Centres universitaris adscrits a la

Grau en Disseny i Producció de Videojocs

DESENVOLUPAMENT D'UN PROTOTIP D'UN VIDEOJOC EN XARXA

Memòria

Jordi Alba Franco
Tutor: Jordi Arnal Montoya
Curs: 2021 - 2022

Dedicatòria

Dedico a aquest treball a Andrew Raya, que m’ha acompanyat durant absolutament tot el procés de creació del treball, m’ha ajudat a tirar endavant en les seccions més dures de tot el trajecte i ha fet de la experiència algo mil cops millor. Sense ella no podria haver tingut la motivació de seguir intentant les seccions més difícils i de donar-li les voltes que fessin falta perquè sortís. Ara ja podrem fer Game Jams de més d’un sol jugador.

Agraïments

Agraeixo el suport de Andrew Raya per ajudar-me a provar tots els prototips, a Sergi Segarra per l’ajut en els models, als meus pares per motivar-me i als meus millors amics per ajudar-me a trencar els prototips per millorar-los... a vegades...

Resum

En aquest treball s'implementen diverses llibreries en xarxa en un prototip creat amb Unity, per la seva posterior comparació en diversos aspectes. Els objectius són implementar diverses solucions per minimitzar els efectes de la latència, la pèrdua de dades i altres efectes habituals en aplicacions en xarxa. Finalment, s'analitzen les implementacions amb eines per simular problemes en la connexió i exposar els resultats en taules, per decidir quina és la millor opció per la necessitat del prototip.

Resumen

En este trabajo se implementan varias librerías en red para un prototipo creado en Unity, para su posterior comparación en diferentes aspectos. Los objetivos son implementar diversas soluciones para minimizar los efectos de la latencia, la pérdida de datos y otros efectos habituales en aplicaciones de red. Finalmente, se analizan las implementaciones con herramientas especializadas para simular problemas en la conexión y exponer los resultados en tablas, para decidir cuál es la mejor opción para las necesidades del prototipo.

Abstract

In this work several libraries are implemented for a prototype created with Unity, for its later comparison in different aspects. The goals are to implement various solutions to minimize the effects of latency, data loss and other common effects on network applications. Finally, the implementations are analysed with specialized tools to simulate connection problems and expose the results in tables, to decide which is the best option for the prototype.

Índex

Índex de figures V

Índex de taules VIII

Glossari de termes X

1. Introducció 1

2. Objectius 3

2.1 Objectius principals 3

2.2 Objectius secundaris 3

3. Marc Teòric 5

3.1 Els inicis 5

3.2 Del multijugador local a en línia 7

3.3 Els nous multijugador 7

3.3.1 Shooters 7

3.3.2 Jocs massius 8

3.4 Xarxes i arquitectures 10

3.4.1 Terminologia 10

3.4.2 Arquitectures 11

3.5 Artefactes en videojocs en línia 14

3.5.1 Latència 14

3.5.2 Predicció 15

3.5.3 Dessincronització 17

3.5.4 Jitter 17

3.6 Motors i llibreries 17

3.6.1 Unity 17

3.6.2 Unreal Engine 22

4. Anàlisi de referents 23

4.1 Source Engine 23

4.2 Battleblock Theater 26

5. Disseny Metodològic i Cronograma 31

5.1 Metodologia 31

5.1.1 Versionatge 31

5.2 Eines i prototipatge 32

5.2.1 Unity 32

5.2.2 Llibreries 32

5.2.3 Decisions finals 33

5.3 Planificació 33

5.3.1 Planificació final 34

6. Anàlisi i resultats 37

6.1 Photon Unity *Network* (PUN) 37

6.1.1 Instal·lació i primers passos 37

6.1.2 Els Components de PUN 38

6.1.3 Mecàniques 39

6.1.4 Player character 39

6.1.5 Compensació del lag i dessincronització 40

6.1.6 Predicció local 43

6.1.7 Menú principal 44

6.1.8 Cercador de sales 46

6.1.9 Escena *Lobby* 47

6.1.10 Escena jugable 48

6.2 *Mirror* **48**

6.2.1 Conceptes generals i primers passos 48

6.2.2 Components de *Mirror* 50

6.2.3 Mecàniques 51

6.2.4 Player character 52

6.2.5 Compensació de lag i dessincronització 53

6.2.6 Predicció local 53

6.2.7 Menú principal 53

6.2.8 Connexió directa 54

6.2.9 Pantalla de la sala 55

6.2.10 Escena del joc 56

6.3 Implementacions addicionals **57**

6.3.1 Personalització 57

6.3.2 Regions 59

6.4 Comparativa entre implementacions **59**

6.4.1 Comparativa General 60

6.4.2 Plans de pagament 63

6.4.3 Simulacions de connectivitat 65

7. Conclusions i reflexió **72**

8. Referències **75**

8.1 Bibliografia **75**

8.2 Ludografia **79**

Índex de figures

Figura 3.1 Art digital. Font: (Noll, 1966) ... 6

Figura 3.2 Conjunt de servidors (*“farm”* en anglès) de “Albion Online” en la seva presentació del 2016. Font: (Salz, 2016). ... 9

Figura 3.3 Diagrama de l’arquitectura P2P. Font: Glazer & Madhav, 2015. ... 12

Figura 3.4 Arquitectura client-servidor. Font: Glazer & Madhav, 2015... 13

Figura 3.5 Codi exemple on es comprova l’autoritat del client a UNet. Font: Sara, ,2022. ... 21

Figura 3.6 Codi exemple de gestió d’autoritats a Netcode. Font: Sara, 2022 ... 21

Figura 4.1 Captura en mode *debug* de *Counter-Strike.* Representat en vermell la posició del personatge al rebre l’impacte. En blau el resultat de la simulació del servidor. Font: Valve Developer Community, 2021. ... 25

Figura 4.2 Animació de llançament horitzontal de *Battleblock Theater*. Font: The Behemoth, 2013... 27

Figura 4.3 Dos jugadors realitzant un *boost* al videojoc Counter-Strike Global Offensive. Font: (CS as fast as possible - FNScene, 2020)... 27

Figura 4.4 Animació de llançament vertical de *Battleblock Theater*. Font: The Behemoth, 2013... 28

Figura 4.5 Exemple d’ús del canó de *Battleblock Theater*. Font: The Behemoth, 2013. ... 29

Figura 4.6 Exemples d’ús de la granota explosiva de *Battleblock Theater*. Font: The Behemoth, 2013 ... 29

Figura 4.7 Obstacles perillosos a Battleblock Theater. Font: The Behemoth, 2013 ... 30

Figura 5.1 Cronograma del treball. Font: Elaboració pròpia ... 34

Figura 5.2 Cronograma del treball representant els temps finals. Font: Elaboració pròpia. 35

Figura 6.1 Representació visual de la no compensació del lag. Escala irregular. Font: Elaboració pròpia. 41

Figura 6.2 Compensació del *lag* utilitzant les dades de la Figura 6.1. Font: Elaboració pròpia. 42

Figura 6.3 Pseudocodi de la funció UpdateNetworkParameters. Font: Elaboració pròpia. 43

Figura 6.4 Captura del menú principal de la versió 0.0.3.0 DEV. Font: Elaboració pròpia a partir del prototip. 45

Figura 6.5 Captura del menú principal de la versió 0.0.7.7 DEV. Font: Elaboració pròpia a partir del prototip. 45

Figura 6.6 Captura de la interfície de cerca de servidors de la versió 0.0.7.7 DEV. Font: Elaboració pròpia a partir del prototip. 46

Figura 6.7 Escena "*lobby"* des del punt de vista del *host.* Font: Creació pròpia a partir del prototip. 47

Figura 6.8 Estructura correcta de trucades client-servidor-client. Font: Elaboració pròpia. 50

Figura 6.9 Menú principal de la versió 0.0.8.1 *Mirror*. Font: Elaboració pròpia. ..54

Figura 6.10 Menú de connexió directa. Font: creació pròpia...............................55

Figura 6.11 Pantalla de la sala multijugador. Font: Elaboració pròpia.................56

Figura 6.12 Menú de personalització amb la categoria de "*Head*" desplegada. Font: Elaboració pròpia. 58

Figura 6.13 Dades que conté un "*Item*". Cada instància emmagatzema informació sobre una peça de roba. Font: Elaboració pròpia. 58

Figura 6.14 Entrada individual dins la base de dades en JSON. Font: Elaboració pròpia. 59

Índex de taules

Taula 6.1 Comparativa general entre *Mirror* i PUN. Font: Elaboració pròpia a partir de Mikson, 2019. 61

Taula 6.2 Comparativa de les funcionalitats de *hosting* que ofereixen les dues llibreries. Font: Elaboració pròpia a partir de Mikson, 2019. 62

Taula 6.3 Comparativa entre les solucions que les llibreries implementen per defecte. Font: Elaboració pròpia a partir de Mikson, 2019. 63

Taula 6.4 Taula de preus de PUN v2 de Photon. Font: elaboració pròpia a partir dels valors de la web oficial de Photon. 64

Taula 6.5 Taula de proves realitzada amb una eina per afegir problemes de xarxa de forma fictícia. Font: Creació pròpia a partir del prototip. 67

Taula 6.6 Taula de proves realitzada amb una eina per simular problemes en la connexió dins el prototip amb. Font: Creació pròpia a partir del prototip. 69

Glossari de termes

Artefacte – Efectes visuals no intencionats causats per problemes en línia.

Autoritari – En context, que la decisió que emet un dispositiu té més valor que un altre en una xarxa.

Bug(s) – Errors en la programació no intencionats que causen un comportament inesperat del software.

Client **–** Ordinador o programa que demana informació a un Servidor a través de la xarxa en una organització de tipus client-servidor.

Indie – Desenvolupament de videojocs independent, que no disposa d'ajut financer d'una distribuïdora.

Lobby – Sala virtual on el client s'espera fins que comenci la partida multijugador.

Peer – En l'arquitectura Peer-to-peer (P2P), un dels sistemes que participa en la connexió de la xarxa.

Servidor – Ordinador o programa que retorna qualsevol tipus d'informació a un Client.

Personatge local – Un personatge local fa referència a aquell que el jugador pot controlar amb el seu client.

Personatge no local – Un personatge no local fa referència a aquell que pertany a un altre jugador i no es pot controlar.

Propietari – Un propietari és el client que disposa de control sobre un objecte instanciat en tots els clients. En canvi, un client que no pot controlar aquell objecte però el té instanciat i espera que el propietari el mogui en comptes d'ell se l'anomena observador. Un client pot ser observador d'uns objectes i alhora propietari d'altres.

1. Introducció

Des dels inicis dels videojocs es va observar el potencial que podrien tenir amb múltiples persones interactuant alhora en un mateix entorn. Amb Doom i Quake com a precursors d'una era de videojocs multijugador, establint una moda que va generar en noves sagues.

Aquest corrent va estar acompanyat de millores tecnològiques i noves maneres de crear experiències multijugador, escalant fins mons on concurrentment es podien tenir centenars de jugadors.

Aquest treball estudia per sobre l'evolució dels videojocs multijugador i la manera de fer quan van sorgir i la metodologia d'avui en dia. Segueix per analitzar diferents eines per obtenir un prototip senzill que permeti provar diferents implementacions de llibreries, amb marge per experimentar funcionalitats descobertes al llarg del treball.

Finalment, es compara les implementacions de manera subjectiva, tenint present la necessitat del prototip i la intenció del disseny. Un apartat important que té el treball és la comparativa dels preus de les llibreries, que per norma general varien segons els usuaris concurrents que l'aplicació té.

Tot i que no tots els objectius del treball s'han complert, el treball ha implementat funcionalitats interessants que amb més iteracions pot ser de profit en futurs projectes.

2. Objectius

L'objectiu d'aquest treball és implementar una llibreria de multijugador en línia en un prototip utilitzant el motor de Unity, dissenyant i programant solucions per artefactes en la jugabilitat del multijugador.

Com a finalitats secundàries del projecte, analitzar el cost en temps i dificultat d'altres llibreries amb la finalitat de comparar-ho amb la primera implementació.

2.1 Objectius principals

- Prototipar un videojoc cooperatiu en línia.
- Experimentar amb la implementació *online*.
- Implementar solucions pels artefactes que ocorren en una partida multijugador.
- Analitzar el comportament del multijugador sota estrès, com alta latència o dessincronització d'elements dinàmics i estàtics.

2.2 Objectius secundaris

- Implementar diverses llibreries multijugador i adaptar les solucions a cada una d'elles.
- Connectar el joc entre diverses plataformes amb una mateixa versió del client.
- Comparar el rendiment de les diverses implementacions amb una anàlisi estadística.

3. Marc Teòric

En aquest apartat es pretén explicar l'evolució que van tenir els primers jocs multijugador, i explicar on van sorgir, que va ser important per als jugadors i com van anar evolucionant fins a l'actualitat.

Primer es veurà l'inici dels videojocs multijugador en local, després els primers videojocs que van implementar modes de joc en línia, detallant el disseny i la tecnologia que van utilitzar llavors i finalment s'exposa la tecnologia que s'utilitza en l'actualitat. Seguit de les definicions dels conceptes emprats en la creació del prototip pel treball.

3.1 Els inicis

Tot i no ser considerat el primer videojoc multijugador de la història, Tennis for Two, va ser una recreació del mateix esport dins un ordinador analògic l'any 1958. Consistia en una pista de tenis separada per una xarxa i dues línies que simulaven ser les raquetes de cada jugador. Cada una d'aquestes raquetes tenia un controlador endollat a l'ordinador mitjançant un cable i s'havia de jugar a poca distància de la màquina. Es va crear amb la idea que fos una demostració pública de la feina que es feia dins el laboratori de Brookhaven, i tot i ser popular, no es va arribar a desenvolupar més i va ser desmantellat. (Donovan, 2010)

A mitjans dels anys seixanta, es va veure en els ordinadors un potencial no només aprofitable per realitzar càlculs i operacions matemàtiques, sinó per entretenir i impressionar. L'any 1961, el club "Tech Model Railroad Club" va començar el que ara és considerat el primer videojoc per l'ordinador PDP-1: "Spacewar!". En aquesta època el públic general no tenia accés a ordinadors ni consoles, i només el van distribuir de forma gratuïta a altres centres on disposessin del mateix model d'ordinador. El joc consistia en dos jugadors amb els seus respectius controladors connectats a l'ordinador, cada un manejant una nau espacial. L'objectiu del joc era destruir a l'oponent sense ser arrossegat dins un forat negre. (Graetz, 1981)

"Spacewar!" no va ser l'única creació digital que va promoure els ordinadors. A. Michael Noll també ho va fer utilitzant patrons matemàtics, fins llavors únicament utilitzats per representar dades de forma visual. (Noll, 1966).

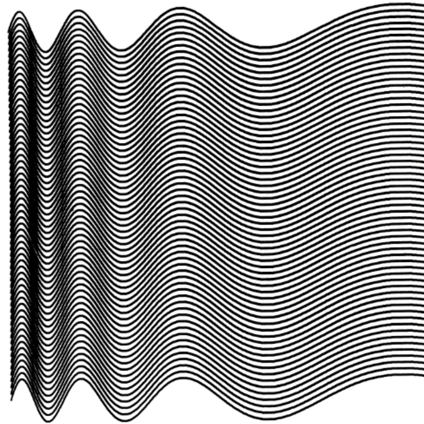

Figura 3.1 Art digital. Font: (Noll, 1966)

És a partir d'aquest punt en la història on es veu la capacitat dels ordinadors per crear i no només com a eina matemàtica programable. Sorgeixen empreses com Activision (1979), Konami (Fundada el 1969, es transforma a jocs el 1973), Nintendo (Fundada el 1889, adopten els videojocs el 1971 amb Magnabox Odyssey)

Sis anys després, Ralph Baer, juntament amb dos enginyers més van presentar un prototip que secretament estaven desenvolupant a l'empresa on treballaven. Aquest prototip, anomenat "Brown Box", és el primer intent per part de Baer de construir i comercialitzar un dispositiu per poder jugar a videojocs a casa. Aquesta consola disposava de dos controladors units amb cables a la consola. L'any 1972 aquest prototip finalment es perfecciona i posa a la venda sota el nom de Magnavox Odyssey. (Donovan, 2010)

La següent evolució dels videojocs multijugador va ser la introducció del cooperatiu de fins a 4 jugadors en format de màquina d'arcade amb els jocs Gauntlet (Atari Games, 1985) i Quartet (Sega, 1986).

Tots aquests videojocs disposen de multijugador en local, on requereixen que els usuaris necessiten controladors físicament connectats i limita la distància a què el

jugador pot estar de la consola. A continuació, es comença a parlar dels primers jocs multijugador en xarxa i com comencen a implementar diversos sistemes.

3.2 Del multijugador local a en línia

La indústria dels videojocs va anar creixent i noves maneres de jugar van començar a sorgir. Entre ells van sorgir jocs multijugador en línia. *Spectre* permetia veure els noms dels contrincants sobre el tanc que controlaven.

