PORTAL II MULTIPLAYER

Authors:

Micael Silva

04/26/2022

Table of Contents

1. First Person Character

1.1. Firing and Spawning Portals

Before we can fire any of the projectiles we have to check that the player is looking at an object that will support those same projectiles when they collide.

So to make this happen we start by drawing a line from the center of the player's camera to his forward vector, if this line collides with an actor we check if the actor has the tag that we specified earlier, in the end we are looking for the PortableWall Tag, if the actor returns this tag we can then fire the desired projectile.

To avoid this check being done every frame we use a timer with a small time interval, giving in the end the same result, but remains more optimized.

If the conditions are met we can then fire the projectile, this projectile will spawn with a default impulse and will check if it hits with an actor that has the PortableWall Tag, if so, it spawns the associated portal. We don't destroy the projectile after spawning the portal because the Portal spawning functions contain a chain of linked behaviors and it will cause a small delay to destroy the portal, as so, we destroy the projectile inside the Portal Spawning function.

1.2. Weapon Inventory

The player has an inventory that is populated by the weapons that he picked up. When he finds a new weapon he will automatically swap from his previous weapon to the new one.

When trying to pick up the same weapon the player will not be able to add it to his inventory since he already has it.

Then we replicate the current weapon that the player is holding to all other clients.

1.3. Weapon Switching

The weapon switching is simply a cycle through an array of the weapons inventory, when pressing Q we will equip the previous weapon, when pressing E we will equip the more recent weapon that has been picked up.

After that we replicate the current weapon that the player is holding to all other clients.

1.4. Health System & Damage

First of all, why did we code our own instead of using the one that is provided by Unreal. Well, we've never liked using things that are given to me without knowing how to make them, this sounds stupid for some people but we believe that by doing our things, we learn way faster, while also, learning with our mistakes. As so, continuing with the health system and damaging the player.

We start by getting the amount of damage to apply on the player’s health system and who applied the damage, then we check if the player who received the damage is not trying to kill himself.

Then we check if our player’s health is enough to keep him alive or if we have to kill him. If we have to keep him alive the game continues and his health is updated, if not, we have to respawn him.

On the player's death we have to clamp our values as well prevent him from taking more damage from future hits. We also made him ragdoll on death so the player who killed him could have some visual feedback. We save our body colors and then we respawn the player by resetting the player Pawn and after that we destroy it. After that we apply the saved cosmetics and we replicate them to all the other players.

2. Portal System

2.1. Portal Self Adjustment

Once we create the portal and receive the information from the projectile, mainly we want to know the location where it collided and the Impact Normal transformed into a Rotator so we can place this portal in the desired location, but before we create the portal we need to replace the old one. Once we replace it we tell you which portal when the player collides, the player should be teleported to that same other portal, so having a Target the player can already know where he is going after colliding with it. Next we check the placement of the newly created portal.

First of all we have to get the vectors associated with the portal we want to adjust and get its limits, these limits are then used to create points where we draw a line to the inside of the object and check if they really collide with something, but for this logic to work we first have to know which object to compare at all its points, to do this we check at the center of the projectile's impact which object it collided with, then we save that object and we are ready to check if at each point the same object exists.

If this object exists in all the points it is not necessary to make adjustments, but if it doesn't exist in the vertical or horizontal points we adjust it according to the point where it doesn't exist. it is important to mention that there is no need to make adjustments from a diagonal point because for a diagonal point to be outside an object, or not find the same object that the center found, because for a diagonal point to be outside then either a horizontal or vertical point will be outside too. so we only check for the really important points, the horizontal and vertical points.

To adjust the position of the portal and the way we do it, we end up using a for loop and for each point we apply a transformation in the desired vector, the problem that followed was to make the adjustments small, and with the same distance. for this to happen when we apply the transformation to the desired vector we multiply it by the current index and divide it by the same giving transformations always with the same distance, whenever we do a transformation we have to re-trace the control lines to know if it is necessary to redo the transformation in the desired direction. Then we repeat this transformation until the object traces the control line and finds in that same point the same object that the center has as reference.

