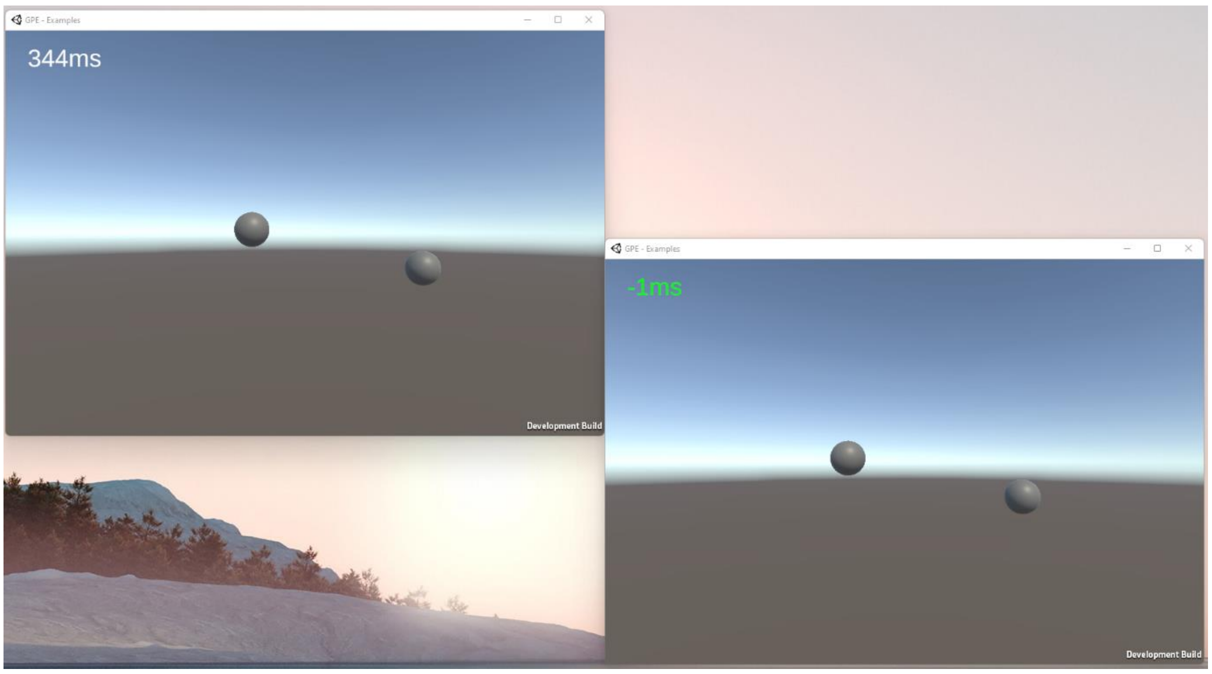
Multiplayer



Date: 08.09.2021
Name: Luca Ruiters
Student nr: 500796991
GitHub: [ru1t3rl/GPE]

Table of Contents

Intro

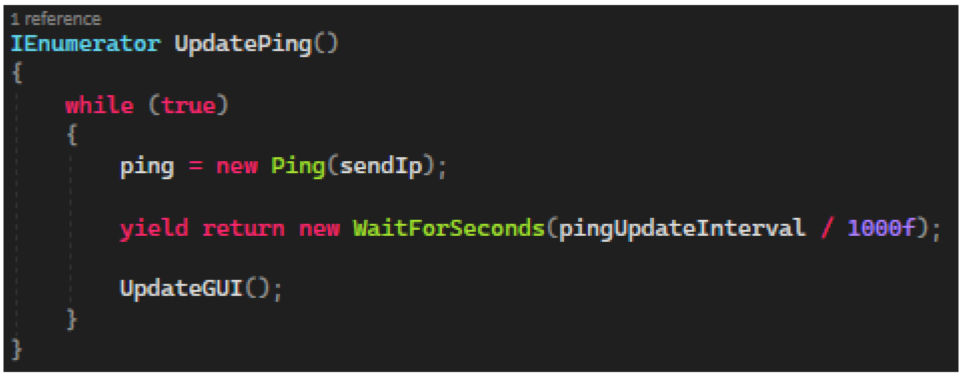
Before I started working in Unity a made a simple priority list. This is what it looked like:

1. A System that checks if the player's connection is too poor.
2. A handshake system
3. Interpolation and Extrapolation
4. Add other objects (e.g., bullets, collision, etc.)