El que més va impactar pel que fa a tecnologia i interès va ser *Doom,* que permetia fins a 4 jugadors simultanis dins una partida multijugador. Amb aquests avenços, concepte de joc i eines creades per *Doom,* la mateixa empresa: *id Software,* va llançar *Quake.* (Fiedler, Networking for Game Programmers, 2010)

El potencial dels jocs multijugador era tan alt que Valve va desenvolupar*: Counter-Strike Global Offensive (*2012), un joc purament amb modes de joc multijugador, i que s'ha transformat en un dels referents mundials pel que fa a tornejos competitius anomenats *eSports*. (Hamari & Sjöblom, 2017)

Tots els jocs que pertanyen a diferents sagues mencionades anteriorment encara tenen entregues l'any 2022, i la implementació multijugador i els motors que ho permeten també han evolucionat.

3.3 Els nous multijugador

En aquest apartat s'expliquen els primers videojocs rellevants i quines decisions van prendre respecte al codi multijugador i per què. S'utilitzen tecnicismes detallats en el següent capítol.

3.3.1 Shooters

Doom (1993) va ser el *shooter* en primera persona que va marcar un punt i a part en el món dels videojocs. Incorporava multijugador versus de 4 persones, primerament local amb un multijugador d'arquitectura *peer-to-peer (P2P)*. (Lewis & Jacobson, 2002)

Mesos després, *id Software* va arribar a un acord amb una empresa perquè preparés servidors per al seu joc. Aquests permetrien als jugadors enfrontar-se a adversaris d’altres llocs del planeta. (Kushner, 2003)

Amb l’èxit que va tenir afegint servidors per connectar tots els jugadors, l’empresa va decidir millorar l’arquitectura en línia abans de publicar la seva segona saga: Quake. Es va fer perquè, segons Lewis & Jacobson el 2002, l’arquitectura *P2P* no permetia ser escalada per gestionar més jugadors alhora sense tenir problemes de rendiment.

La nova arquitectura de *Quake*, la de client/servidor, permetia al servidor gestionar la partida i sincronitzar-la amb tots els jugadors alhora. El client només havia d’interpretar les dades i renderitzar el resultat en pantalla.

L’any 1999, *Quake III Arena* i *Unreal Tournament* van sortir al mercat com el primer el successor multijugador de *Quake II* i *Unreal Tournament*, com a producte competitiu amb una jugabilitat similar. Ambdós productes van ser referents en la indústria pel disseny i la tecnologia que implementaven.

Per aquest treball és important el canvi d’arquitectura P2P a Client / Servidor que va realitzar *Doom* i *Quake*. Els conceptes que els creadors van aprendre i posteriorment compartir són de gran utilitat per el desenvolupament del multijugador en línia.

3.3.2 Jocs massius

Aquest apartat parla sobre el disseny i implementacions d'alguns jocs massius multijugador, “*massive multiplayer online*” o *MMO.*

Són jocs dissenyats per allotjar centenars o més clients simultàniament, tots compartint el mateix món.

Segons Darby el 2012, per crear un MMO es necessiten dues coses. Un servidor que gestioni les dades de l'usuari i una aplicació que faci de client perquè es pugui jugar al teu joc. Darby afegeix que dins el seu servidor gestor d'usuaris hi ha diferents servidors, encarregats de diverses tasques:

- Servidor Mestre: Administra tots els altres servidors. Administra i manté les dades.
- Servidor de comptes: Autoritza als usuaris a jugar i els permet escollir personatge.
- Servidor "Proxy": Redirigeix una connexió al servidor correcte. És l'únic que el jugador pot contactar.
- Servidor d'instàncies o zones: Un joc MMO disposa de moltes àrees de joc. Per reduir la càrrega d'usuaris connectats simultàniament a un sol servidor, es distribueixen les zones en diferents servidors. Darby posa l'exemple d'una ciutat molt concorreguda.

Aquesta estructura es pot veure representada de manera similar en la Figura 3.3, on es mostra un diagrama de la xarxa del *MMO* "Albion Online".

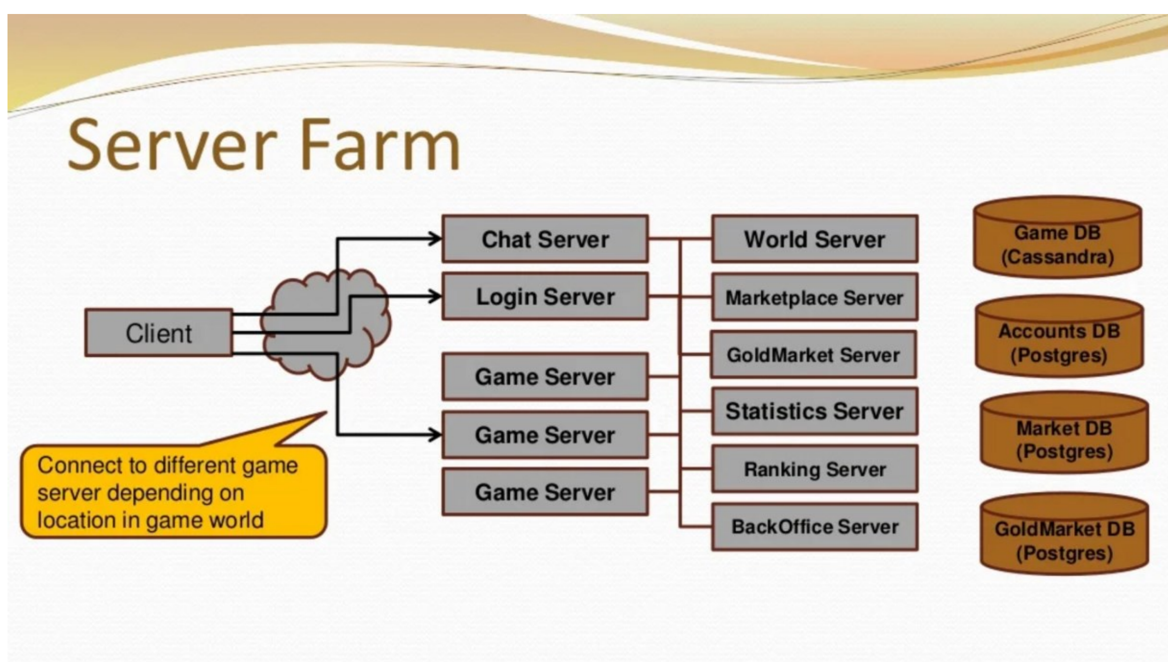

Figura 3.2 Conjunt de servidors (*"farm"* en anglès) de "Albion Online" en la seva presentació del 2016. Font: (Salz, 2016).

L'objectiu d'aquest treball no és desenvolupar un MMO. Però l'estructura o xarxa de servidors que despleguen altres jocs serveixen per aprendre l'estructura.

3.4 Xarxes i arquitectures

En aquest capítol es mostren les primeres implementacions multijugador al llarg dels anys, com han evolucionat les visions d'aquests, una explicació de la terminologia emprada per finalment exposar artefactes que poden alterar l'experiència de joc.

3.4.1 Terminologia

En aquest subapartat s'explica la terminologia que s'utilitza al llarg del treball. Si bé no es troba un terme dins aquesta categoria, es pot trobar dins l'apartat "glossari de termes".

A continuació es detallen conceptes bàsics de xarxes.

- *Round Trip Time* (*RTT*): Com recullen Glazer & Madhav el 2015, *RTT* és el temps que tarda un paquet d'informació en anar de l'origen al destinatari més el temps que tarda a tornar a l'origen la confirmació del missatge. No Bugs Hare afegeix el 2015 que aquest valor depèn majoritàriament del proveïdor d'internet del jugador, i que els valors mínims són difícilment reduïbles per un desenvolupador. Aquest valor es refereix com a latència més endavant en el treball, i s'utilitza com a sinònim. Dins l'apartat "Artefactes en videojocs en línia" s'explora més aquest valor contextualitzat en els videojocs.

- *Packets*: Segons IR Media -una empresa de solucions de telecomunicacions- el 2022, els *packets* són agrupacions de dades enviades a través de la xarxa. Aquest concepte s'aplica a totes les accions que necessiten informació d'internet. El *Packet Loss* és la pèrdua d'aquesta informació mentre es trametia. Citant la mateixa font, la principal causa que llisten és la congestió de la xarxa i que al reensamblar els *packets* en falti informació.

- "*Jitter*": S'anomena així al problema que consisteix a rebre *data packets* de manera desordenada. (Fiedler, State Synchronization, 2015)

- *Remote Procedure Call (RPC)*: Segons Photon el 2022, els RPC són funcions que un client o servidor mana executar en un o diversos clients de la mateixa xarxa.

3.4.2 Arquitectures

Una arquitectura de xarxa és el disseny de les funcions que realitzarà cada dispositiu que participi en una. Existeixen diverses arquitectures i cada una compleix unes funcions.

A continuació s'expliquen les que s'han utilitzat en videojocs en el passat i en el present, les motivacions per utilitzar diferents metodologies i detalls històrics.

3.4.2.1 Peer-to-peer

La primera entrega de la saga de *Doom* disposava d'un multijugador de quatre jugadors simultanis. Els quatre es comunicaven mitjançant l'arquitectura *P2P.* (Fiedler, Networking for Game Programmers, 2010)

En aquesta arquitectura, tots els ordinadors participants estan connectats entre si. Tots transmeten informació als altres constantment. Suposa que el nombre de connexions i la informació que s'ha de transmetre per la xarxa augmenta exponencialment per cada nou ordinador que hi participi. (Glazer & Madhav, 2015)

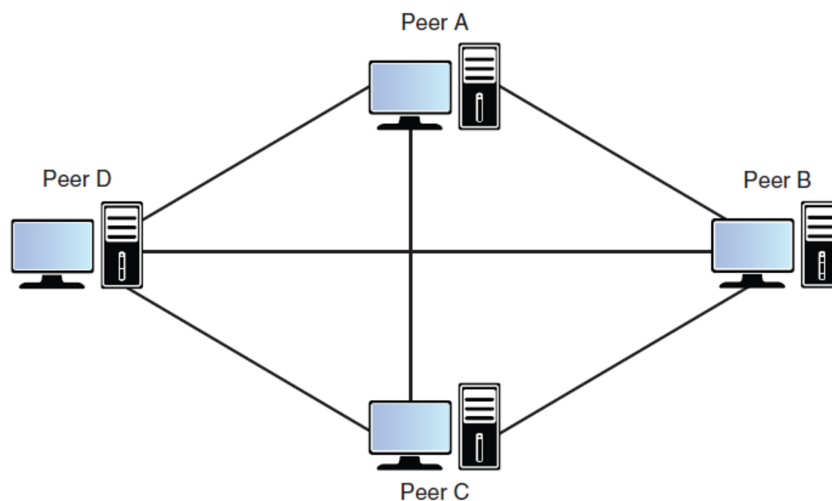

Figura 3.3 Diagrama de l'arquitectura P2P. Font: Glazer & Madhav, 2015.

La limitació més gran d'aquest sistema és que si un client té mala connexió a internet, els altres clients han de parar el joc fins que rebin la informació d'aquest. Això suposa que els jocs frenètics s'interrompin si de sobte un dels jugadors no pot transmetre informació. També suposa una dificultat sincronitzar un nou jugador a mitja partida, ja que hi ha diverses simulacions actives i cap és essencialment la correcta. (Fiedler, Networking for Game Programmers, 2010)

Per aquests motius aquesta arquitectura no s'utilitza en gèneres frenètics de videojocs, i es queda en jocs per torns o d'estratègia a temps real (RTS), perquè la informació que s'ha de transmetre acostuma a ser menor.

La següent arquitectura és la que implementa la majoria de videojocs multijugador i està establerta com a estàndard dels videojocs.

3.4.2.2 Client/Servidor

Tal com detalla Carmack el 1996, després de la implementació del multijugador en línia a *Doom*, es va adonar de la poca escalabilitat que tenia. Per cada client havia de realitzar connexions a cada un dels altres clients, escalant exponencialment per cada un. Pel desenvolupament de Quake, Carmack va decidir adoptar l'arquitectura de client/servidor.

Aquesta arquitectura consisteix a convertir un mateix client de la partida en un servidor autoritari. En comptes que els clients enviïn els inputs a tots els altres (com en P2P), en l'arquitectura de client/servidor tots els clients informen el servidor. El servidor gestiona i valida les accions que cada un ha pres, retornant l'estat final a tots els clients. (Carmack, 1996)

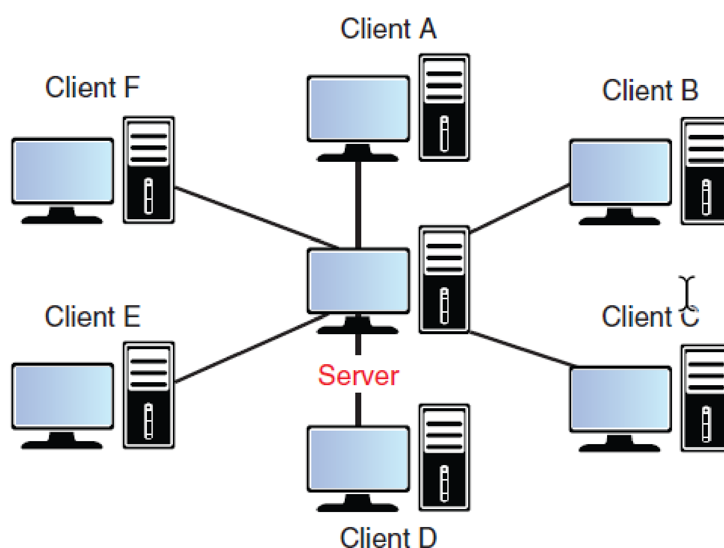

Figura 3.4 Arquitectura client-servidor. Font: Glazer & Madhav, 2015.

Aquest procés redueix la càrrega en tots els clients, ja que en una estructura pura de client/servidor, les funcions del client són recollir els inputs del jugador, enviar-los al servidor i representar en pantalla la simulació del servidor. (Fiedler, Networking for Game Programmers, 2010)

Posant un servidor com autoritari comporta un retard en les accions del client. La informació ha de viatjar del client al servidor i tornar, per cada decisió que pren. El servidor llavors no només torna el valor a un, sinó a tots els clients de la xarxa. Aquestes accions tarden temps a completar-se. Aquest temps s'anomena latència. (Glazer & Madhav, 2015)

Tot i la latència que té el client amb el servidor, a diferència del P2P aquesta arquitectura no ha d'esperar la resposta d'un client abans d'executar més simulacions, així que la qualitat de la connexió passa a dependre de la distància física al servidor i de l'amplada de banda del mateix client, en comptes de dependre

de la pitjor connexió dels clients. (Fiedler, Networking for Game Programmers, 2010)

Com diu Carmack el 1996, un dels avantatges d'aquesta arquitectura són l'augment de jugadors màxims per partida, per exemple jocs com Battlefield poden tenir 64 jugadors dins una mateixa partida. En aquest cas, Battlefield utilitza servidors dedicats, que significa que el procés del servidor va a part del joc. Videojocs grans fan servir el que s'anomena un servidor dedicat "*headless*", que no és un client i només fa de servidor, sense renderitzar gràfics, només gestionar. Una altra categoria de servidor dedicat és el de "listen". On el servidor també és un client dins la partida. La terminologia emprada per aquest tipus de client/servidor és "*host*". (Glazer & Madhav, 2015).

3.5 Artefactes en videojocs en línia

En aquest apartat es detallen alteracions de l'experiència dels jocs en línia causats per la implementació tècnica del multijugador o per l'estat de la connexió a la xarxa.

3.5.1 Latència

La latència, *network latency o ping,* segons la patent (Estats Units d'Amèrica Patent núm. 6,475,090 B2, 2002) i (Estats Units d'Amèrica Patent núm. 5,899,810, 1999) és una mesura que representa el temps que un missatge tarda a passar entre dos nodes d'internet.

En videojocs, aquest concepte de latència es traslllada, tal com descriuen Glazer i Madhav el 2015 "el temps entre una causa visible i una resposta visible" (p.200)[1] per tant, cal mantenir una latència baixa per una millor experiència de joc. La latència es mesura en milisegons (ms), i com menor el valor millor és la qualitat de la connexió. Glazer i Madhav també expliquen l'any 2015 que "La latència pot rondar de 16 a 150 ms abans que l'usuari noti que el joc s'alenteix o no respon

[1] "...the amount of time between an observable cause and its observable effect." (traducció del autor)

correctament", exposant que és un efecte important que s'ha d'intentar reduir per oferir una bona experiència.

Un dels motius principals d'una major latència es dóna per la distància física entre els dos dispositius. Menor distància significa menor *ping*. Altres motius pel qual pot incrementar aquest valor són els problemes físics, com falta de memòria, capacitat de processament o una sobrecàrrega del *software*, que retarden l'enviament o la rebuda de les dades. (No Bugs Hare, 2015)

En el videojoc es poden implementar sistemes per reduir l'impacte del *lag*. Si el *ping* és raonable, es poden suavitzar els moviments entre les posicions que el jugador vol anar o fins i tot predir on estarà.