If it then finds the same reference object then we can move on to the next point that needs to be adjusted, if it no longer needs to be updated it moves on to the next, and so on until all the checks are done.

But if it exceeds the maximum number of transformations it means that with the transformations made the portal ended up adjusting more than half of its length, so we destroy the portal to prevent it from having such a large adjustment.

2.2. Traveling Between Portals

To teleport the player we need to know when he is in the portal zone, if he is we start checking if all the conditions are valid for the teleport to happen. If he is not in the portal zone, there is no need to check the conditions.

First we verify that the player is crossing the portal, to do this we create a Plane and we Validate if LastPosition and Point meet their positions with the Plane.

Then we check if the player is at the front of the portal, this condition is validated if the dot product of the created Plane is greater than the point we passed in the function, this point being the player's location relative to the portal. Then, if all the previous conditions are valid, we give permission to the player to be teleported.

To teleport the player, first we start by saving his speed to later apply after teleporting, we also convert the position and rotation of the player relative to the portal then Set the player Location and Rotation when Teleporting to the portal, after that Adjust the player Control rotation in so he faces the same direction and the same angle that he enters in the portal, after that we can set the speed of the new player and reverse the speed to maintain the flow of the player movement since the rotation of the player will be reversed because he enters the portal with one rotation and after being teleported we reverse that rotation.

2.3. Portal Illusions

To make the portal illusion we must convert the Position and Rotation of the player Camera.

As so the SceneCapture location and rotation of that portal’s Target must be updated every frame to maintain the illusion of the portal, just like a “window”, then we should have a dynamic clipping plane so the SceneCapture can hide everything that is between it and the portal, not rendering it in the portal surface.

2.4. Redirecting Shots & Projectiles

When shooting with a trace if the trace hits a portal it will be redirected by him. To make this possible we spawn an actor that will help redirect the shot, on that portal’s target and we convert his location and rotation based on the target. Then we set his position to the converted location, for the rotation we have to set it based on the converted rotation and add the controller rotation of the player that shooted to the portal. Since these are new traces that are created by the portal we have to check again if they found a player, if so they will apply the damage from the weapon that has been used by the player when shooting. After all this we destroy the actor that helped us redirect the shot.

The projectile works a little bit differently, we start by saving his velocity in the moment of impact. Then we convert the rotation and the position. Then we set the projectile position, and the rotation, for the rotation we have to set it based on the converted rotation and add the controller rotation of the player that shooted to the portal then we apply the previous saved velocity with the dot product of his vectors. After that we invert the velocity so the projectile can keep his momentum and velocity to the new trajectory.

3. Weapons

3.1. Weapon Base

Our weapon base class contains all the information needed to create any type of weapon, hence we have shooting types, Linetrace or Projectiles, Reloadable or Not, and all the weapons that currently exist.

This class Handles every common action between all the weapons that currently exist, such as holding ammo, handling fire-rates, reloading, receiving ammo, and pick-up logic, and finally the shooting. We also made the reload so that the player would not lose the bullets that were in the clip, and if the player tries to reload and he does not have sufficient bullets to fill the clip, the clip size will update with the amount of the current bullets.

The weapon logic is not implemented inside this class, since we have weapons that derive from this main class.

When shooting instead of looking through which type of weapon it is, we filter the results. This is way faster than simply checking for every single weapon that has been created, as so in this project, we check if the weapon is of type Linetrace or Projectile.

By doing this we cut the options by half already, in this case we have a small amount of weapons, but in a bigger project we have a variety of weapons we would make sub filters to achieve the weapon in the quickest time possible.

3.1.1. Rocket Launcher

The rocket launcher spawns a projectile that will look for anything to blow up, unless if he finds a portal, if so he will be redirected to the new location provided by the portal traveling.