Looking at this list you can see my priority was to make a stable server and when I have achieved that add more objects. Sadly I haven't been able to complete number 4.

Bad Connection

To check if the player has a bad connection we keep updating the user's ping in the background. If his ping gets above a certain level, he will disconnect from the server.



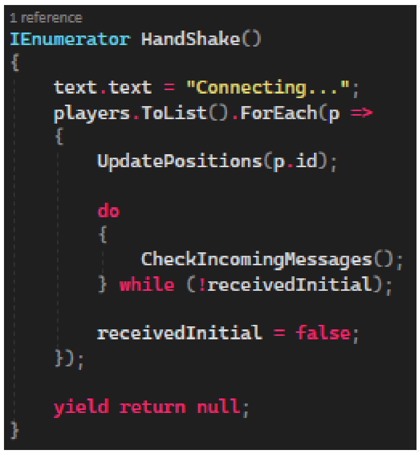
```
1 reference
IEnumerator UpdatePing()
{
    while (true)
    {
        ping = new Ping(sendIp);

        yield return new WaitForSeconds(pingUpdateInterval / 1000f);

        UpdateGUI();
    }
}
```

Handshake

At the start of the application/scene, we execute the HandShake method. This simply updates all player positions and in the CheckInCommingMessages Method also sets an initial delay. This delay will, later on, be used for interpolation and extrapolation.



```
1 reference
IEnumerator HandShake()
{
    text.text = "Connecting...";
    players.ToList().ForEach(p =>
    {
        UpdatePositions(p.id);

        do
        {
            CheckIncomingMessages();
        } while (!receivedInitial);

        receivedInitial = false;
    });

    yield return null;
}
```

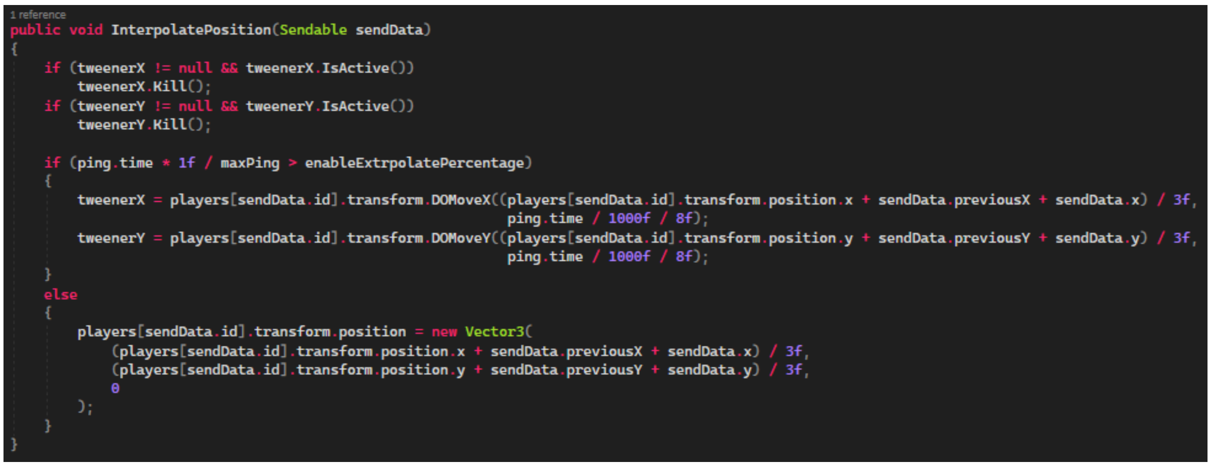
Interpolate & Extrapolate

In the environment, we use interpolation and extrapolation to smoothen the user's experience even if he has a very high ping. While testing I also discovered the interpolation and extrapolation methods combined do hide packet loss a little bit. As long as there aren't tons of packages missing

Interpolate

To make the interpolation even smoother, I interpolate between the user's predicted position, current position and previous position. If the user's ping is bigger than a certain percentage we will use DoTween to make it extra smooth.