A partir d'uns valors és difícil oferir una bona experiència a jugadors amb alt *ping*, ja que podria resultar en prediccions o simulacions errònies.

En molts casos, videojocs competitius on les partides es prenen amb serietat, l'acció que es pren respecte a un jugador amb alt *ping* és expulsar-lo. Tot i ser una mesura dràstica, aquests mateixos videojocs tenen infraestructures que ofereixen servidors a diferents punts del planeta per reduir la distància física entre el jugador i el servidor, reduint el *ping*.

3.5.2 Predicció

La implementació del multijugador del *Quake* mencionada al capítol anterior requeria que cada moviment fos confirmat pel servidor abans que el jugador pogués veure que s'havia mogut en el seu propi joc local. Els multijugador moderns implementen solucions que eliminen aquest artefacte.

Carmack (1996), el programador original de *Doom* i *Quake*, va dissenyar tot el canvi d'arquitectura de la segona saga. Al seu diari, traduït, escriu:

> La diferència més gran és la predicció del moviment per part del client. Ara permeto al client especular sobre el resultat del moviment del jugador abans que arribi la simulació autoritària del servidor... el servidor té la paraula final,

així que el client prediu el moviment basat en l'última simulació vàlida per part del servidor.

Aquesta solució carrega al client amb més execució de codi que l'arquitectura pura de client/servidor.

En essència el que disposa el jugador quan es connecta a una partida multijugador és una aplicació que registra el moviment i les accions que realitza per enviar-les al servidor. El que el client llavors mostra és el resultat d'aquelles accions.

El principal problema d'aquesta implementació sense predicció local, és que el personatge només es mouria quan es rebés la resposta del servidor. Donat que les dades tenen una *latència,* el client tardarà tant a moure's com latència tingui. El que la predicció local fa és que el client mostri a temps real el resultat de les accions del jugador, sense esperar la resposta del servidor. Això és possible realitzant la mateixa simulació que fa el servidor però en el client. Suposa una major càrrega de processament, ja que no només renderitza el resultat sinó que computa tots els canvis que ha de gestionar. (Carmack, 1996)

La predicció local no substitueix l'arquitectura "client-servidor" autoritària. El client simula la seva versió, i la transmissió de la simulació entre client-servidor també es realitza. Si hi ha discrepàncies entre les simulacions, el client ha de corregir aquells tots aquells elements que difereixen en el seu estat. Fiedler (2010) destaca, com a dificultat d'aquesta implementació, el fet d'aplicar la correcció del servidor per sobre de la del client. Pot causar que objectes que en local s'hagin activat, per exemple una porta, i que en aplicar la correcció la porta es tanqui de cop davant el jugador. En aquest cas podria haver sigut que en la simulació local el jugador complia tots els requisits per poder obrir la porta, però en el servidor no ho fes, per exemple perquè no estava a la distància mínima d'interacció.

Solucionar aquest artefacte és de summa importància en videojocs competitius que necessiten precisió en les simulacions per registrar i comparar esdeveniments amb milisegons de diferència. Si no el jugador pot sentir que no l'ha eliminat un altre jugador sinó que ha estat el joc que no funciona correctament i no l'ofereix una bona experiència. (Glazer & Madhav, 2015)

3.5.3 Dessincronització

La dessincronització és l'estat en què es troba un client o un servidor durant una sessió multijugador on hi ha informació, sigui objectes, personatges o dades, que no concorden amb els valors correctes.

La pèrdua d'informació és un dels principals causants de la dessincronització, i pot causar que durant un temps un jugador disposi d'una simulació errònia.

3.5.4 Jitter

Com s'ha explicat dins el subapartat de terminologia, el *jitter* és el problema de rebre *data packets* de manera desordenada.

Això és causat per la xarxa o gestions externes al desenvolupador. Per solucionar-ho, Fiedler dins el seu article de State Synchronization el 2015 proposa implementar un *jitter buffer.* La funcionalitat d'aquest *script* és emmagatzemar *data packets* quan el client ho rep i distribuir-los de manera ordenada i espaiada.

3.6 Motors i llibreries

En el mercat hi ha diversos motors que implementen multijugador. L'objectiu d'aquest treball consta en contrastar el rendiment, la facilitat d'implementació i el cost de diverses llibreries.

Per això el treball necessita un motor on es puguin programar les solucions llistades anteriorment, on es puguin gestionar diversos clients amb els seus permisos i es puguin connectar almenys fins a quatre jugadors. A continuació s'han contrastat diversos motors que permeten multijugador, ja sigui a través de llibreries de tercers o implementat de base.

3.6.1 Unity

Unity disposa de diverses llibreries per implementar multijugador. Amb l'objectiu d'implementar-ne un que ofereixi solucions per transmetre dades a través de la xarxa sense massa dificultats, i la rellevància del projecte es mantingui en el disseny

i la correcta programació de les solucions descrites en els apartats de referents i marc teòric del treball.

Les llibreries amb més rellevància a Unity, que tenen més suport en fòrums, servidors o amb la intenció d'obtenir ajuda en la implementació del multijugador són:

- *PUN*, una extensió de *Unity* creada per una empresa externa al motor que ofereix l'arquitectura Client/Servidor i servidors dedicats en diferents regions. És personalitzable i ha estat utilitzat en diversos projectes comercials de tipus *indie*. La mateixa empresa, "Photon", ven altres llibreries de xarxa per Unity.
- *MLAPI*, una llibreria creada per la mateixa empresa que desenvolupa Unity, anunciada el 2019 com a substituta de l'antiga llibreria *UNet*. Usa la mateixa arquitectura de client/servidor amb similituds en la implementació.
- *UNet*, la llibreria que precedeix a MLAPI, basada en l'arquitectura de Client/Servidor. Deprecada amb bugs poc temps després de sortir a Beta. (Mirror, 2022)
- *Mirror*, una llibreria de codi obert basada en l'arquitectura Client/Servidor. Centrada en la facilitat d'ús i en la implementació de jocs multijugador massius.

3.6.1.1 Photon Unity *Network*

En apartat s'explica els aspectes que ofereix aquesta llibreria. La següent informació s'ha extret de la web i la documentació de Photon el 2022.

La llibreria multijugador *Photon Unity Network (PUN)* es defineix a si mateixa com una llibreria per jocs multijugador. Introdueix la cerca de partides (*matchmaking*) a sales on permet sincronitzar els objectes a través de la xarxa. Inclou una connexió ràpida a través dels servidors dedicats de l'empresa.

Inclou diferents configuracions de protocols -que són les normes amb les quals s'envia la informació a través de la xarxa perquè dos ordinadors s'entenguin-, servidors a diferents continents, suport per diferents plataformes i dispositius i escalabilitat per afegir més jugadors a la mateixa partida.

Disposa d'una configuració per establir un servidor privat com a gestor de tots els jugadors perquè puguin cercar partides en línia. Això també ho ofereixen com a servei amb limitacions d'usuaris mensuals i usuaris concurrents.

Proporcionen guies bàsiques per preparar la primera connexió amb dos clients, i de més avançats per implementar compensació del *lag*.

3.6.1.2 *Mirror*

En apartat s'explica els aspectes que ofereix aquesta llibreria. La següent informació s'ha extret de la web i la documentació de *Mirror* el 2022.

Mirror deriva de UNET, la llibreria de gestió de xarxes desenvolupada el 2015 per Unity. *Mirror* va néixer quan UNET va parar el desenvolupament l'any 2017 aproximadament i Unity va obrir el codi per la seva *HLAPI* (*High*-Level *API*). Van solucionar *bugs* i netejar el codi per posteriorment publicar-lo a la "Unity Asset Store".

Mirror ofereix compatibilitat amb diferents "*low-level transports"*, que són llibreries per gestionar la transmissió de dades a través de la xarxa.

Ofereixen components amb funcionalitats genèriques que faciliten l'aprenentatge i el desenvolupament dels videojocs que l'utilitzin. Components com NetworkTransform que sincronitza la posició d'objectes entre clients, o NetworkBehaviour que gestiona les connexions, desconnexions i les autoritats.

També ofereix diverses implementacions del que s'anomena "Interest Management". Que gestiona quins elements sincronitzar amb el servidor per reduir amplada de banda. Aquestes implementacions permeten que els jocs massius s'escalin sense que l'amplada de banda del client se sobrecarregui, ajuda a la visibilitat entre jugadors i a través d'objectes que no permeten la visió i finalment ajuda a reduir les instàncies on tramposos poden obtenir informació d'altres clients.

Mirror disposa d'una gran comunitat, amb més de 10.000 persones a fòrums comunitaris i 100.000 descàrregues a l'any. És completament gratuït i de codi obert.

3.6.1.3 MLAPI / Netcode for GameObjects

En apartat s’explica els aspectes que ofereix aquesta llibreria. La següent informació s’ha extret de la web i la documentació de MLAPI, dins de Unity, 2022. Diversos articles s’han utilitzat al llarg d’aquest subapartat, com els de Sara (2022), editora de la documentació de *Unity Multiplayer Networking*.

MLAPI és una llibreria desenvolupada per Unity Technologies, amb la versió 0.1.0 anomenada “MLAPI”. Durant el desenvolupament del treball, aquesta llibreria s’ha actualitzat amb un nou nom. La versió 1.0.0 d’aquesta llibreria s’anomena “*Netcode for GameObjects*” o simplement “*Netcode*”.

Netcode ofereix suport a Windows, MacOS, Linux, iOS, Android, plataformes XR derivades de Windows, Android iOS, i plataformes tancades com consoles (PlayStation, Xbox o Nintendo Switch).

Deriva de l’antiga llibreria de Unity Technologies, MLAPI i UNET.

Ofereix components similars a *Mirror*, ara canviats per la nova actualització. “Network*Identity*” passa a dir-se “NetworkObject”.

Algunes de les novetats que Netcode mostra en la seva web és la reducció de comprovacions al cridar funcions des d’un client o un servidor. Com es pot veure en la Fig. 3.6 abans de cridar cap funció es comprova l’autoritat del client.

```
public void Start()
{
    if (isClient)
    {
        CmdExample(10f);
    }
    else if (isServer)
    {
        RpcExample(10f);
    }
}
[Command]
public void CmdExample(float x)
{
    Debug.Log("Runs on server");
}
[ClientRpc]
public void RpcExample(float x)
{
    Debug.Log("Runs on clients");
}
```

Figura 3.5 Codi exemple on es comprova l'autoritat del client a UNet. Font: Sara, ,2022.

Com es pot veure en la Fig. 3.7, a Netcode no es realitza la comprovació d'autoritat "*isClient*", sinó que es truquen les dues funcions alhora i internament es gestiona.

```
public override void OnNetworkSpawn()
{
    ExampleClientRpc(10f);
    ExampleServerRpc(10f);
}
[ServerRpc]
public void ExampleServerRpc(float x)
{
    Debug.Log("Runs on server");
}
[ClientRpc]
public void ExampleClientRpc(float x)
{
    Debug.Log("Runs on clients");
}
```

Figura 3.6 Codi exemple de gestió d'autoritats a Netcode. Font: Sara, 2022

Netcode ofereix un sol transport oficial, amb suport de transports desenvolupats per tercers.

És gratuït sota llicència MIT permissiva amb codi obert.

3.6.2 Unreal Engine

Unreal Engine és desenvolupat per Epic Games. Es va publicar per primer cop el 1998 amb la versió 1.0, fins al 2022 quan Unreal Engine 5.0 va ser publicat. Escrit en C++, de codi obert i gratuït fins a assolir un mínim de beneficis. Permet desplegar a Windows PC, Linux, PlayStation 4 i 5, Xbox X, S i One, Nintendo Switch, Google Stadia, macOS, iOS, Android, diverses plataformes AR i plataformes VR com SteamVR i Oculus. També disposa d’una botiga per accedir a *assets* fabricats per ells o la comunitat. (Epic Games, 2022)

Unreal Engine permet desenvolupar jocs multijugador en línia a través del seu sistema de *Networking*. Dins la seva pàgina web de *Networking* el 2022, disposen d’elements com RPC, Steam Sockets per connectar amb la plataforma Steam de Valve i tutorials entre d’altres per facilitat la implementació.

4. Anàlisi de referents

Dins aquest capítol es mostren els referents que s'han utilitzat com a principals idees en dissenyar el prototip i implementar les llibreries.

4.1 Source Engine

Aquest motor de videojocs desenvolupat per l'empresa *Valve*, va ser utilitzat per crear jocs de la saga *Half-Life*, *Counter-Strike* i *Left 4 Dead*. Llençat l'any 2004 com a successor de *GoldSrc*, i en essència una modificació del motor original de *Doom* i *Quake*.

Creat amb una visió de futur, *Source Engine* permetia evolucionar incrementalment amb la tecnologia. El mòdul rellevant per al treball és el de multijugador, que s'utilitza en jocs com *Team Fortress* 2, *Counter-Strike*: *Source* i *Left 4 Dead*.

En un bloc de desenvolupadors de Valve, entren en detall com el van implementar. Tot el que es menciona a continuació prové del mateix bloc de desenvolupadors oficial citat a les referències. (Valve, sense data)

Segons expliquen els desenvolupadors de Valve, el seu sistema multijugador se centra a minimitzar l'impacte de la pèrdua de dades o que la fluïdesa del joc es vegi afectada, sobretot en jocs d'acció que requereixen respostes ràpides del jugador.

Per reduir els efectes els servidors dels jocs que utilitzen Source implementen tècniques com compressió de dades i compensació de *lag.* Tècniques que són invisibles al client, però milloren l'experiència del jugador exponencialment. No es limiten a les tècniques per part del servidor, sinó que el client també implementa predicció i interpolació de posicions per millorar la fluïdesa de la partida.

Cada servidor té un temps assignat entre simulacions (*tick)* per evitar sobrecarregar la màquina. Aquest temps es diu *timestep* i a *Source* varia depenent de quin joc i quin servidor ho suporta. El bloc parla de 15 milisegons (ms), a 66 simulacions per segon aproximadament. Les simulacions per segon s'anomena *tickrate*. Llavors els

servidors dels quals parla el *Valve* tenen una configuració de: 15 ms entre cada *tick,* resultant un *tickrate* de 66.

A cada *tick*, el servidor processa tota la informació enviada pels usuaris, normalment l’obtinguda pels perifèrics (ratolins, teclats...). Aquesta informació no es tramet constantment, sinó a intervals determinats pel servidor.

Després de tot això, realitza la simulació pertinent. En acabar-la, decideix si algun client necessita informació sobre l’estat del món generat. No s’envia la informació sencera de tot el món, seria un paquet d’informació massa gran i els jugadors amb pitjor connexió podrien patir pèrdues d’informació. El que *Source* fa, és trametre els canvis rellevants que s’han fet durant aquella simulació. Per exemple, el mapa i tots els elements estàtics i dinàmics se sincronitzen al principi de la partida i els estàtics s’ometran en futures actualitzacions perquè no han de patir cap canvi.

Un cop la simulació s’acaba i el client la rep, aquest últim interpreta les dades rebudes per reproduir-les localment.

Aquestes dades entre client-servidor necessiten transportar-se a través d’internet, i depenent d’on es trobi l’ordinador central significa que ha de recórrer una distància física. Això comporta cert endarreriment en el transport de la informació que diversos jugadors poden estar enviant i intentant rebre constantment. Per posar un exemple, si un jugador dispara a un altre en una posició que veu en local, però resulta que té *ping* elevat i no ha rebut la informació que el jugador ja no era allà, el servidor podria denegar l’impacte que en local havia fet.

Valve soluciona aquests efectes registrant a quin moment exacte de la partida s’executa cada acció, recreant el món amb la informació que té de tots els jugadors en aquell moment del temps i trametent els resultats correctes a tots els altres.

Per entrar en detall en cada part, poder executar el codi que compensa el lag que de diversos clients, el servidor guarda les últimes posicions dels jugadors (en aquest cas, les de l’últim segon). Quan hi ha una acció que comporta alguns jugadors, com disparar i rebre un impacte, el servidor calcula el temps que tarda la informació a viatjar (*ping*) del jugador que dispara. El valor del *ping* el calculen segons el temps que porta el servidor, amb la latència mitjana que té aquell jugador

i la configuració del client sobre les interpolacions. Llavors, recuperant les posicions dels jugadors just en aquell moment del temps es genera una simulació on el tret és realitzat, i comprova si ha impactat o no.

Figura 4.1 Captura en mode *debug* de *Counter-Strike.* Representat en vermell la posició del personatge al rebre l'impacte. En blau el resultat de la simulació del servidor. Font: Valve Developer Community, 2021.

Aquesta solució permet una compensació del *lag* per evitar efectes visuals de teletransport o desplaçaments incorrectes. Però amb aquesta correcció inherentment apareix el problema de rebre impactes quan estàs a cobert en local quan el servidor t'ha tirat endarrere per compensar un altre jugador.

