scoreboard is updated inside the `ScoreBoard` component by broadcasting the new score when the local score increases. The spawning of objects is broadcasted using messages inside the `Spawner` component. When a button is clicked and an object needs to be spawned, the component broadcasts the ID of the object to be spawned to the network. When the network receives the message, it will spawn the same object locally.

2.4 Game Design

The game design includes the start and end of the game. At the start of a new game, the putt counter is set to zero. When a collision is detected between the ball and the club, the putt count is incremented by one. To end a round, we have a trigger in the golf hole. If the ball remains in the hole for more than two seconds, the round is considered finished. We then reset the putt counter to zero and move the ball back to its initial position.

Out of Bounds To prevent players from hitting the ball out of bounds, we place a plane under the golf track and tag it with the label *Out of Bounds.* When a putt collision is detected, we first save the last position of the ball and perform the putt action. If the ball falls out of bounds, a collision between the ball and the plane is detected, and we move the ball back to its last position.

2.5 Work Distribution

- Andrei: Track Editing, Networking
- Xiyao: Physics Simulation, Game Design

3 Fixed Bugs

3.1 Ownership

Since the ownership messages are asynchronous, it's possible for ownership messages to be overridden. For example, I could become the owner of an object, but while I'm broadcasting the message that I am the new owner, another user could also send a message claiming ownership. In this scenario, both users would consider themselves not to be the owner. To solve this issue, we added a timestamp requirement before ownership can change. If I gain ownership of an object, another user can only gain ownership after a two-second delay. We consider this delay to be natural since the process of gaining and losing ownership involves grabbing and releasing an object, which we estimate takes at least two seconds to complete.

3.2 Collision Detection

We have observed that during a single putt action between the ball and the club, multiple collisions can be detected. To ensure accurate putt counting, we have implemented a cool-down period of 1 second. If a collision is detected within this period, subsequent collisions will not be counted towards the putt count.
We have also observed that some collisions are detected when putting the club near the ball without actually making contact. After investigating the detected collisions, we found that some collisions have a force of zero, which we call "ghost collisions". We suspect that this issue is related to Unity's physics engine. To address this problem, we have set a threshold of `Mathf.Epsilon`. If the force of the detected collision is lower than this threshold, it is not counted as a proper collision.