At first, using DoTween didn't work very well. They kept bouncing back and forth between the predicted position and its current position. This mainly had to do with the tweener not being finished when it start its next task. To fix this, I saved the tweeners as a variable and check if they are still effect.

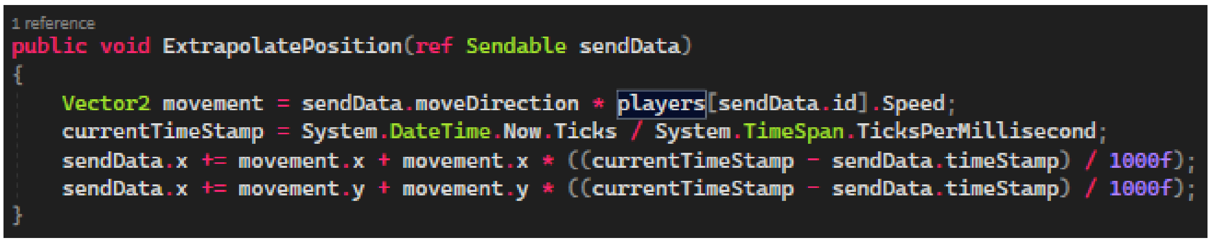


```
1 reference
public void InterpolatePosition(Sendable sendData)
{
    if (tweenerX != null && tweenerX.IsActive())
        tweenerX.Kill();
    if (tweenerY != null && tweenerY.IsActive())
        tweenerY.Kill();

    if (ping.time * 1f / maxPing > enableExtrpolatePercentage)
    {
        tweenerX = players[sendData.id].transform.DOMoveX((players[sendData.id].transform.position.x + sendData.previousX + sendData.x) / 3f,
                                                          ping.time / 1000f / 8f);
        tweenerY = players[sendData.id].transform.DOMoveY((players[sendData.id].transform.position.y + sendData.previousY + sendData.y) / 3f,
                                                          ping.time / 1000f / 8f);
    }
    else
    {
        players[sendData.id].transform.position = new Vector3(
            (players[sendData.id].transform.position.x + sendData.previousX + sendData.x) / 3f,
            (players[sendData.id].transform.position.y + sendData.previousY + sendData.y) / 3f,
            0
        );
    }
}
```

Extrapolate

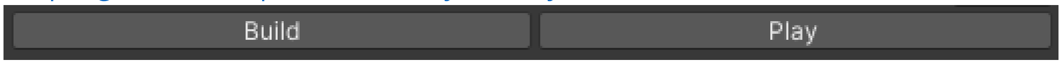
In the extrapolation method, we also perform a small prediction based on the user's previous movement. Which we calculate based on the previous position and its current position. This vector will then be normalized and multiplied by the wanted speed. This will then be used in another formula which will use the delay to perform the extrapolation.