En el text afegeixen que la detecció es podria fer en local, però corren el risc d'exposar el joc a les trampes, i que clients fraudulents enviessin informació incorrecta dels altres jugadors per tenir un avantatge sobre ells.

El segon artefacte que mencionen dins una pàgina diferent, és que si un jugador realitza un moviment i ha d'esperar al fet que el servidor confirmi el moviment, el joc en local només funcionaria quan es rebés aquella informació.

La intenció del videojoc frenètic és que sigui fluid i l'experiència sigui bona sense disposar d'una connexió òptima. Per això Valve implementa la predicció. La

predicció d’on estarà el jugador en cada moment és realitzada pel client sense esperar la resposta del servidor. Més tard es compara amb la decisió autoritària del servidor per veure si és errònia i s’ha d’aplicar canvis en la simulació local. (Valve, 2021)

L’altra tècnica que utilitzen per evitar que els personatges es moguin erràticament o es transportin entre diferents posicions és la interpolació.

Partint de la idea que Valve exposa, un servidor genera 20 actualitzacions del client. El client funciona a 60 actualitzacions per segon (*frames* per segon o *fps*). El client renderitza 3 o més *frames* per cada actualització del servidor que rep. El client podria moure’s 3 cops abans que el servidor li denegui la posició i el retorni endarrere, en el pitjor dels casos transportant-lo a una zona perillosa.

La solució a aquest problema que implementen és el *buffer* de dades. És una llista on es guarden les dades que el servidor ha enviat els últims instants. El client llavors utilitza aquestes dades per corregir el joc al llarg de diverses simulacions. D’aquesta manera, encara que un jugador perdi la informació de la següent simulació, pot usar les antigues per crear la il·lusió que el joc funciona correctament.

4.2 Battleblock Theater

Battleblock Theater és el principal referent quant a mecàniques del treball. És un joc de plataformes en 2D cooperatiu de dos jugadors o competitiu de 4 jugadors.

Disposa de moviment lateral, ajupir-se i doble salt. Pel projecte, el moviment d’aquest joc s’ha utilitzat com a referent. Es vol un moviment precís sense inèrcia en el personatge local. El moviment generat per agents externs al personatge es tractarà com a inèrcia i funcionarà respecte a la matriu món. Això pot potenciar el descontrol quan un altre jugador vulgui causar el caos durant la partida, forçant moviment en els altres personatges per causar estralls en el progrés del nivell.

Les mecàniques principals consten de diferents cops cos a cos per empentar i/o elevar a enemics o aliats, amb la finalitat de superar obstacles o llençar-los als perills del nivell i eliminar-los. Els jugadors tornen a aparèixer en punts de guardat, però causa que hagin de tornar a superar part del nivell.

Figura 4.2 Animació de llançament horitzontal de *Battleblock Theater*. Font: The Behemoth, 2013

Aquest és el llançament en horitzontal. Quan el jugador llançador manté pressionat la tecla activadora i algú col·lideix amb l'esquena del seu personatge, aquest últim és propulsat en diagonal, cap amunt i en la direcció horitzontal en què es movia. Aquest salt conjunt no esgota el doble salt i permet arribar a plataformes on un de sol no podria arribar.

Per superar elements a una alçada superior a la que pot arribar un personatge amb doble salt o elevant-se sobre algun objecte, és utilitzar un altre jugador. A aquesta tècnica s'anomena *boost*. Consisteix a usar un altre jugador com a plataforma per arribar a punts alts. S'implementa aquesta mecànica cooperativa en jocs com "Battleblock Theater", la saga "Counter Strike" o "Rust".

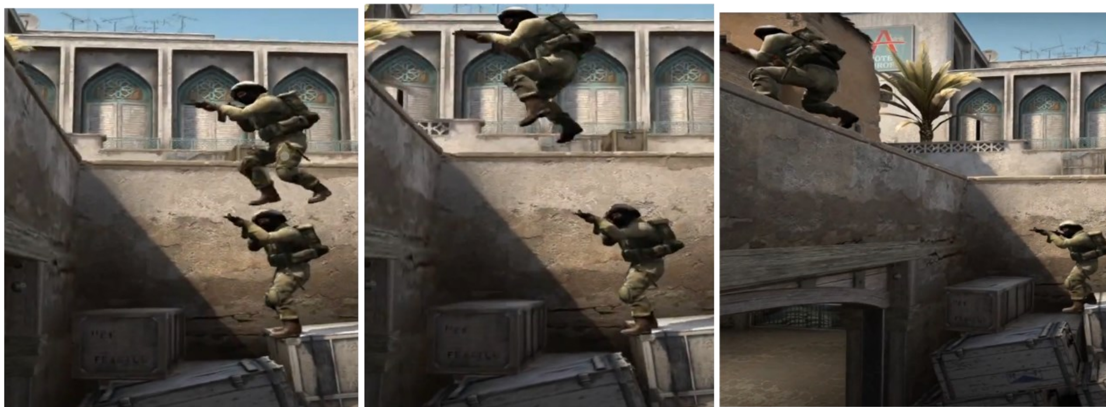

Figura 4.3 Dos jugadors realitzant un *boost* al videojoc Counter-Strike Global Offensive. Font: (CS as fast as possible - FNScene, 2020)

És una alternativa cooperativa que incita als jugadors a no dependre dels elements del mapa.

Una altra manera de superar obstacles verticals amb mecàniques és la d'empènyer verticalment a un personatge que reposa sobre el cap d'un jugador. Aquest moviment desplaça verticalment per assolir llocs elevats sense afegir desplaçament horitzontal. No esgota el doble salt i l'alçada equival al *boost*.

Aquesta mecànica afegeix varietat per superar obstacles i intentar no avorrir amb la repetició de la mateixa mecànica.

Figura 4.4 Animació de llançament vertical de *Battleblock Theater*. Font: The Behemoth, 2013

El joc també incorpora dos espais per escollir entre diverses eines que afegeixen maneres d'eliminar personatges, superar obstacles i guanyar independència de l'altre jugador si s'està jugant en local.

El canó dispara plataformes que s'enganxen a les parets durant cert temps. Permeten crear elements dinàmics per superar els nivells de manera creativa. Aquests elements també es poden fer servir per distorsionar el progrés d'altres jugadors.

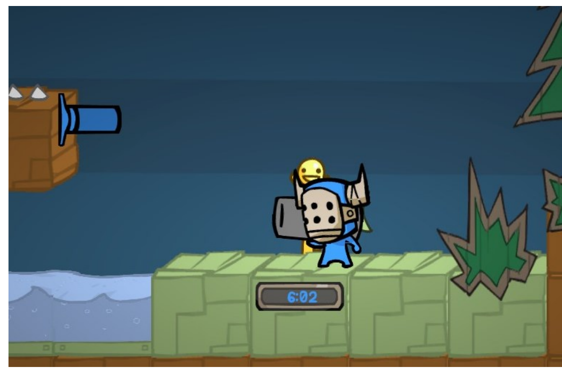

Figura 4.5 Exemple d'ús del canó de *Battleblock Theater*. Font: The Behemoth, 2013.

La granota explosiva crea un personatge mòbil que camina sol pel nivell. Si qualsevol criatura se li acosta, la granota explotarà, eliminant totes les criatures que tingui al voltant.

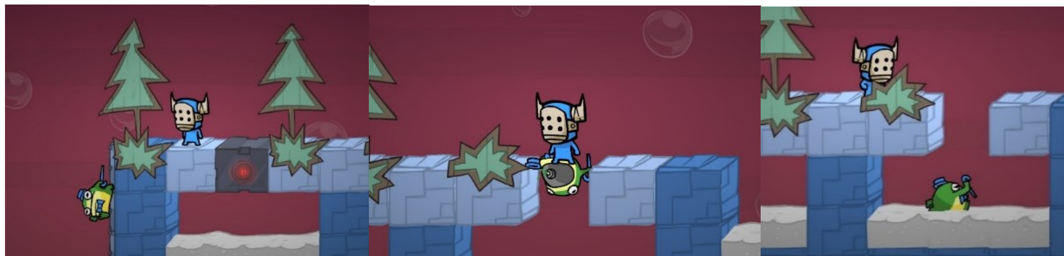

Figura 4.6 Exemples d'ús de la granota explosiva de *Battleblock Theater*. Font: The Behemoth, 2013

Aquesta granota té l'habilitat de caminar per parets i sostres, és invulnerable a les trampes i té col·lisió amb el jugador. L'ús principal és per superar zones de perill utilitzant-la com a plataforma per ràpidament reiniciar el salt.

Altres eines més ofensives que pot escollir utilitzar el jugador son: mines, granades, boles de foc, boles de gel i boles corrosives. Totes elles amb comportaments similars, dissenyades per eliminar enemics, o als jugadors si no tenen cura.

Així com l'aigua i les trampes, també hi ha làsers elèctrics que maten instantàniament al jugador.

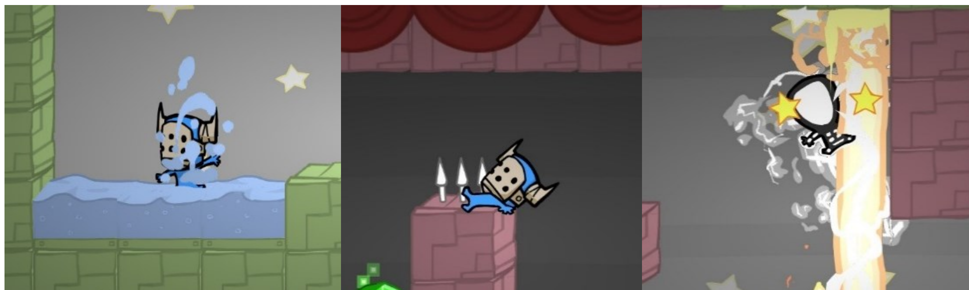

Figura 4.7 Obstacles perillosos a Battleblock Theater. Font: The Behemoth, 2013

Aquests elements en conjunt amb enemics i els blocs especials creen entretinguts dissenys de mapes per superar sol (amb eines) o en cooperatiu. Per al prototip, les trampes no s’ha considerat afegir-les per centrar la jugabilitat en la interacció entre personatges.

5. Disseny Metodològic i Cronograma

5.1 Metodologia

El treball es divideix en tres fases, corresponent a les determinades per l'entitat com a seguiment del treball.

La primera fase consta de l'anàlisi dels referents, on s'analitzen implementacions teòriques dels videojocs en xarxa multijugador des del punt de vista dels desenvolupadors, a quines conclusions van arribar amb la tecnologia que disposaven i quines dificultats els portaria tenir aquestes implementacions en el futur.

També s'inclouen referents quant a jugabilitat i estètica que es vol obtenir en el prototip. Aquesta base perdurarà al llarg del treball, tot i ser una part més secundària del treball, és rellevant per poder provar el joc en diferents situacions.

La secció del marc teòric mostra cronològicament els avenços rellevants del desenvolupament dels videojocs multijugador cooperatiu o en línia. S'hi destaquen jocs que van causar un punt d'inflexió en tecnologia i com es veien els jocs socialment.

Una secció està dedicada a les arquitectures que es fan servir avui en dia en els videojocs, i a continuació s'expliquen els *bugs* i artefactes comuns que poden sorgir en implementar un multijugador en línia. Aquesta secció del treball s'anirà expandint a mesura que es realitzi la segona fase, durant la implementació sorgiran problemes que no s'han pogut anticipar.

5.1.1 Versionatge

El prototip s'ha desenvolupat amb un seguit de versions similars a l'estructura del *Semantic Versioning* (*SemVer*). On cada número de la versió té un significat en l'estat de l'aplicació (Preston-Werner, 2013). En videojocs multijugador s'utilitza per no mesclar jugadors amb diferents clients en les mateixes sales, i que disposin tots del mateix estat del joc.

L'estructura habitual és la següent (1.2.3.4)

- 1. *Major*
- 2. *Minor*
- 3. *Patch*
- 4. Revisió *build*

En el prototip s'ha utilitzat aquest sistema fins a arribar a la versió 0.0.8.1, on s'ha tancat el desenvolupament i s'han començat les proves comparatives.

5.2 Eines i prototipatge

5.2.1 Unity

Pel desenvolupament del producte s'utilitzarà el motor *Unity.* Aquest motor disposa de complements desenvolupats per la comunitat que permeten accelerar el procés de creació i prototipatge del videojoc. És un motor accessible i popular el qual disposa d'ajuda en fòrums en problemes usuals al llarg del desenvolupament d'un prototip amb multijugador.

Aquesta decisió s'ha pres perquè durant el treball es pugui designar més temps a la investigació i implementació del multijugador.

5.2.2 Llibreries

La mateixa empresa que ha creat *Unity* disposa d'una llibreria específicament dissenyada per funcionar amb el seu motor. Han creat exemples que faciliten la implementació del multijugador, i un cop obtingut el coneixement teòric de com funciona un videojoc en línia implementar aquest pot suposar menys dificultós.

Per altra banda, la llibreria *PUN*, una empresa aliena a *Unity,* ha creat una llibreria multijugador pel motor. És una de les llibreries més extenses entre els usuaris, amb diversos jocs en el mercat que l'utilitzen. També ofereixen suport de servidors dedicats, si fos necessari escalar el joc.

El treball implementarà primer *PUN*, i posteriorment les llibreries *Mirror, MLAPI* i *UNet.*

5.2.3 Decisions finals

El prototip s'ha realitzat amb Unity, i les llibreries PUN i *Mirror* en aquest ordre cronològic. S'ha utilitzat models i components de la botiga de Unity per agilitzar el desenvolupament, llistats en els annexos.

Tasques com el moviment de la càmera i generació s'ha fet servir Cinemachine, que permet configurar la posició, angle i "*collision avoidance"* perquè mantingui el personatge enfocat.

Amb les dues versions s'ha fet servir el versionatge mencionat anteriorment. Tots els executables vàlids duts a termes estan numerades amb un fitxer de text que recull els canvis i idees d'aquella actualització.

No s'han emprat servidors dedicats. S'ha explorat la idea, però per la facilitat d'accés i la baixa latència del servidor local se n'ha prescindit.

5.3 Planificació

S'estima que el temps d'implementació de la llibreria sigui curt, però la programació de les solucions analitzades suposi un temps llarg a causa de les iteracions i la recerca que suposarà. En el cronograma següent es mostra com s'ha plantejat el desenvolupament del treball.

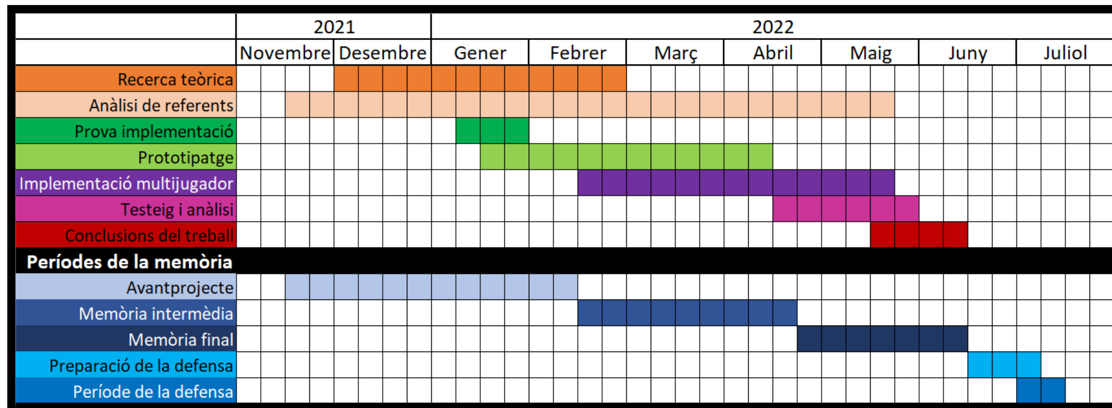

Figura 5.1 Cronograma del treball. Font: Elaboració pròpia

Els referents s'allarguen fins que la implementació del multijugador acabi, ja que apareixeran artefactes i *bugs* que es desconeixen ara mateix, però altres usuaris ja hagin solucionat.

La implementació de prova es va realitzar per tenir un primer contacte i veure les dificultats que sorgien al principi. També va ajudar a comprendre els problemes als quals alguns referents s'havien enfrontat.

5.3.1 Planificació final

La planificació final s'ha vist alterada pels motius que s'havien previst. La implementació de les llibreries, tant PUN com *Mirror*, es van veure dificultades per la falta d'experiència en la temàtica.

El desenvolupament amb PUN s'ha vist afectat per la implementació de codi d'interpolació personalitzat, a causa que no es va trobar una alternativa a utilitzar el *Character Controller* de Unity.

La implementació de *Mirror* es va començar tard per culpa del retard en el desenvolupament amb PUN. Es va haver de canviar elements del joc i l'estructura no era la mateixa que amb PUN, per el que alguns elements no van poder ser reciclats.

La comparativa, anàlisi i redacció de tots els elements dels resultats es van realitzar dins el període de temps de les dues últimes setmanes. Es pot veure reflectits aquests temps dins la Fig. 5.2.

