Project Nautilus
Programmer's Report

Connected Games Development Assignment

GIORGOS KARAMBASIS RODRIGUEZ (K2376268), SLEIMAN GHANEM (K2403892)

GROUP 1

08/05/2024

Contents

1. Introduction to game Implementation

The following report is a development report of the game Nautilus. Nautilus is a multiplayer, team-based battle royale that was made using the Unity game engine. It is an 8-player game, where players are divided into 4 teams and have to navigate an underwater cave in a submarine. They need to search and collect loot, fight off enemy submarine teams, and by all means necessary find the only extraction item available and to escape the caves. Eight players will submerge only a 2-player submarine will manage to extract and survive.

Players controlling each submarine will be able to move around it and interact with its many components. The submarines include a 5-slot diegetic inventory system, gun stations that can be upgraded to provide different firing modes and a pilot station to allow for rotation and translation of the submarine. All features are multiplayer-ready, with Server-Client interaction and ownership control through the use of Photon PUN 2.

Figure 1: Screenshot of Submarines Battling in the game

Each player will also have to register an account on a remotely hosted SQL Database that is accessed through the game's menus. The database keeps track of player currency as well as upgrades they can pick up and extract in the game. Every time a team extracts with loot, it is saved in their account and the database. The next time they start a game, they will be able to bring their upgrades in, to give them a head start.

Lastly, certain common game implementations were used to improve the game experience and efficiency, such as OOP paradigm programming, Object pooling and Coroutine-based programming.

2. Related Game Implementation Features and Practices

One of the main inspirations for the game was *Barotrauma* (*Figure 2*) by Undertow Games and FakeFish. We took inspiration from the mechanics of this game to guide our feature implementation. While Barotrauma is a PvE game, in contrast to our PvP focus, it was enlightening in its use of the movement of player characters inside a moving vehicle.

Figure 2: Barotrauma (FakeFish, 2019) (Left), Nautilus (our game) (Right)

The Photon framework for Unity was used to create the network and to maintain synchronisation between the players. In particular, PUN 2 (ExitGames, 2018) was used, it streamlines room creation and mitigates any concerns related to latency and connection issues. Great consideration was taken when it came to the amount of information transferred over the network, leaving little to be synchronised other than ship, player and bullet locations and occasional *Remote Procedure Calls* (RPC).

A common practice implemented in multiplayer games is the idea of object pooling, used to lessen instantiations and destructions of objects in the game scene. This concept was implemented by the use of a bullet pool, having the master client hide and enqueue bullets instead of destroying them, effectively recycling them, instead of creating new ones. This practice indubitably improved networking and hardware performance for the game.

Furthermore, version control was used to allow for a streamlined merging of code between programmers. Nautilus uses Plastic SCM as its version control software, which allows for creating multiple branches of the same code in the repository. This allowed programmers to work independently of each other without one interfering with the code of the other. This resulted in the assimilation of mechanics at an extremely enhanced rate, especially towards the end of the development with rapid changes occurring frequently. Lastly, its version control capabilities allow for immediate reverting to a previous version of code should any file corruption occur.

3. Analysis of the Implementation

3.1 Database & Lobby System

The first step of implementation begins in the main menu. All players are required to register and consequently log in to play the game. Upon registering, they define a username and a password, that is hashed and salted for security purposes, and they then have ownership to an account that keeps track of their in-game currency and submarine weapon upgrades.

The data is saved in a remote SQL Database that is called upon using PHP using the UnityWebRequest method. A call is made to the PHP code and data is saved or returned from the database. The remote database is hosted through 000webhost, which allowed for a free option to host a database online.

Once registered, players can create a lobby through Photon's Room methods. In the lobby (*Figure 3*), every player's nickname is visible via the Player Listing menu, and when they select a team, their name will be updated via PunRPC calls to inform what team each player belongs to.

Figure 3: Screenshot of the Lobby Menu

This is where players can select to bring any weapon upgrades they have saved on their account to the game. They are presented with three different weapon types and upon choosing one and starting the game, the upgrade is removed from their account via a query to the database.

3.2 Menu to Game Transition

All teams and upgrades chosen in the Lobby are saved into a Photon Hashtable, which is then uploaded via Player.CustomProperties. This way the information is transferred across scenes when the Master Client starts the game. All the players were then sorted into 2-index arrays, in which the first player is responsible for instantiating the team submarine and their teammate's photonviews, and the second player looks for their player photonview and transfers it to themselves.

