Victor Haskins

Game Programming II

Lab 07

Introduction

This round of lab work compounds upon the networking basics and shows more state changes as well as working with client-side prediction methods.

Methods

We start by acquanting ourselves with the model in the prefab we are using, UnityGuy. We are then directed towards our focal point of coding: the player control script. We will start with a struct called PlayerState that will hold the the individual elements of the player position and rotation followed by a syncvar for that PlayerState just called state. We initialize the state SyncVar with an InitState() function and then set up a function SyncState() that transposes the state SyncVar variables onto the prefab's actual transform and rotation. To finish this first section, we are adding the InitState() and SyncState() functions to the Start() function to be run at startup.

The next addition is the Move() function and is admitted to be the most difficult. We send in the parameters of the previous PlayerState, and the new Key input from the player. We start with creating blank position and y-rotation variables. We then set up a switch to run through possible Key codes for the newKey, giving forward/backward, left/right, and rotation orders. We see later that these are tied to world settings and not player settings. After reading what key is being read and the appropriate changes are stored in the temporary values, a new PlayerState is returned where the temp values added,(positive or negative,) to the old state.

Now the update() function is...updated. We preface the next chunk inside an if statement checking if this is the local player running the update and are given the warning that any isLocalPlayer code is overlooked during the awake() function as well as needing to be in a script that extends the NetworkBehaviour class. We give the isLocalPlayer section an array to check for valid commands and then runs a foreach function checking all valid entries against the input. If the input isn't a recognized command, the loop iteration is skipped. One less thing to goto and run I suppose. If the command is recognized though, we are sent to the next function we will make, the command function CmdMoveOnServer.

This new function takes the input command, calls the move command and enters the current state and input information.

This will finish the first round. We save our work by duplicating the player prefab, removing the script from the old prefab, creating a new script we have labeled Lab02b\_PlayerControlPrediction.cs. After copying and pasting the old code into the new script we make the following changes.

We start with replacing the state syncVar with a new one called serverState. Since this script will be used to show server reconciliation, this new syncVar will also need to update a queue of KeyCode values that we will add called pendingMoves. It will be updated by being given a "hook" which is a property that will be added to the SyncVar to look like [SyncVar(hook = "OnServerStateChanged")]. This way, when the SyncVar variable changed, the function specified in the code will run. We'll get to the that function in a bit. We also add another PlayerState variable that will only be used by the client called predictedState. At this point it is also important to note that the library System.Collections.Generic needs to be added to the script so a queue will run.

For our Start() function changes, after the InitState() function is called but beore SyncState(), we add a line have our predicted state variable to equal the serverState, which we are soon changing to modify the serverState syncVar. We also add an if statement block to check if this is run on the local player. If so, then we instantiate the pendingMoves queue and run the later created function UpdatePredicted State();

To signify the further changes, our PlayerState struct received a new int variable, movementNumber, to keep track of the actual state as it is being used. We also make changes to the InitState function--and later, Move function--as needed.

Next, we update the update()...again. Inside our foreach loop, and further inside the if statement, after checking if the command is valid, it is added the the queue with the .Enqueue method and followed by again calling the not yet made UpdatePredictedState().

We quickly look at our command function to update our use of the state syncvar to now change the serverState syncvar.

The syncState() function takes a quick addition of the new PlayerState variable to read whether to use the predicted state or the server state, depending on if the client is the local player or not, respectively.

We now get to the hook function, OnServerStateChanged, reading in a new playerState parameter. Our serverState now becomes the new state and if the queue set earlier isn't empty, if our moveNumber isn't past the current prediction, we dequeue it until it is even and run the UpdatePredictedState() function.

Now we finally get to the UpdatePredictedState() function. All it does is set up the predicted state to equal the serverState. It then runs all the moves in the queue sequentially.

Another checkpoint and we are off to figure out animations. We add another enum specifically for telling whch animation to use, add this enum to the end of the PlayerState struct, the InitState, Move function, and add another variable called characterAnimationState to the script. We also take the time to add a public Animator variable called controller to modify animations with this enum change.

We then add code that keeps the players from walking off into space in the update function after the foreach loop by enqueuing a blank command.

The move function also modifies the animationState with a new function called CalcAnimation() this will change the animation between Idle, walking forwards and walking backwards. the position and rotation changes are called here so this is where the enum would be called.

Finally, the SyncState function changes the controller to render the appropriate animation by sinding in the casted enum value. One last checkpoint where we modify the animation values a little and the assisted lab work is done. I had to stop at this point to get the lab written up.

Conclusions

It was interesting seeing more implementation of Client-Side Prediction. Some simple errors kept me from running the code early on. I can start to see where all the complexity comes in after having to write up the process and where the animation comes in, but I'd need more time to pour over it. Good luck finding the time.

To be honest, I didn't do the On-Your-Own section in favor of getting other homeworks done as well as recognizing it was too late in the night when working on it and favoring sleep.

Questions

*What is the difference between [Client], [Server], and [Command]?*

Client functions are only to be run by the client, not any other user or the server.

Server functions, by the same logic, are only run by the server and no one else.

Command functions are how clients request actions on the server, essentially telling the server what they want to happen so only clients can call them. Furthermore only the local client can call it, not duplicates from other clients and these command functions must start with "Cmd...".

*What is a hook and what is it used for?*

A hook is a property added to a Sync Variable, or SyncVar, is used to specify a function to be used when the variable changes on the server's simulation, or the "actual simulation."

*What is client-side prediction?*

Client-side prediction is the client's side of the game simulation running with the inputs from the player directly after they are entered. These actions are done before the server has a chance to process them.

*What is latency compensation?*

An extension of the C.S.P. process, the client takes the inputs that have been sent off to the player, and runs them as quickly as possible on the client's software to give the appearance of immediacy.

*Why is client-side prediction and latency compensation used?*

They are used to give the player a more immersive experience. If the client game only in lockstep, the disconnect between inputs and actions would be jarring. It would take the user out of the experience as well as having the player remember the inputs they have entered to predict they're own movements and actions.