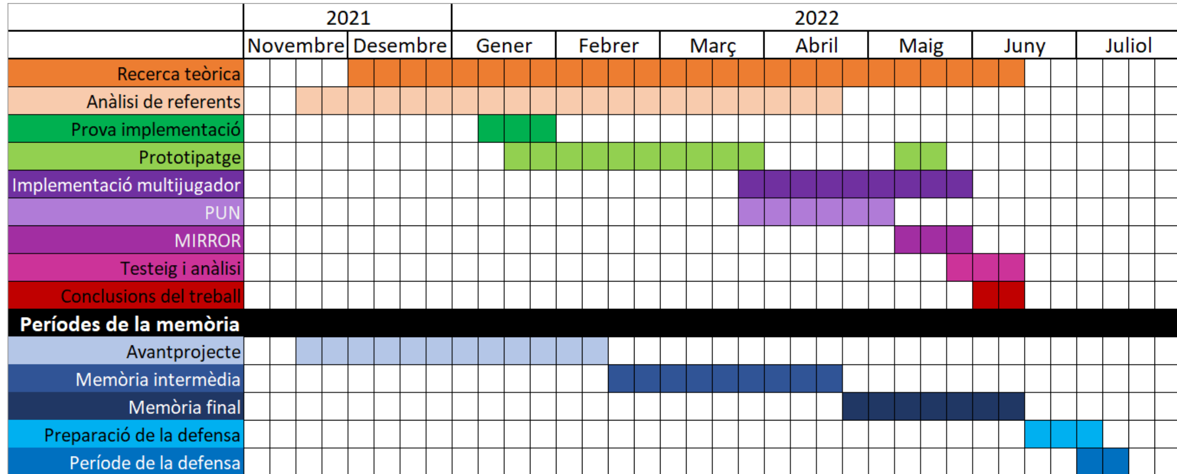

Figura 5.2 Cronograma del treball representant els temps finals. Font: Elaboració pròpia.

6. Anàlisi i resultats

En aquest capítol, es detallen els passos realitzats per obtenir el prototip de cada una de les llibreries escollides.

Primerament, es detalla la llibreria de PUN, la primera que es va implementar a Unity en aquest projecte, seguit de la de *Mirror*. Després de les dues explicacions, es fa una comparativa dels diferents aspectes que tenen les dues llibreries, tant en l'àmbit econòmic com en el tècnic.

6.1 Photon Unity *Network* (PUN)

En aquest apartat es descriuen els passos previs a la implementació de la llibreria que PUN requereix per començar a utilitzar els seus serveis, seguit dels procediments incrementals per aconseguir el prototip.

També es descriuen les solucions aplicades i com s'han programat en sintonia amb aquesta llibreria.

6.1.1 Instal·lació i primers passos

En descarregar *PUN* de la Unity Asset Store, es va configurar la connexió amb el servidor mestre de *Photon*. Part de la configuració es genera automàticament en realitzar la primera connexió, com la identificació del joc. L'altra part s'ha completat manualment, amb les necessitats del projecte.

La regió s'utilitza per cercar només servidors que estiguin localitzats dins la mateixa zona física. Una regió pot encapsular una ciutat, un país o un continent. Un jugador que es connecti a una regió diferent de la que es troba patirà més *lag.* Pel prototip, s'ha fixat la regió a "eu" acrònim d'Europa. Es poden marcar preferències sobre les regions, però si cap s'ha establert, quan el client s'intenta connectar *Photon* automàticament escull la regió amb millor *ping*.

Com que s'utilitzen els servidors externs que ofereix PUN, les opcions de *server, port* i *Proxy server* són irrellevants. Si es volgués albergar el servidor per gestionar

les connexions, s'hauria de configurar amb la nova adreça i ports del servidor personalitzat.

No s'ha modificat el protocol de connexió de PUN durant el desenvolupament del prototip. Es durà a terme proves sobre quin rendiment té cada protocol en un altre apartat del treball.

Finalment, es configura que en la depuració de tota la informació sigui completa, posant les variables "*Network Logging*" i "*PUN Logging*" a "*ALL*" i "*Full*" respectivament.

6.1.2 Els Components de PUN

El *script* que permet a PUN sincronitzar objectes a través de la xarxa s'anomena *PhotonView*.

Aquest *script* s'encarrega d'assignar la propietat i el control dels objectes instanciats dins la partida. Els personatges dels jugadors en formen part.

En carregar l'escena, si el jugador no disposa de personatge, aquest s'instancia i s'atorga la propietat. Quan se sincronitzen les escenes entre tots els jugadors, els personatges no locals es vinculen amb el seu original per poder rebre les dades que s'envien. *PhotonView* recupera tots els scripts que heretin de classes creades per PUN, com *MonoBehaviorPunCallbacks*, que permeten tenir accés a funcions en línia, i les sincronitza amb els originals equivalents.

PUN ofereix scripts per sincronitzar valors com posicions i animacions sense tocar codi addicional. El seu procediment és el mateix, però comporten dos inconvenients majors pel prototip. No permeten l'accés a aquestes dades i s'apliquen forçosament en totes les instàncies sincronitzades. Suposa un problema en aquesta implementació, ja que el component *CharacterController* no permet que la posició s'alteri mitjançant el *Transform*.

Aquest sistema s'utilitza amb èxit per sincronitzar elements que es puguin emular amb precisió en diversos clients, com objectes físics parts d'un puzle.

6.1.3 Mecàniques

Les mecàniques s'activen a través d'input. A diferència del moviment del jugador, són accions que no ocorren constantment, sinó amb menys freqüència.

Estan implementades les mecàniques en el prototip de manera que quan s'activa una tecla, es crida una funció que genera una caixa de col·lisió invisible que cerca altres jugadors. Si en troben un, en retornarà la referència i seguidament cridarà una funció dins aquell *PlayerController.*

Per fer que tots els altres jugadors sàpiguen que s'ha interactuat amb aquell personatge, s'envia una execució remota de codi (*Remote Procedure Call o RPC*). Un RPC és una funció del component *PhotonView*, heretat de la classe *MonoBehaviourPun* que permet anunciar a altres clients que executin certes funcions.

En aquest cas, s'utilitza la referència a l'altre personatge per trametre al seu *PhotonView* l'avís d'executar una funció sobre ell mateix i tots els altres personatges amb els quals estigui sincronitzat.

Com que es disposa de predicció local, també es crida el PlayerController d'aquell objecte no local perquè simuli el mateix moviment.

Totes les mecàniques de llançament d'altres jugadors es fan a través d'aquest sistema.

Altres funcions que es fan a través dels RPC són. Sincronitzar els vestits, el nom del jugador sobre el personatge i la càrrega del nivell quan el *host* comença la partida.

6.1.4 Player character

El PlayerCharacter és el personatge que cada jugador controla dins una partida.

Aquest objecte disposa de diversos components per controlar-lo. Per moure'l i comprovar col·lisions s'utilitza el CharacterController, proporcionat per Unity ofereix informació sobre les col·lisions després d'executar cada moviment i ignora les

físiques d'altres objectes. No es pot moure amb físiques i s'ha de programar el càlcul de moviments un mateix.

El script *PlayerController* és el principal cervell de l'objecte. A partir dels inputs del *PlayerInput* realitza les accions que vol el jugador.

El cervell s'encarrega de gestionar cada personatge, sigui local o no local. Si un personatge és local, interpreta els *inputs* i simula els moviments. Si no és local, es mourà en direcció la posició que el propietari dicti.

Dins la funció *NetworkUpdate* del *PlayerController* s'hi implementen solucions per mitigar l'efecte dels artefactes mencionats en capítols anteriors.

Els artefactes als quals se'ls hi ha aplicat una o vàries solucions són:

- Dessincronització
- Lag
- Predicció local

6.1.5 Compensació del lag i dessincronització

La dessincronització al llarg del desenvolupament sorgia quan un personatge no local es movia més enllà on l'original anava. L'error va aparèixer quan es va aplicar la compensació de *lag*.

Aquest últim consisteix a escalar la velocitat del personatge no local, amb la intenció de reduir el temps que tarda a moure's fins a la posició de l'original. El pensament darrere aquesta idea s'expressa amb el següent exemple i fórmula matemàtica de la velocitat.

$$\Delta x = vt \tag{1}$$

Un personatge local comença a caminar en el frame 0. Camina durant 10 frames (o 166ms a 60fps) i es para. El seu ping és de 5 frames (83ms a 60fps).

El personatge no local que representa aquell primer, rep el moviment al frame 5. Hipotèticament, el ping del jugador baixa a 1 frame, aquest personatge no local

caminarà des del frame 5, quan rep l'acció, fins al frame 10, que rep l'ordre de parar. (Figura 6.1)

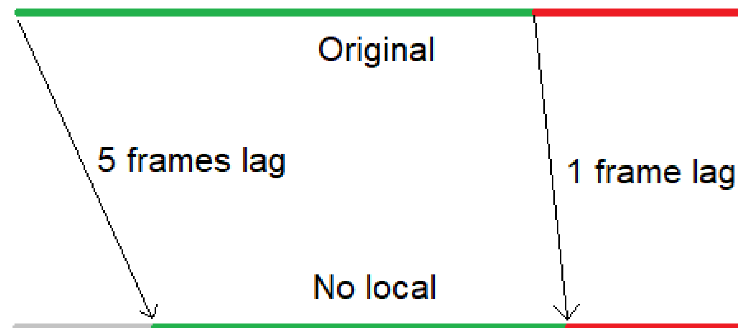

Figura 6.1 Representació visual de la no compensació del lag. Escala irregular. Font: Elaboració pròpia.

El personatge no local haurà caminat durant 5 frames, i l'original 10. Seguint la fórmula de la velocitat de (1), per arribar a la mateixa posició en la meitat de temps, el personatge hauria d'anar al doble de velocitat. És poc escalable en jocs ràpids o amb servidors amb més *ping.*

Aquesta solució escalava la velocitat del personatge a una que fos adient per la compensació del lag. La velocitat que obtenia no es limitava per poder compensar alts valors de *ping.* L'ocurrència en aquestes situacions era un desplaçament curt dels personatges en molt poc temps, dificultant la interacció entre jugadors. Finalment, es va optar per una opció que no modifiqués les propietats preestablertes dels personatges, com la velocitat de salt, la de moviment i la gravetat. Valors que amb el disseny del joc s'han ajustat per una bona experiència de joc.

La solució final d'aquest artefacte consta en fer durar el moviment el mateix que ha durat en l'original. Com es pot observar en la següent Fig. 6.2 l'objectiu és igualar el temps que el personatge no local (barra superior dins la Figura) tarda a realitzar tot el desplaçament que l'original (barra inferior).

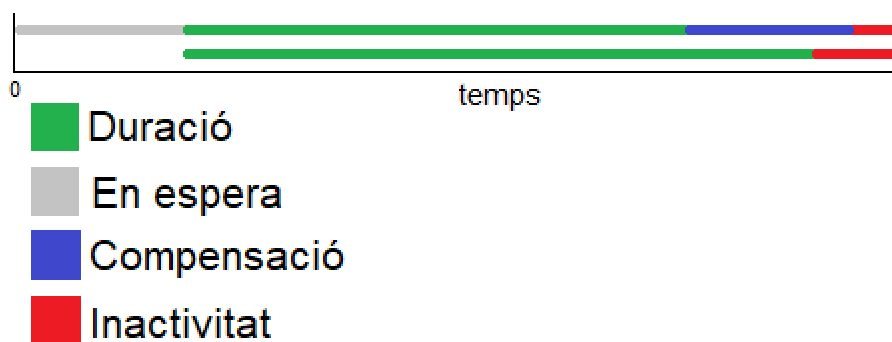

Figura 6.2 Compensació del *lag* utilitzant les dades de la Figura 6.1. Font: Elaboració pròpia.

S'ha aconseguit guardant les dades del personatge original en una cua en format de *struct* anomenada TimedNetworkPosition. En aquesta struct s'hi guarden la velocitat, la posició, la rotació i la inèrcia, juntament amb el *ping* que ha tardat a arribar aquella última actualització, anomenada "duració".

Quan es recupera variable TimedNetworkPosition de la cua, es prepara un temporitzador amb el valor "duració" que emmagatzema. Com que també es retira la variable de la cua, la longitud es redueix en 1. Si no hi ha cap valor dins la cua, no arriba a considerar si ha passat el temps. Es quedarà sempre en aquells valors si no arriba cap altra actualització. (Figura 6.3)

Al moment que arriba una nova actualització des de l'original, la longitud de la cua és superior a 0 i entra dins la comprovació. Si ja s'havia tret un valor de la cua abans i estava planificat activar el temporitzador, s'activa. Aquest temporitzador evitarà que una nova posició sobreescrigui la variable *NetworkPosition*, variable que determina cap on es dirigeix un personatge no local.

Un cop s'acaba el temporitzador, es permet recuperar el següent valor de la cua i iterar el procés.

```
Comprovar si es té control sobre el personatge, i hi ha connexió
            Si es té, comprova si hi ha una nova actualització a la cua.
                        Si es així i s'ha d'activar el temporitzador

                                    Estableix el temporitzador al temps al valor recollit.
                                    Bloqueja el temporitzador.

                        En cas contrari i si ha acabat el temporitzador

                                    Guarda en una variable el primer struct de dades de la cua.
                                    Estableix les variables segons les dades recollides.
                                    Activa la variable de control que activa el temporitzador.
                                    Neteja la struct de la cua.
```

Figura 6.3 Pseudocodi de la funció UpdateNetworkParameters. Font: Elaboració pròpia.

Això permet que el personatge es mogui amb les mateixes característiques que l'original, durant el mateix temps.

En actualitzacions més recents, el temporitzador passa a ser de tipus "float" a tipus "int". I en comptes d'emmagatzemar el temps en milisegons, es fa amb frames. S'aconsegueix dividint el valor del *ping* entre 60 i arrodonint a la baixa. Fa més reactiu el moviment del personatge, reduint el temps que no se li permet recollir una actualització.

Amb aquest sistema la dessincronització és menys present, ja que els moviments s'imiten gairebé a la perfecció.

6.1.6 Predicció local

Per la predicció local, s'utilitza aquest sistema de cues per obligar a un personatge no local a buidar la cua de dades i afegir com a primera el moviment que es vol executar. Es fa ús en mecàniques en què personatge interactua amb un altre. Per exemple en la mecànica d'empènyer.

La implementació consta en buidar la cua i establir el temporitzador d'actualització a 0, perquè instantàniament s'apliqui el moviment que es vol. Això permet a un personatge local veure el resultat de les seves accions abans que el servidor admeti que és vàlid.

Aquesta arquitectura no és de servidor autoritari, sinó de client autoritari. La diferència és que en aquest prototip, el servidor creurà que el client sempre té raó.

Per exemple pot empènyer un personatge que tingui proper, encara que dins la simulació del *host* no sigui prou a prop. El codi s’executarà dins el client, enviarà al *host* de la partida que ell ha realitzat aquella acció correctament, i en tots els clients es reflectirà aquella acció.

6.1.7 Menú principal

Un cop completada la configuració de la llibreria, s’ha preparat la connexió al servidor mestre de PUN i generat *callbacks* per cada esdeveniment que tramet. Aquests *callbacks* estan localitzats dins un script anomenat *Launcher.* Aquest també s’encarrega de gestionar totes les connexions i desconnexions dels *lobbies* i les *rooms.*

Launcher es troba dins l’escena del menú principal, la primera escena que el projecte carrega forçosament.

Tots aquests mètodes disposen d’informació enviada pel *Master* que donarà detalls per cada acció en línia del client. En aquest estat del prototip es fa servir per depurar informació i carregar l’escena que correspon a l’estat del client (el menú principal, la partida o la sala d’espera).

Les primeres versions del prototip disposaven d'una interfície amb un botó que connectava el client amb una sala aleatòria i un espai per posar un nom. La connexió amb el *master* es realitzava automàticament en executar el prototip.

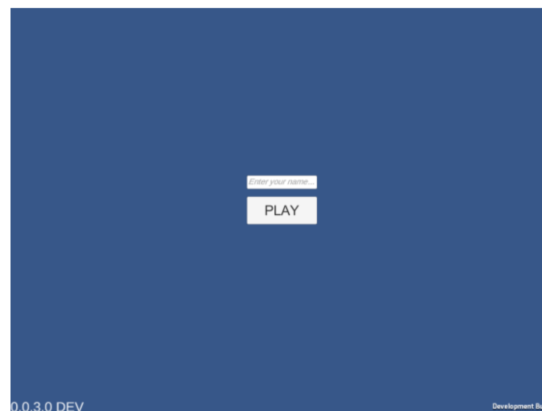

Figura 6.4 Captura del menú principal de la versió 0.0.3.0 DEV. Font: Elaboració pròpia a partir del prototip.

L'inconvenient més gran d'aquesta implementació és el requisit de connexió per accedir a qualsevol part de l'aplicació, aquesta havent-se d'establir en engegar l'aplicació, sense manera de tornar a connectar amb el *master* un cop dins.