If he hits another thing he will explode, upon on exploding, he will search for players inside his blast radius if he finds a player the projectile will get the players distance from the center to the location that the player is in, after check the distance, the projectile will calculate the amount of the damage multiplier will be applied to him, based on that distance. Is important to say that we will always receive full damage if we are inside 20% of the blast radius to the center, if not he will calculate the damage multiplier starting from that 20% mark to the end of the radius.

The damage calculation, this one was tricky, but pretty important, the distance check would not know if the character was behind a wall, and to fix it, we decided to check from the center of the blast radius for player body parts.

In the end if a character's knees are behind a wall but the arms are not, we would damage what is exposed but not what's hidden.

With this, we have our radial damage complete and the possibility to avoid damage if the character is hidden behind a wall.

The logic behind this weapon is pretty simple:
We shoot in a straight line from the center of the camera to the “infinite and beyond”. If we find a player we apply the damage from our base class, but if we find the head of the player we have to get his max health and apply all that damage to him, so he will never survive.

If the trace found a portal, he will redirect.

3.1.2. Assault Rifle

The logic behind this weapon is pretty simple:
We shoot in a straight line from the center of the camera to the “infinite and beyond”. If we find a player we apply the damage from our base class, but if we find the head of the player we have to get his max health and apply all that damage to him, so he will never survive.

If the trace found a portal, he will redirect.

3.1.3. Shotgun

The logic behind this weapon is pretty simple:
We shoot in a straight line from the center of the camera to the “infinite and beyond” but we apply a random value to the end of the trace so he can spread from the center, he keeps doing this until the max number of spread shots is achieved. If we find a player we apply the damage from our base class.

If the trace found a portal, he will redirect.

4. Ammo

4.1. Ammo Base

When colliding with ammo the object will check if the player has the weapon or not, if so, the ammo will be delivered to the player who collided with it, if not the ammo will stay on her spot and wait for someone to pick it up.

5. Game Start

5.1. Color Attribution & Match Start

When a player logs-in he will be added to an array and their input will be disabled until a value that is incremented reaches the number of max players per match, if so, the game will start.

Upon starting the game the players in the array will get his colors and their input enabled.

6. Sessions

The Unreal engine provides all the basics you need to host and join a game. However, once we exit the editor, the simple action of joining a server no longer works.

Within the editor, the Unreal engine provides an online sub-system that is no longer present once you are in a standalone game instance.

What we need to do is implement this on our own, or use one of the provided, for development purposes the OnlineSubSystemNull is suggested to work for LAN gameplay only. And we have used it. So no port-forwarding is required since it only works in a LAN environment

For simplicity purposes we can only join the first session that is found from the created ones, and we cannot destroy sessions and handle network errors upon host disconnecting, the implementation of the previous is not hard to do but we decided to go simple. Although we are not trying saying that the previous implementations are unnecessary which they AREN'T.

So to make this beauty work, we start by enabling the OnlineSubSystem module in your project Build.cs. This module will offer us all the Session related code. An OnlineSubSystem has a lot of other functionalities, such as user data, statistics, achievements, and much more. But we are only really interested into the "SessionInterface" of it.

And now we know that there are two major Interfaces involved: IOnlineSubsystem and IOnlineSession each handles crucial parts of integrating our game with the subsystem.

In the end everything is basically driven by sessions, we notify a service about our servers presence, the service is then queried by clients who want to find active game sessions.

A client then requests to join a given session, and then if everything is well, the client is allowed to travel to that server.

7. External Info

7.1. Internet Sources

How were the portals in Portal created? | Bitwise

Why 0.1 Does Not Exist In Floating-Point - Exploring Binary

GlobalVectorConstants::KindaSmallNumber | Unreal Engine Documentation

Multiplayer in Unreal Engine: How to Understand Network Replication

Actor Replication | Unreal Engine Documentation OnlineSubsystemNull | Unreal Engine Community Wiki

7.2. Book Sources

“A Tour of C++”, Bjarne Stroustrup, 2013