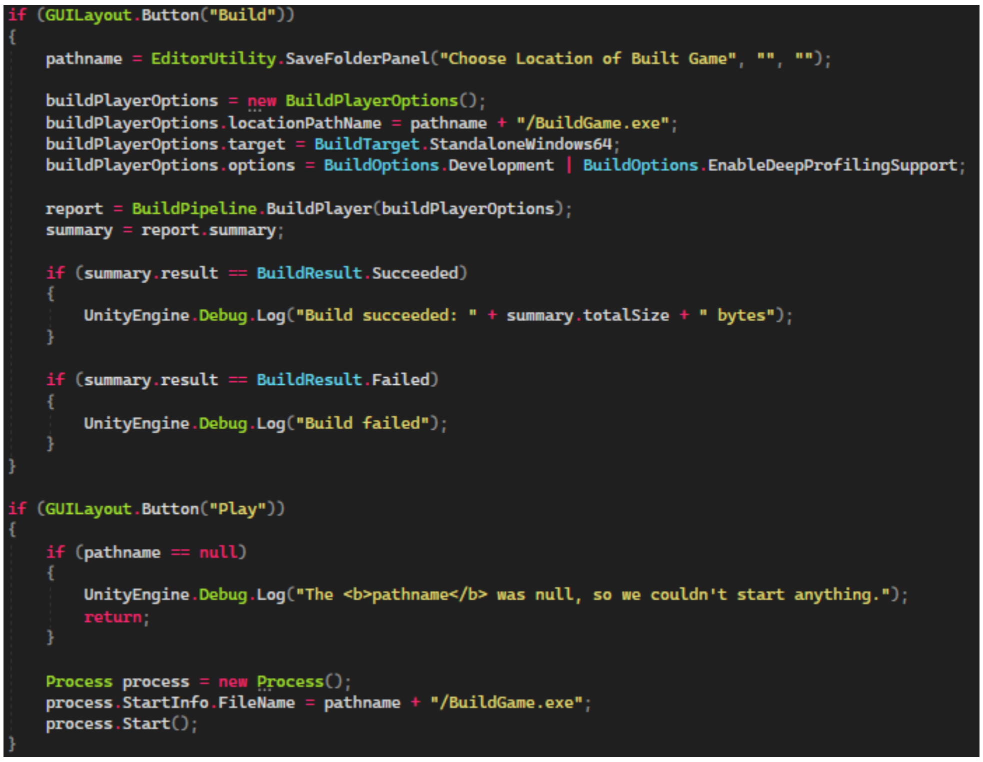


```
1 reference
public void ExtrapolatePosition(ref Sendable sendData)
{
    Vector2 movement = sendData.moveDirection * players[sendData.id].Speed;
    currentTimeStamp = System.DateTime.Now.Ticks / System.TimeSpan.TicksPerMillisecond;
    sendData.x += movement.x + movement.x * ((currentTimeStamp - sendData.timeStamp) / 1000f);
    sendData.x += movement.y + movement.y * ((currentTimeStamp - sendData.timeStamp) / 1000f);
}
```

Extra

To make my life a bit easier, I made a simple custom editor in which I could make a new build and start the build as well. Unity's documentation about the BuildPipeline helped (Unity - Scripting API: BuildPipeline.BuildPlayer (unity3d.com)).





```
if (GUILayout.Button("Build"))
{
    pathname = EditorUtility.SaveFolderPanel("Choose Location of Built Game", "", "");

    buildPlayerOptions = new BuildPlayerOptions();
    buildPlayerOptions.locationPathName = pathname + "/BuildGame.exe";
    buildPlayerOptions.target = BuildTarget.StandaloneWindows64;
    buildPlayerOptions.options = BuildOptions.Development | BuildOptions.EnableDeepProfilingSupport;

    report = BuildPipeline.BuildPlayer(buildPlayerOptions);
    summary = report.summary;

    if (summary.result == BuildResult.Succeeded)
    {
        UnityEngine.Debug.Log("Build succeeded: " + summary.totalSize + " bytes");
    }

    if (summary.result == BuildResult.Failed)
    {
        UnityEngine.Debug.Log("Build failed");
    }
}

if (GUILayout.Button("Play"))
{
    if (pathname == null)
    {
        UnityEngine.Debug.Log("The <b>pathname</b> was null, so we couldn't start anything.");
        return;
    }

    Process process = new Process();
    process.StartInfo.FileName = pathname + "/BuildGame.exe";
    process.Start();
}
```

Conclusion

If I had to describe how it went, I would name it a love-hate relationship. Figuring out how to do certain things was very difficult. But when I found a solution, it turned out to be pretty simple. When I have to continue with multiplayer, I would try to implement a rollback system.