El disseny iterat del menú principal disposa de botons per connectar al servidor a voluntat del jugador. El botó apareix quan no es disposa de connexió i el client està al menú principal.

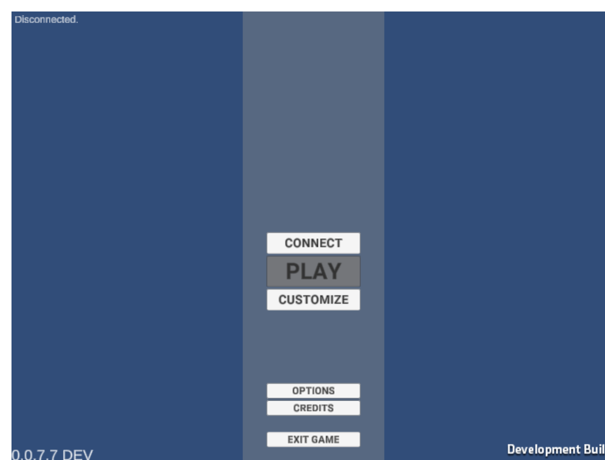

Figura 6.5 Captura del menú principal de la versió 0.0.7.7 DEV. Font: Elaboració pròpia a partir del prototip.

Les altres opcions de menor rellevància per la implementació de la llibreria de *PUN*, però part del desenvolupament del prototip disponible en l'última versió són: l'escena de personalització del personatge les opcions i els crèdits.

El menú de personalització disposa de diferents peces que el personatge es pot equipar i utilitzar el vestit mentre juga en línia.

6.1.8 Cercador de sales

La finestra del cercador de sales disposa de l'antic element per introduir el nom del jugador i poder crear la sala.

Quan un jugador crea una sala, es crea amb les opcions per defecte, i el nom d'aquesta és el nom del jugador més "*'s server.*" Més endavant s'afegirà l'opció de personalitzar la quantitat de jugadors permesos (per defecte 4) i el nom de la sala.

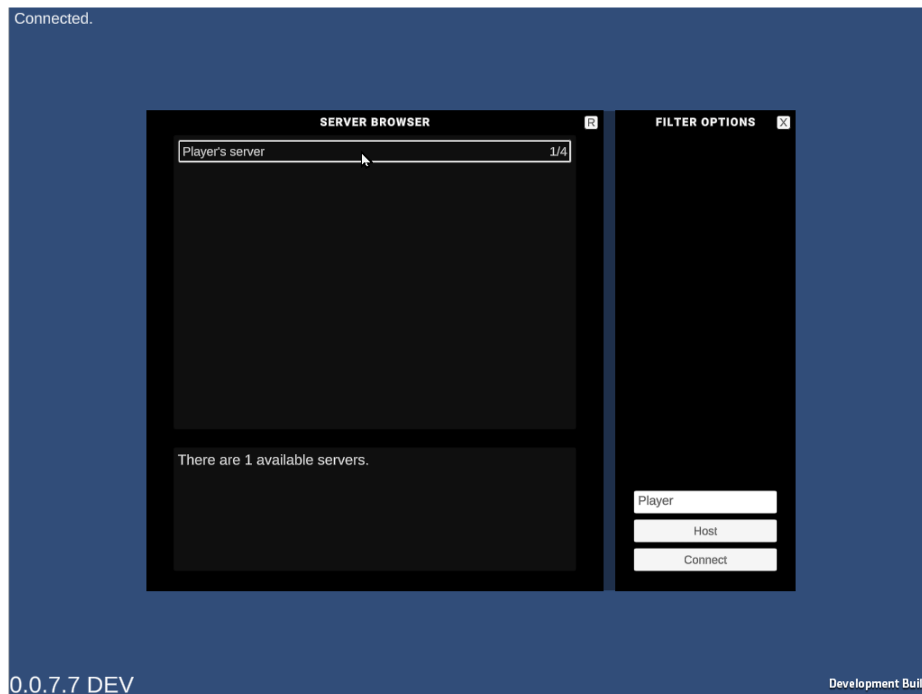

Figura 6.6 Captura de la interfície de cerca de servidors de la versió 0.0.7.7 DEV. Font: Elaboració pròpia a partir del prototip.

Automàticament el client que ha creat la sala passa a ser el *host* de la sala i en el seu joc se li carrega l'escena de "*Lobby*", on es mostren tots els jugadors que hi ha actualment amb els seus vestits.

En ser el *host*, el jugador pot escollir quan començar la partida i no permetre altres jugadors dins la sala.

Figura 6.7 Escena “*lobby”* des del punt de vista del *host.* Font: Creació pròpia a partir del prototip.

6.1.9 Escena *Lobby*

L’escena de *Lobby* és la primera sala que es carrega en generar una partida i serveix de sala intermèdia entre el menú principal i el joc. Aquí els jugadors poden veure’s i interactuar amb les mecàniques de joc.

Aquesta escena disposa d’un element vital, i és el *script PUN_GameManager.* Aquest disposa de la referència al *prefab* del personatge i és l’encarregat d’instanciar-lo quan un jugador s’uneix a la partida. Quan s’instancia, el component *PhotonView* del *prefab* determina el client que l’ha instanciat i li atorga control complet. Cap altre client pot exercir un canvi directe amb *inputs* sobre aquell personatge un cop està assignat.

El mateix script de *PlayerController* disposa d’una referència al component *PhotonView* i pot saber si és el personatge local o no. Si no ho és, es gestiona el moviment a partir de les actualitzacions que es reben a través del servidor de *Photon*. Dins l’apartat del *PlayerController* es detalla com s’ha integrat PUN amb aquest script ja existent per poder utilitzar-lo en una partida multijugador en línia.

Quan un jugador abandona, PUN s’encarrega de destruir el personatge associat a aquell client. Això és cert dins totes les escenes que continguin personatges instanciats a través de PUN amb components *PhotonView*.

6.1.10 Escena jugable

L’escena “*game*” disposa d’un *GameManager* que s’encarrega d’enviar un RPC a tots els clients perquè es transportin a unes posicions predefinides que serveixen com a punt d’inici de la partida.

Dins aquesta escena, es desenvolupa tot el *gameplay loop*, on els jugadors poden utilitzar les mecàniques d’empènyer per interactuar entre si. El seu objectiu és recollir tots els punts que hi ha dins l’escenari, situats a diferents punts del mapa. Els punts estan situats en zones on els jugadors no poden assolir l’alçada de les plataformes per si sols, amb el propòsit que cooperin per aconseguir-los tots mitjançant l’ús de les mecàniques mencionades anteriorment.

En la versió final del prototip, l’escena serveix per validar la implementació de totes les mecàniques i la sincronització d’aquestes a través del multijugador.

6.2 *Mirror*

Dins aquest capítol s’explica el desenvolupament amb la mateixa estructura que s’ha fet a l’anterior apartat, encara que amb *Mirror* no s’hagi seguit el mateix ordre de desenvolupament.

Mirror s’ha implementat gràcies a l’ajut de tutorials de DapperDino (2020) i amb l’ús de la documentació de la mateixa llibreria. La UI, els *prefabs* i l’estructura del “*NetworkRoomPlayer*” i “*NetworkGamePlayer*” s’ha aconseguit gràcies als tutorials, mentre la funcionalitat específica i algunes necessitats variades, com la roba i els noms, les ha desenvolupat el treball.

6.2.1 Conceptes generals i primers passos

Aquesta llibreria funciona amb connexió *IP* directa. Implementa un sistema per instanciar els jugadors que va amb el seu component de *NetworkManager*. Això implica que el buscador de servidors i el gestor de connexions està obsolet i s’ha de reconstruir.

L'estructura de *Mirror* de trucades RPC també funciona diferent. Existeixen diferents atributs que ofereixen executar funcions com a RPC. Aquests són:

- ClientRpc. Si una funció té aquest atribut, només el servidor pot executar-la, i si ho fa, ordenarà a tots els clients que l'executin també.
- TargetRpc. Igual que l'atribut anterior, però en comptes de tots els clients, només un en concret l'executarà.
- Command. Només es pot trucar des d'un client i ordenarà al *host* o servidor que executi aquella funció.

Al reaprofitar l'estructura de PUN, que només hi havia un tipus de funció RPC, aquesta estructura es va haver de canviar. Es va causar algun *bug* durant el procés a causa de la incertesa d'en quin client s'executaven les funcions.

Quan es va tenir clara l'estructura, seguint la Fig. 6.8, perquè un jugador pugui interactuar sobre un altre jugador sense ser el *host*, ha de trucar la funció CmdServerSetInertia(). Si el jugador A vol empènyer al jugador B, el jugador A ha de cridar CmdSetInertia(B, Vector3), on Vector3 és la força i direcció. Això executarà la funció en el servidor, que seguidament remetrà l'ordre a tots els altres clients, executant la funció amb atribut ClientRpc amb referència a B.

```
[ClientRpc]
2 usages  new *
public void RpcSetInertia(Vector3 value)
{
    SetInertia(value);
}

[Command]
1 usage  new *
public void CmdServerSetInertia(PlayerController pc, Vector3 value)
{
    pc.RpcSetInertia(value);
}
```

Figura 6.8 Estructura correcta de trucades client-servidor-client. Font: Elaboració pròpia.

S'ha pogut reutilitzar part de l'estructura que ja es tenia a *PUN*, però canviant totes les instàncies on s'implementaven *RPCs*, realitzant comprovacions d'autoritat prèvies perquè *Mirror* no donés error.

6.2.2 Components de *Mirror*

La llibreria de *Mirror* afegeix diversos components a Unity. Aquests permeten sincronitzar l'estat dels objectes i jugadors amb totes les instàncies dels altres clients.

Els components que s'han utilitzat per el desenvolupament del prototip amb *Mirror* són els següents:

- *NetworkAnimator*: Sincronitza l'estat de l'animació i els paràmetres amb què treballa el component *Animator* base de Unity. L'única excepció és amb els *triggers*, que s'han d'establir manualment amb les funcions *SetTrigger()* i *ResetTrigger()* del component de *Mirror*. Aquestes funcions es truquen des del *script "PlayerAnimatorController"*.
- *NetworkIdentity*: Atorga control de cada objecte i el sincronitza en línia. És el component que cada objecte que s'instancia dins les escenes ha de tenir per poder ser sincronitzat. Disposa d'informació de quin personatge és el local, si és el *host* i la identificació d'aquest.
  En la implementació del prototip, aquest component s'ha afegit als punts que es poden recollir, als jugadors, als gestors de les connexions i al manager de puntuació.

- *NetworkManager*: Aquest component s'ha utilitzat com a base per crear dos *scripts* personalitzats: "*NetworkLobbyManager*" i "*NetworkGameManager"*. Aquest disposa de funcions per gestionar totes les connexions i obtenir informació sobre l'estat d'aquestes. És el centre de control del joc multijugador, per tant, el més important.
- *NetworkTransform*: Sincronitza amb totes les altres instàncies de l'objecte la posició, rotació i escala de l'objecte, així com les translacions que realitza. En la implementació, el personatge de cada jugador implementa aquest component. No requereix codi addicional i funciona correctament amb el component "*CharacterController*" de Unity.

La llibreria ofereix components addicionals que no s'han utilitzat en aquesta implementació.

6.2.3 Mecàniques

Les mecàniques estan executades en la seva completament en el servidor, només en la implementació de *Mirror*.

S'ha canviat a aquest sistema, ja que un client no pot executar codi sobre un altre client directament dins de *Mirror*.

El procediment que es realitza ara és:

1. Es registra l'input en el client.
2. El mateix client comprova si pot dur a terme aquesta acció en l'estat actual.
3. Es genera una càpsula de col·lisió davant del personatge local per comprovar si algú pot rebre la interacció.
4. S'envia l'ordre al servidor mitjançant una funció amb atribut *"command"*.
5. El servidor rep l'ordre i executa una funció amb atribut *"ClientRpc"*, informant a tots els clients del fet que han d'executar aquesta mateixa funció localment.
6. Quan el client rep l'ordre, executa la funció.

En aquesta implementació, l'equivalent als passos anteriors, però per activar la mecànica d'empènyer, és la següent.

1. Es registra l'input d'empènyer en el client A.
2. Es comprova que no estigui a l'aire o estigui realitzant una altra acció.
3. Es genera una càpsula de col·lisió davant del personatge local per intentar trobar un altre jugador.
4. Si es troba un altre personatge (referenciat aquí com a B), s'envia al servidor una ordre per executar la funció de *"CmdServerSetInertia(PlayerController, Vector3)",* enviant com a referència el personatge B.
5. Aquesta funció fa que el client B executi *"RpcSetInertia(Vector3)"*, que mitjançant RPC avisa a tots els observadors perquè executin el següent pas.
6. Quan cada client rep l'ordre, executen l'última funció de la cadena *"SetInertia(Vector3)",* que s'encarrega d'establir la velocitat del personatge B amb el paràmetre de tipus Vector3.

6.2.4 PlayerCharacter

El *PlayerCharacter* s'ha vist modificat per adaptar l'estructura de trucades d'un servidor autoritari. Com que un client no pot causar cap efecte sobre un altre sense autoritat, les funcions d'aquest tipus s'han d'enviar a través del servidor.

Les trucades a les funcions de les mecàniques es fan a través del servidor, tot i que la simulació del client es considera correcta en detectar col·lisions pels llançaments.

El *GameObject* del jugador disposa de dos components nous. El primer és el "*NetworkGamePlayer*", i el segon el "*AccessoryEquipper*".

El primer script s'encarrega de registrar el prefab del jugador dins una llista dins el "*NetworkLobbyManager".* Serveix per dues coses. Una és per tenir un control dels *prefabs* dels personatges instanciats durant la partida. I l'altra com a emmagatzematge de dades del jugador, com el nom i el codi identificador del vestit. Aquestes variables del jugador estan sincronitzades dins la partida, així cada client pot saber quina roba i quin nom tenen els altres jugadors per poder configurar-ho localment.

6.2.5 Compensació de lag i dessincronització

Mirror implementa el seu propi sistema de compensació de lag. El mateix *script* és configurable i permet escollir si se sincronitza tots o alguns valors del *Transform* i si s'interpolen els valors per suavitzar el moviment final.

El que fa el component és guardar cada actualització que envia el propietari i guardar-ho en una *struct*. Cada instància d'aquesta *struct* es guarda en una llista per la seva posterior gestió.

Quan s'ha d'actualitzar la posició de l'objecte, s'interpola a partir del InverseLerp de la diferència del temps entre actualitzacions i el ping que tenia anteriorment el jugador.

Per estalviar amplada de banda (*bandwidth*) el mateix component gestiona si els valors que controla han canviat o no, evitant trametre'ls quan no ho han fet.

No s'ha realitzat cap canvi en aquest àmbit i aquesta part del prototip l'ha integrat *Mirror*.

6.2.6 Predicció local

Mirror no implementa predicció local de base. El treball tampoc l'ha implementat a causa del temps de desenvolupament amb aquesta llibreria.

6.2.7 Menú principal

El menú principal de *Mirror* és el mateix que PUN. A excepció del botó de "*Connect*". *Mirror* no requereix una connexió abans de poder jugar en línia, sinó que la connexió es realitza al moment, a través de la *IP* i el menú que apareix al clicar el botó de "*Play*".

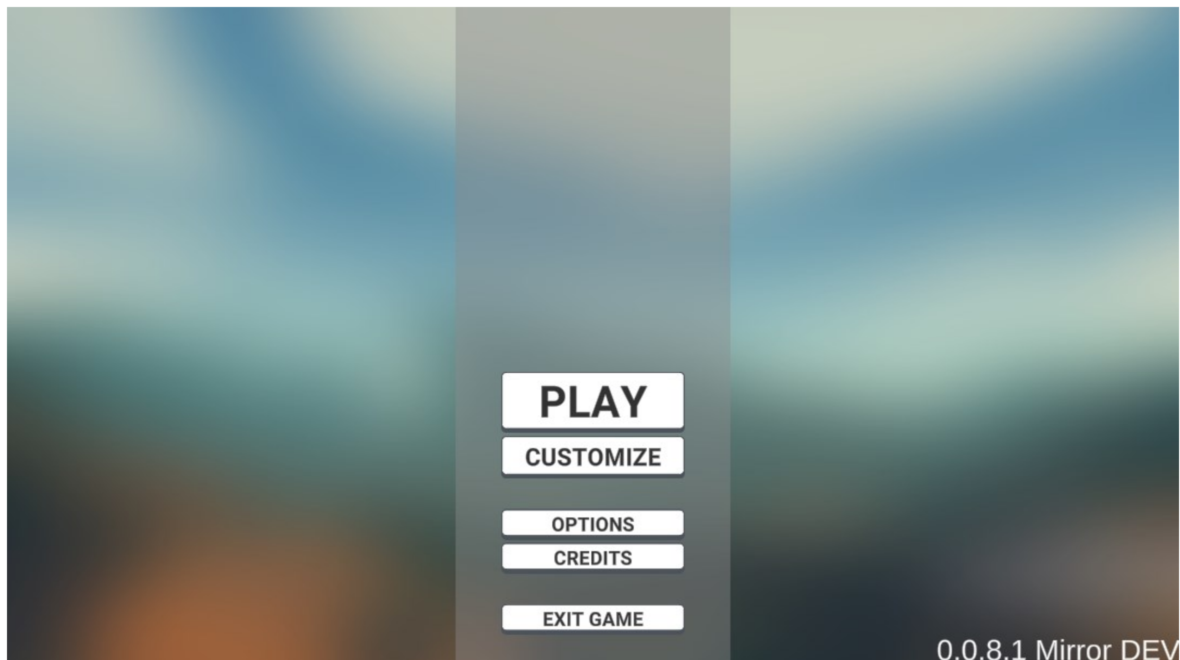

Figura 6.9 Menú principal de la versió 0.0.8.1 *Mirror*. Font: Elaboració pròpia.