3.3 Synchronization

Synchronization is a fundamental aspect of any multiplayer, and in Nautilus, synchronizing was established through two methods that were particularly efficient and practical. The main effort that came from synchronization was the players' movements. As this was a client-server-based model, the practical way of synchronizing was to allow the players to have full control and ownership of the characters that they should control. The main flaw with this method is that a player requires the ability to move the character, as well as the submarine and the turrets that will fire the bullets. As such, certain adjustments had to be made so that whenever a player seeks to move the submarine or one of the turrets, they are given ownership of the specific photonviews, or else the movement of the ship/turrets will not be synchronized with the systems of the other players.

```
// Update is called once per frame
Unity Message | 0 references | Changed locally
void Update()
{
    if (playerIsInHelm) //if the player is controlling the ship
    {
        if (Input.GetKeyDown(KeyCode.F) && counter == 0) //if the counter is 0 and the player presses space
        {
            counter++;
            ship.GetPhotonView().TransferOwnership(player1);
            Debug.Log(player1);
            Debug.Log(ship.GetPhotonView().Owner);
            myCamera.transform.position = new Vector3(ship.transform.position.x, ship.transform.position.y, myCamera.transform.position.z);
            myCamera.orthographicSize = shipCameraSize;
            myCamera.transform.parent = ship.transform;
        }
        else if (counter == 1 && ship.GetPhotonView().IsMine) //if the counter is 1 and the player owns the photon view
        {
            square.GetComponent<PlayerController>().enabled = false;
            ship.GetComponent<ShipController>().enabled = true;

            if (Input.GetKeyDown(KeyCode.F))
            {

                counter--;
                ship.GetComponent<ShipController>().enabled = false;
                square.GetComponent<PlayerController>().enabled = true;
                myCamera.transform.position = new Vector3(square.transform.position.x, square.transform.position.y, myCamera.transform.position.z);
                myCamera.orthographicSize= playerCameraSize;
                myCamera.transform.parent = square.transform;
            }
        }
    }
}
```

Figure 4: Code for swapping between controlling the submarine and the character, found in the SwitchController Class

To resolve this, two key concepts were used. First, once the character enters the location where the steering wheel is located, only then will the player be allowed to press the button that will allow for the switch of control. Furthermore, to prevent another player within the same submarine from pressing the transfer button and receiving control as well, a counter was implemented so that only the player that is in the ship control switch game object was able to move the ship, making it so that even if another player pressed the switch button, only the player who owns the ship will be able to switch control and move it. Similar logic is applied to the turrets to allow for the control and movement of the guns.

3.4 Optimization

After the movement of all characters, ships and turrets were synchronized via photon transform views, priority came minimizing any network-related discrepancies with the missiles being fired by the turrets. To begin with, repeatedly re-instantiating bullets with their photon transform views was considered too computationally expensive, and as a result, it was decided that bullet object pooling was necessary to decrease what would be relayed over the network. To synchronize this, the most efficient solution was to create a singular bullet pool that would only be created by the master client.

```
[PunRPC]
0 references | Changed by jorgekaramba@gmail.com on Friday, May 3, 2024
public void RequestBulletFromMaster(Vector3 position, Quaternion rotation)
{
    // Only the Master Client should handle bullet requests
    if (!PhotonNetwork.IsMasterClient) return;

    // Get a bullet from the bullet pool
    GameObject bullet = BulletPool.Instance.GetBullet();

    // If a bullet was successfully retrieved from the pool
    if (bullet != null)
    {
        // Set the bullet's position and rotation
        bullet.transform.position = position;
        bullet.transform.rotation = rotation;

        // Activate the bullet
        bullet.SetActive(true);

        if (gameObject.GetComponent<AudioSource>() != null)
        {
            gameObject.GetComponent<AudioSource>().Play();
        }
        if(gunBarrelVFX != null)
        {
            gunBarrelVFX.Play(withChildren: true);
        }

        StartCoroutine(ActivateRPCAfterDelay(bullet));

        // Synchronize the bullet's state with all clients
        photonView.RPC("SyncBulletState", RpcTarget.Others, bullet.GetPhotonView().ViewID, bullet.transform.position, bullet.transform.rotation);
    }
}
```

Figure 5: Code for Requesting bullets from the bulletpool

By making the master client the only one with a certain amount of reusable bullets, synchronizing among all clients by using RPC calls that go through to the client first would allow for simple synchronicity, as well as for reduced network traffic overall.

3.5 Inventory System & Items

The diegetic inventory system created for the submarine had to be synchronized between players, as more than one player would make use of the same inventory. This was done through PunRPC calls, every time a player chose to pick up, use or discard an inventory item. It was also important to synchronize enemy players' inventories, since when they perished, they would have to drop their loot for the other players to pick up.

Item	Description	Extractable Upgrade
Health Pickup	Heals Player Submarine	
Ammo Pickup	Replenishes Turret Ammo	
Upgrade 1: Single-Shot Gun	Single-Fire weapon upgrade	✓
Upgrade 2: Chain Gun	Faster fire rate upgrade	✓
Upgrade 3: Shotgun	Multiple-projectile upgrade, but slower fire rate	✓
Scrap	In-game currency achieved when extracted	✓
Extraction Beacon	Extraction item that has to be used in the designated Extraction area. Upon collection, the submarine's location will be revealed to all other players. 30-second timer to extract upon use in the extraction area	

Table 1: List of Available Inventory Items

The inventory has 5 slots and can hold a collection of 7 different items, each doing serving its own unique purpose listed in the table above (*Table 1*).

Using the Extraction Beacon within the extraction area beacon triggers the Victory Condition, destroying all other player submarines, and saving all *extractable items* (Table 1) currently in the victor's ship on the players' SQL Database. However when players pick up the extraction beacon they are marked on the map for other players to see (Figure 6).