En les dues versions es va fer un canvi visual de tota la interfície per donar un aspecte més polit. Es va utilitzar "*Clean Settings UI*" de la "*Unity Asset Store"* per fer tots els elements de la *UI*.

6.2.8 Connexió directa

En pitjar el botó anomenat "Play" dins l'escena principal, apareix una pantalla amb dos espais en blanc, com es pot veure en la Fig. 6.9.

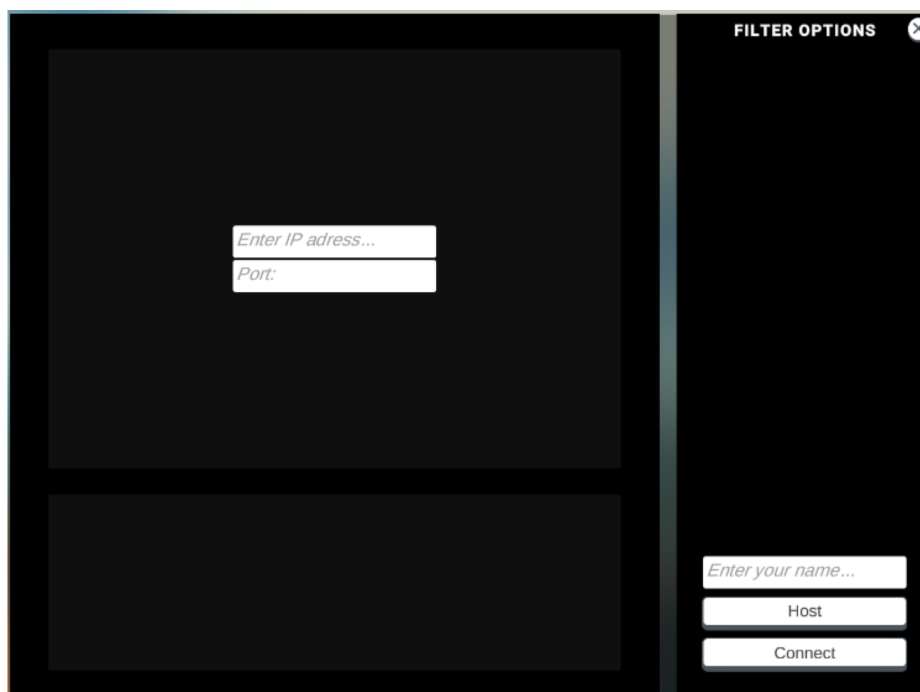

Figura 6.10 Menú de connexió directa. Font: creació pròpia.

El requadre superior és on el jugador pot introduir l'adreça IP del servidor on es vol connectar, i l'inferior és pel port. El valor per defecte de la IP és "localhost" i el port és el predeterminat de Mirror, el 7777.

Si un jugador ho desitja, també pot fer de host del seu propi servidor. Només necessita pitjar el botó "Host" i començarà el procés de creació del servidor.

Si un host crea un servidor correctament o un client s'uneix a un, li apareixerà una pantalla mostrant quants jugadors hi ha actualment, i si estan llestos o no, com dins la Fig. 6.10.

Si el jugador és el host, li apareixerà un botó que li permetrà començar la partida. Un cop una partida comença, no s'hi permeten nous jugadors.

6.2.9 Pantalla de la sala

Quan apareix la pantalla de la sala, el joc ja està en línia. Si és un servidor, comença a escoltar noves connexions a través de la *IP* i el port. Si és un client, ja està sincronitzant l'estat de les seves preferències (nom i roba) pel seu posterior ús.

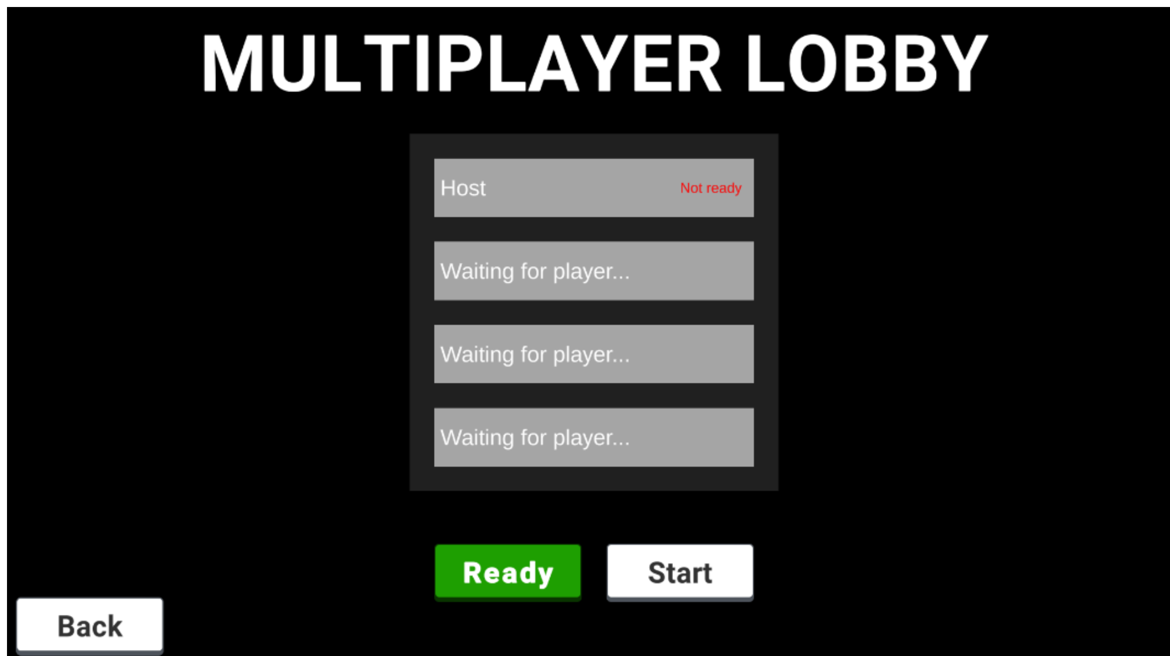

Figura 6.11 Pantalla de la sala multijugador. Font: Elaboració pròpia.

El menú que apareix en pantalla en la Fig. 6.10 s'instancia i s'activa quan es crea una sala o quan un jugador s'uneix a la partida.

Quan un jugador està llest, només ha de pitjar el botó "Ready". El que fa el botó és cridar una funció en el servidor que canvia la variable IsReady de tipus *bool*. El servidor quan rep aquest canvi, té associada una funció a través de l'atribut SyncVar

6.2.10 Escena del joc

L'escena del joc disposa de diversos elements que s'han canviat per adaptar-los a l'estructura de Mirror.

Primer es van adaptar els punts de color que s'han d'aconseguir en tot el nivell. Es va reordenar les trucades de manera que quan qualsevol jugador col·lidís amb un, el servidor sempre rep la trucada que s'ha d'afegir un punt. Un cop el servidor fa el canvi, avisa a tots els clients de la nova puntuació, a través d'un *hook* dins l'atribut *SyncVar.* Quan la variable amb aquest atribut canvia, automàticament s'executa la funció que està "*hooked"* (associada) al canvi, enviant com a paràmetres el valor nou i l'antic.

En el cas del treball, aquesta funció fa saltar una *Action*<int, int>. Aquesta acció pot emmagatzemar funcions que com a paràmetres acceptin dos variables de tipus "int". I quan salta, s'executen les funcions de la UI que reben *m_CurrentScore* i *m_MaxScore* i actualitzen l'apartat visual de tots els clients en sincronia amb el servidor.

L'altre canvi, són el "*SimpleGameManager"*, que conté una llista de punts d'aparició que es poden utilitzar per transportar els jugadors quan es carrega l'escena. Aquest "*SimpleGameManager"* funciona localment, amb l'excepció que el punt l'escull sempre el servidor. Així mai apareixeran dos jugadors en la mateixa posició.

6.3 Implementacions addicionals

6.3.1 Personalització

A part del botó de "*Play*" al menú principal, el prototip també disposa d'un altre botó: "*Customize*". Aquest botó carrega una escena diferent on el jugador pot personalitzar, sense connexió requerida, el seu personatge. Pot escollir guardar el conjunt de roba amb un nom de fins a 16 caràcters, que es guardarà en un fitxer dins la seva carpeta de Documents. Aquest fitxer persisteix durant sessions i l'usuari pot fer servir conjunts creats durant amb anterioritat.

Aquesta secció també inclou 2 sets per defecte, el "Knight", amb armadura de cavaller, i "Empty", que com el nom indica, no té cap peça de roba equipada.

El jugador disposa de fins a 13 categories d'on escollir diferents elements per equipar. Cada una conté diferents peces per poder personalitzar al complet el conjunt.

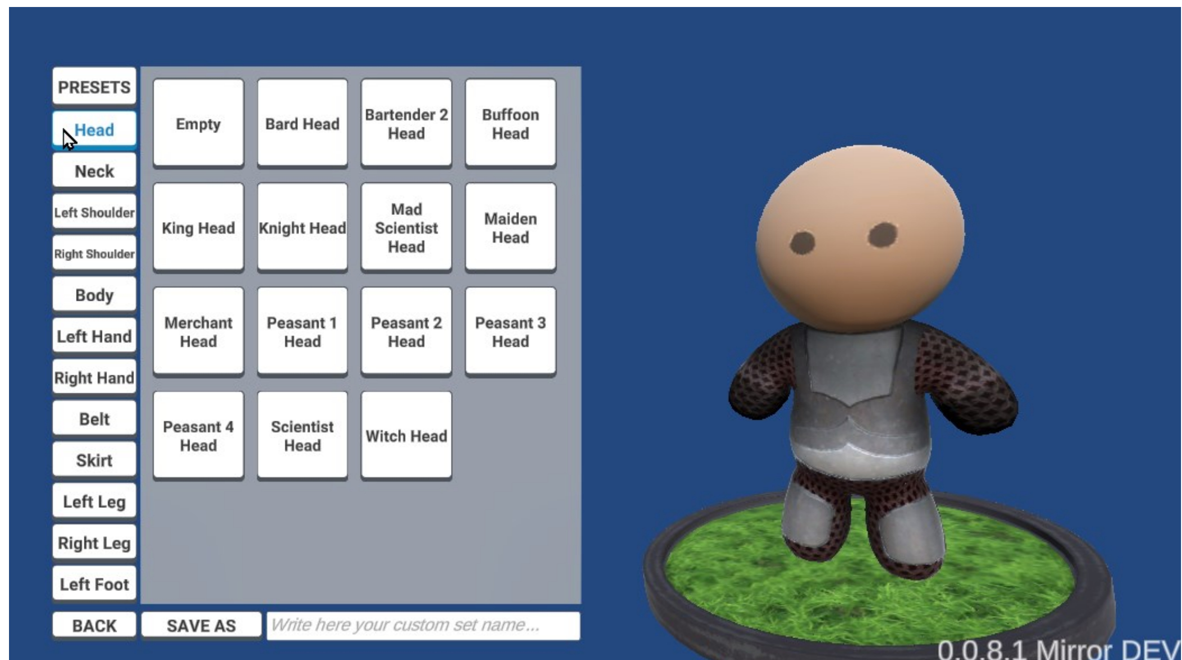

Figura 6.12 Menú de personalització amb la categoria de "*Head*" desplegada. Font: Elaboració pròpia.

Cada categoria es genera a partir d'un fitxer JSON que conté entrades per cada objecte. Cada botó és un *prefab* al qual se li assigna un número identificador. Quan l'usuari en clica un, s'envia l'ordre d'equipar aquell objecte en el personatge.

Aquests identificadors corresponen a objectes guardats dins el fitxer JSON, que funciona com a base de dades persistents. Els objectes es guarden amb l'estructura de dades mostrada en la Fig. 6.12 i una entrada dins la base de dades és com la Fig. 6.13.

```
public class Item
{
    public int itemID;
    public string itemName;
    public string itemDescription;
    public string itemSlug;
    public Slot itemSlot;
    [NonSerialized] public GameObject itemPrefab;
    [SerializeField] public string resourcePath;
```

Figura 6.13 Dades que conté un "*Item*". Cada instància emmagatzema informació sobre una peça de roba. Font: Elaboració pròpia.

```json
{
  "itemID": 601,
  "itemName": "Knight Head",
  "itemDescription": "",
  "itemSlug": "knight_head",
  "itemSlot": 1,
  "resourcePath": "Gear/Knight/Knight_Head"
},
```

Figura 6.14 Entrada individual dins la base de dades en JSON. Font: Elaboració pròpia.

Aquesta llista s’ha generat automàticament a partir de tots els fitxers dins la carpeta de “Resources/Gear/”. També permet actualitzar-la amb nous elements afegits dins la carpeta, mantenint les entrades antigues i afegint les noves sense perdre informació.

6.3.2 Regions

Durant el procés d’implementació de PUN es va intentar filtrar els servidors per regió. Es va afegir una configuració dins el menú d’opcions, a la categoria “Game”, que permetia seleccionar una de dues regions. Es van seleccionar les dues que oferien millor *ping:* Europa i Rússia.

Un cop Photon estava desconnectat permetia canviar de regió, per filtrar els servidors que estaven localitzats dins una en concret. Aquesta implementació no va funcionar i es va apartar per no interrompre el desenvolupament.

El menú de selecció ha quedat implementat dins l’última versió de PUN, però tota funcionalitat se li va desactivar.

6.4 Comparativa entre implementacions

En aquest apartat es realitza una comparativa entre les dues llibreries implementades.

Primer es comparen utilitzant els resultats dels subapartats anteriors, després s’exposen els plans de pagament i finalment els resultats interpretats de les simulacions de connectivitat.

6.4.1 Comparativa General

Dins aquest subapartat es mostren en forma de taula les diferències entre les dues llibreries. S’han fet servir de referència documents i taules de Mikson el 2019, CTO de Futuclass i desenvolupador de videojocs des de 2015. Les taules de Mikson es troben en les referències. Les taules que es veuen a continuació són d’elaboració pròpia a partir de les categories que Mikson llista en els seus documents.

Una de les majors diferències entre les llibreries és la funció de *host* que Photon no té. Photon estableix una simulació com a mestre per tenir una simulació autoritativa, mentre que *Mirror* té un client que fa de servidor.

El problema amb *Mirror* és que de base, si el *host* perd la connexió o tanca l’aplicació, la partida acaba i tots els jugadors són retornats al menú principal. Segons Mikson dins les seves taules el 2019 es pot implementar un sistema de migració de *host.* Photon realitza el canvi de mestre quan es desconnecta l’actual. És més senzill pel fet que no és qui manté les autoritats dels clients i no gestiona les connexions, sinó que ho fa un extern.

	Mirror	Photon
Arquitectura	Client – Servidor	Clients connectats entre sí amb un mestre.
Integració a Unity	Components de *Mirror*. Network*Identity* i NetworkBehaviour com a base de la implementació.	Configuració de la llicència a priori. Regió, transports i llicència. Utilitza PhotonView per gestionar cada objecte.
Autoritat màxima	Servidor dedicat	Client mestre.
Migració de Host automàtica	No. Requereix implementació.	Inclòs, fet automàticament per Photon.
Connexió entre diferents versions del client.	Senzill d'implementar, el host hauria de comprovar-ho a l'acceptar un client.	Automàtic. Una variable determina la versió i la comparativa es fa automàticament.
Callbacks	S'afegeixen per herència.	S'afegeixen per herència i interfícies.
Codi font	Codi obert.	Només una part està oberta, la d'alt nivell.
Multiplataforma	Entre totes les plataformes segons usuaris de fòrums-	Entre totes les plataformes, confirmat per desenvolupadors.

Taula 6.1 Comparativa general entre *Mirror* i PUN. Font: Elaboració pròpia a partir de Mikson, 2019.

Mirror permet una millor gestió dels servidors privats, en permetre que un client faci de servidor. Photon no ho permet, i tots els servidors han d'anar a través dels seus serveis. Això impedeix implementar servidors fora de línia i en LAN. Dins la Taula 6.2 es poden observar amb més detall les comparacions de les dues llibreries.