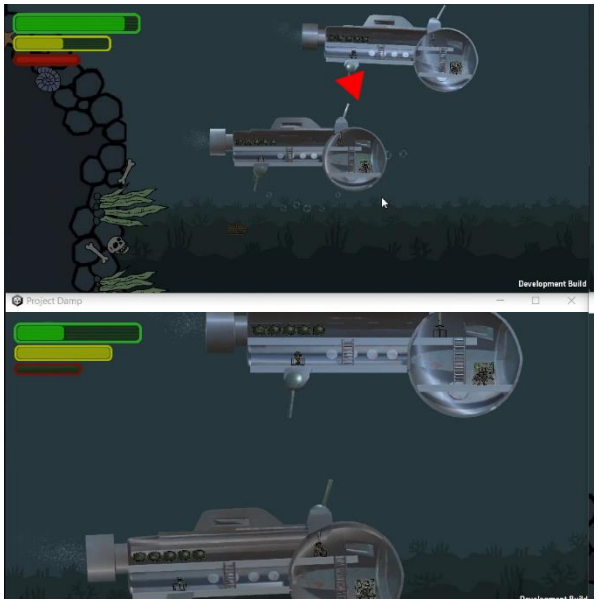

Figure 6: Player lead to Submarine with Extraction beacon by marker (Above), Player in submarine holding the extraction beacon defending their ship (Below)

4. Results and Feedback on the Implementation

There was a few features missing from the prototype that were in the design document, namely the hostile fish AI, minor details such as breakable cave walls for shortcuts and a mini-map for easier navigation. While these features would enhance the game experience, it was decided to leave them out of the prototype to prioritize bug fixing and ensure a more stable game.

Network performance proved to be stable, provided a good internet connection. To further attest to that, all games have ended with the gameplay cycle being fully achieved. There were bugs uncovered, with Master Client transfer, that were diagnosed and promptly solved. They arose from Photon's Room settings, where by default, every player that leaves the scene, destroys all the photonviews they own.

However, the biggest of accumulation of bugs encountered were before or after the game had ended. Before entering a game, there are a few instances where the lobby has problems occurring among its players and its updates. When a player leaves the room through unexpected means, such as turning off the computer or closing the game from the task manager, the game does not detect that a player has left the room.

The most critical of errors can occur from the saving of items, as well as from signing in. The SQL database that is used in Nautilus has an occasional error where the request may time out, and the player needs to request again. This may cause a problem when it comes to saving the inventory items after the game has ended. This occurs because the SQL database is stored on a free web-hosting service, and as such, its performance is unpredictable, and it comes with occasional timeouts. Because of this, people may win the game, but will not get their items saved.

However, no problems were found in the gameplay loop itself. The bullets' and ship's movements were synchronized among all the players. The transfer of ownership was always streamlined and allowed for constant switching between players, and all steps of the loop proceeded smoothly. Master Client transfer as well as the bullet pools returned no errors.

5. Conclusions and Future Work

In conclusion, Nautilus, and its many features and capabilities are multifaceted and highly complex to enable a proper game experience as intended by its designers. It makes use of several gaming principles such as Server-Client interaction, Object Pooling, among others to allow for an efficient and smooth gameplay experience.

The use of PUN 2 allowed for an intuitive network connection and has allowed for a more simplified, personalized experience and network connection. The use of an SQL Database allowed for remote player data storage, regardless of its limitations due to the low-tier server host.

The true difficulties lay in the synchronization of all of its vital moving game objects, and the ability to give the players the right and the ownership to move these vital game objects. The Bullet Object Pool was essential in ensuring that network traffic was not caused by the constant creation and destruction of bullets that needed to be tracked.

Finally, future development plans for Nautilus are described below as a list of additions and improvements:

- A revisit into the networking to account for abrupt disconnections, and improving the SQL database connection and its web hosting service.
- The gameplay loop can be further improved by implementing more submarine stations in the form of a mini-map/radar station, and additional players per submarine to accommodate for this change.
- Player versus Environment (PVE) in the form of hostile Fish AI would be a great addition to the game, providing more resource sinks for players.
- Improved visuals, would allow for an enhanced gameplay experience.
- Lastly, an in-game store, where the players would be able to buy new upgrades that would be implemented into the gameplay would cement the game as a complete work.

6. References

- Exit Games. (2018). Photon Unity Networking 2 (PUN 2) [Software]. Available from: https://www.photonengine.com/pun

- FakeFish. (2019). Barotrauma [Video game]. Undertow Games.

- Moreno, J. (2023). Cartoon FX Remaster Free | VFX Particles | Unity Asset Store. [online] Available at: https://assetstore.unity.com/packages/vfx/particles/cartoon-fx-remaster-free-109565

- Unity Technologies. (2023). *Unity* (Version 2023.1.2f1) [Computer software]. Unity Technologies.

- Yughues, N. (2021). Yughues Free Metal Materials | 2D Metals | Unity Asset Store. [online] Available at: https://assetstore.unity.com/packages/2d/textures-materials/metals/yughues-free-metal-materials-12949