	Mirror	PUN
Hosting	Dedicated Server	Photon Cloud ho auto gestiona.
Transport Layer protocols	KCP (UDP personalitzat per menor lag), Telepathy (TCP) UNet UDP, Ninja.WebSockets.	UDP,TCP,Websockets escollibles dins la configuració de Photon.
Dedicated Servers	Implementat de base. Permet crear un client sense personatge, que només actuï de servidor. Són funcions diferents, StartServer i StartHost.	Photon permet pagar per utilitzar els seus servidors com a dedicats. Més informació dins l'apartat de preus de les llibreries.
Listen Servers	Sí, el client també és el servidor.	Sí, però el client no és servidor, sinó el mestre amb la simulació autoritària.
Peer-To-Peer	No permet l'arquitectura de P2P	No permet l'arquitectura de P2P
Local Area *Network* (LAN)	Sí, permet connectar a "localhost" o a la *IP* del host dins la xarxa.	No ho permet, Photon obliga a totes les connexions que passin a través dels seus servidors.
Mode fora de línia.	Sí, amb funcionalitat per comprovar si el client està connectat.	Sí, amb funcionalitat per comprovar si el client està connectat.
Servidor sense renderitzat gràfic. (Headless)	Sí, es permet personalitzar la implementació.	No, els servidors són gestionats per Photon.
Connexió *IP* directa	Sí, totes les connexions per defecte es fan a través de la *IP* i el port.	No, es realitza per sales o a través d'un buscador de sales.
Descobriment de servidors	No, però es pot implementar un servidor extern per gestionar les connexions.	Sí, els clients reben una llista de servidors de l'aplicació disponibles.
Descobriment de servidors LAN	Sí. Explora les connexions disponibles LAN, segons *Mirror*.	No. LAN no està permès.

Taula 6.2 Comparativa de les funcionalitats de *hosting* que ofereixen les dues llibreries. Font: Elaboració pròpia a partir de Mikson, 2019.

	Mirror	PUN
Compensació de lag	No, s'ha d'implementar manualment.	No, s'ha d'implementar manualment.
Predicció local	No, s'ha d'implementar manualment.	No, s'ha d'implementar manualment.
Moviment autoritari	Sí, el component de NetworkTransform permet establir que el moviment del client sigui autoritari.	Sí, segons la taula utilitzada per fer aquesta.
Interpolació / Extrapolació	Sí, però funciona sempre que es faci servir el NetworkTransform.	Sí, però funciona sempre que es faci servir el PhotonTransformView

Taula 6.3 Comparativa entre les solucions que les llibreries implementen per defecte. Font: Elaboració pròpia a partir de Mikson, 2019.

Mirror i PUN no disposen de cap mètode per compensar el lag o predir el moviment del client. Aquest treball va implementar aquests elements en el prototip amb PUN, allargant el desenvolupament, mentre que *Mirror* implementava un component que permetia moure un *GameObject* amb *Transform* i *CharacterController* de Unity alhora. Amb aquest component, el treball no es va veure obligat a implementar el mateix que a PUN.

Per projectes amb més fidelitat en la sincronització del moviment, seria necessari disposar d'unes funcions d'interpolació i extrapolació, per predir el moviment i generar-ne en cas de *packet loss.*

6.4.2 Plans de pagament

En aquest subapartat es detallen els preus de les llibreries implementades en aquest treball.

En aquest cas, *Mirror* és de codi obert i completament gratuït.

Per altra banda, PUN disposa de diferents preus segons la quantitat d'usuaris concurrents (CCU). Dins la Taula 6.1 es detallen les ofertes de PUN. Els preus s'han obtingut de la web oficial de Photon.

CCU	Preu	Trànsit de dades	Condicions Especials
Fins a 20 CCU	Gratuït	60 GB trànsit al mes.	Màxim de 16 jugadors per sala. No ampliable.
Fins a 100 CCU	95$ un any	0.3 TB trànsit al mes.	Limitat a una sola compra
Fins a 500 CCU	95$ al mes	1.5 TB trànsit al mes	-
Fins a 1000 CCU	185$ al mes	3.0 TB de trànsit al mes	-
Fins a 2000 CCU	370$ al mes	6.0 TB de trànsit al mes	-
Escalable. Fins a 2000 CCU + extra per cada usuari per sobre de 2000	580$ al mes + 0,29$ al mes per CCU	6.0 TB de trànsit al mes.	Màxim de 50.000 CCU
Pla personalitzat	S'estableix usuaris i preu amb l'empresa segons necessitat	-	Aplicable a jocs il·limitats, suport prioritari, servidors dedicats, plugins i altres serveis

Taula 6.4 Taula de preus de PUN v2 de Photon. Font: elaboració pròpia a partir dels valors de la web oficial de Photon.

Totes les categories tenen un límit de 16 jugadors per sala i un màxim de 500 missatges per segon per cada sala existent. Només les categories que paguen poden ampliar aquestes quantitats.

Per altra banda, Photon ofereix un servei d'allotjament de servidors. *Mirror* no ho fa, però dins la seva documentació ensenya com preparar Unity per fer de servidor i pujar-lo a un servei d'allotjament de tercers.

Pel treball, el valor en aquest cas s'ha trobat en la versió gratuïta, que ha permès fer les proves de les dues llibreries. *Mirror* és completament gratuït, però requereix

més coneixement per implementar, PUN extreu feina i complicacions, apujant el preu segons el volum esperat d'usuaris.

6.4.3 Simulacions de connectivitat

En aquest subapartat es mostren els resultats de les proves de xarxa. Estan categoritzades en taules, i cada prova representa una combinació de tres elements que afecten la qualitat de la connexió. Aquests són: la latència, el *jittering* i el *packet loss.*

Aquestes proves s'han realitzat amb eines que les mateixes llibreries implementen. Photon i *Mirror* disposen de components, anomenats "Photon Lag Simulation GUI" i "Latency Simulation" respectivament, que permeten simular els tres elements mencionats en el paràgraf anterior.

Al costat de cada entrada de les taules hi ha un comentari. Aquest és subjectiu segons el que s'ha vist alterat respecte el comportament normal del joc, definint com a normal, el fet de jugar sense cap aquestes eines activades.

Dins la Taula 6.2 es troben els valors aplicats en una sessió amb condicions de connexió òptimes. Correspon al prototip implementat amb *Mirror*.

Latency	Jitter	Packet Loss	Comentari
0ms	0%	0%	Funcionament normal del joc
50ms	0%	0%	Funcionament normal, l'efecte de la latència és visible al observar varis clients alhora.
100ms	0%	0%	Algunes accions son visiblement retardades al comparar els clients, podria alterar la jugabilitat al moment d'utilitzar les mecàniques.
150ms	0%	0%	Els llançaments es veuen retardats, es nota el temps que el servidor tarda en confirmar algunes accions i la falta de

			predicció local fa que el jugador que realitza el llançament el vegi després que el qui ho rep.
200ms	0%	0%	El mateix que el comentari superior i els jugadors es poden ficar dins un l'altre si es mouen alhora cap al mateix lloc.
250ms	0%	0%	A partir d'aquest valor de latència ja es considera poc jugable, les animacions no concorden amb les accions i els llançaments es reben des de posicions incorrectes.
0ms	50%	0%	No es nota cap alteració en la jugabilitat.
50ms	50%	0%	En algun moment el joc corregeix un moviment durant la partida, però no afecta de manera greu a la jugabilitat
50ms	100%	0%	Tots els paquets reordenats causa que els jugadors es reposicionin habitualment. En alguna ocasió ha causat dessincronització.
100ms	100%	0%	Causa dessincronització quan diverses accions s'executen seguides, sigui moviment, interaccions o salts. Els personatges es transporten d'una posició a una altra i a vegades tarda a actualitzar. Les rotacions a vegades es dessincronitzen. El joc es considera no jugable.
0ms	0%	2%	Aquest valor és el que Photon expressa en la seva web que no supera l'internet d'avui en dia. Succeeix alguna dessincronització amb la rotació, però no es veu cap ocurrència addicional
0ms	0%	10%	Un packet loss del 10% causa posicions i rotacions incorrectes que no afecten greument la jugabilitat. Tot i que es poden perdre accions importants durant la partida, es considera jugable.
0ms	0%	20%	Un packet loss del 20% causa posicions i rotacions errònies que afecten directament a la jugabilitat. Es pot interactuar amb un client dessincronitzat en una posició falsa.

50ms	0%	2%	Ocasionalment, el personatge es congela a mig moviment i es reincorpora en la posició correcta.
50ms	50%	2%	Ocasionalment, s'han de corregir la posició i rotació dels personatges. No causa greus efectes en la jugabilitat.
50ms	50%	5%	El moviment s'interpola, en alguns casos sense informació, diferint notablement de la simulació original. Causa efectes visibles en la jugabilitat.
100ms	50%	20%	La combinació dels tres valors relativament alts causa un gran efecte negatiu sobre la jugabilitat, veient a altres jugadors transportar-se entre posicions i congelant-se a mig moviment. No es considera jugable i és aconsellable tancar la connexió.
200ms	100%	5%	Aquest és un cas extrem que combina les tres categories. Els efectes combinats causa que el personatge es transporti constantment i les accions es realitzin des de posicions incorrectes. No té temps a interpolar per corregir la posició que vindrà perquè les canvia per altres antigues sense coherència. No es considera jugable, i és aconsellable tancar la connexió.

Taula 6.5 Taula de proves realitzada amb una eina per afegir problemes de xarxa de forma fictícia. Font: Creació pròpia a partir del prototip.

Dins la Taula 6.3 es troben les proves realitzades en una sessió amb un servidor de *Photon* i el client de PUN. A causa del servidor extern, les proves no s’han fet sota condicions òptimes, i s’ha de considerar que pateixen d’una latència per defecte de 65(+/- 7) ms. Aquesta latència no està representada dins la Taula 6.3.

Es ressalta que la implementació s’ha desenvolupat en aquest treball en base les guies que *Photon* ofereix. El codi de la interpolació del moviment és del mateix treball (descrit dins l’apartat de resultats) i s’ha iterat durant molt menys temps que la de *Mirror*, que descendeix de UNET (2015), una llibreria creada per professionals del sector.

Latency	*Jitter*	*Packet Loss*	Comentari
0ms	0%	0%	Amb una latència afegida nul·la el prototip és estable i es pot jugar.
50ms	0%	0%	La implementació personalitzada a Photon no és estable al incrementar la latència. S'encadenen tots els moviments però degut que s'escalen les distàncies segons la latència, els moviments tarden massa temps en executar-se i es mouen més del compte. Valors majors de 50 ja causen enormes problemes en la sincronia del joc.
0ms	50%	0%	Els personatges llisquen, però utilitzen posicions acceptables que corregeixen habitualment. És jugable amb una experiència alterada.
50ms	50%	0%	Els personatge llisquen entre posicions i es transporta varies vegades quan ha d'arribar a una nova posició. No ofereix una bona experiència i no es considera jugable.
0ms	100%	0%	Els paquets desordenats causen desincronització i moviments que no pertoquen. El joc no ofereix una bona experiència, tot i que podria ser jugable, subjectivament més que l'anterior prova.
50ms	100%	0%	Els personatges llisquen i tarden a assolir les posicions correctes. Es mouen impredictiblement fins a arribar a l'última posició. El joc no es considera jugable.
0ms	0%	2%	Es dessincronitza la rotació ocasionalment. La interpolació pot arribar a corregir errors tan petits.
0ms	0%	5%	Es dessincronitzen *events* ocasionalment. Estats del jugador com mecàniques no s'activen correctament en altres clients.

0ms	0%	10%	Similar al 5%, no es dessincronitza el moviment, però sí que l'estat del jugador. La interpolació aconsegueix corregir la falta d'informació.
0ms	0%	20%	Similar al 10%, no es dessincronitza el moviment, sí l'estat del jugador, però quan la interpolació no assoleix corregir el moviment, s'activa part del codi que força una posició correcta. Visualment, és brusc i pot afectar l'experiència del joc.
0ms	0%	33%	El personatge es transporta a diferents posicions habitualment, fa canvis sobtats de moviment o canvia el seu estat bruscament. No es considera injugable, però una pèrdua de dades així no és acceptable en jocs on la integritat és vital.
0ms	0%	50%	Els moviments del personatge són incorrectes, es transporta constantment. Un *packet loss* tan alt també ha comportat una desconnexió forçada per part de *Photon*. No és jugable..
0ms	0%	100%	PUN desconnecta al jugador automàticament quan no rep cap senyal durant uns segons.
50ms	0%	2%	L'efecte de la latència i el packet loss causa que el personatge es mogui erròniament i massa tard com per mantenir una jugabilitat fluida.
50ms	50%	2%	Aquesta última prova combina els tres elements. El resultat és una combinació de tots els problemes que cada un causa, exagerats per l'efecte dels altres. Visualment, el personatge es mou tard, de manera errònia i utilitza interpolacions errònies constantment.

Taula 6.6 Taula de proves realitzada amb una eina per simular problemes en la connexió dins el prototip amb. Font: Creació pròpia a partir del prototip.

Aquestes proves permeten observar quins efectes causen els elements de les taules anteriors. Amb els resultats, es podrà millorar la implementació del treball en el futur, centrant els esforços en els problemes detectats.

7. Conclusions i reflexió

El treball ha aconseguit prototipar un joc de plataformes 3D cooperatiu de fins a 4 jugadors. Ha estat un desenvolupament curt, basant el disseny del joc en interaccions entre clients del multijugador. Introduint mecàniques d'empènyer o d'agafar altres jugadors ha permès examinar l'estructura de les trucades client-servidor quan els clients no tenen autoritat sobre altres jugadors. Aquest disseny ha requerit un aprenentatge fora de l'esperat, necessitant canviar l'estructura que del joc quan no tenia multijugador.

Un cop completades aquestes reestructuracions, es va implementar PUN. Amb aquesta llibreria van sortir problemes de compatibilitat, requerint la sincronització de variables i interpolació personalitzada. Aquest procés va allargar el temps de desenvolupament, que encara podia seguir dins els marges previstos. Un cop completat, però, la jugabilitat no era del tot fluida, i es va decidir afegir la predicció local del moviment i a també la compensació del *lag* dins la interpolació. Totes aquestes implementacions van finalment endarrerir el desenvolupament, desorganitzant el cronograma.

Mirror no ha donat problemes en afegir-lo el projecte, ja que el codi de la interpolació ja el portava de base i l'aspecte més important és que funcionava a la primera amb el CharacterController del personatge.

Des del punt de vista final, tot i que aquesta implementació hagi portat la major part del temps dedicat a PUN, el resultat ha estat positiu i la interpolació té punts forts que eviten problemes amb el *jitter* i -parcialment- el *packet loss.* Per futurs projectes s'ha arribat a la conclusió de crear un controlador pensat amb la idea de network primer, per estalviar tots aquests processos aliens a la creació del videojoc.

Respecte a l'objectiu d'implementar diverses llibreries multijugador i adaptar les solucions, es va decidir no intentar implementar-ne més. Aquesta decisió es va prendre quan va començar el desenvolupament amb *Mirror*. Els motius principals van ser a causa dels retards, i perquè *Mirror* és l'evolució de *UNet*, amb aparentment millor rendiment.

Tot i això, es va provar d'implementar les solucions personals del treball de PUN a *Mirror*. El resultat ha estat negatiu i no s'ha pogut completar l'objectiu. El motiu és que el codi personalitzat de PUN no funcionava a *Mirror*, necessitant una estructura diferent. Es recorda que PUN funciona amb un servidor de l'empresa extern i un personatge és el mestre, mentre que *Mirror* disposa de *host*, on el servidor que també és un jugador. També s'ha utilitzat el NetworkTransform de *Mirror* per moure als personatges, en comptes d'utilitzar la interpolació personalitzada. En aquest cas, el motiu ha estat que el component ja realitzava la interpolació automàticament, i solucionava el problema del moviment a través dels components *Transform* + *CharacterController*. Aquest fet va permetre agilitzar la implementació de *Mirror* i centrar l'esforç en la reestructuració i en menú de la partida.

Com a punt final, el creador de personatges ha estat un afegit per donar vida als personatges del multijugador. Inicialment, va ser per identificar els personatges dins la partida i només es podien seleccionar conjunts de roba predefinits, com la versió de PUN 0.0.5.0 que durant la partida es podia canviar un jugador de roba amb un senzill menú. Més endavant es va considerar com a afegit valuós, i es va implementar menús, més roba i funcionalitats abans d'entrar a la partida, fet que facilitava una mica la sincronització. Ha estat relativament poc temps en desenvolupament i es creu que ha aportat un aire més polit al prototip.

Pel que fa a les llibreries, *Mirror* ha costat més de comprendre en implementar, però l'arquitectura de trucades, la interpolació amb variables exposades i tot el codi obert han permès un millor aprenentatge pel futur. *PUN* no es descarta per implementar de manera ràpida i senzilla un prototip, sempre que es comprovi que els elements existents siguin compatibles i no s'hagi de fer grans canvis.

En conclusió, el projecte ha estat ambiciós en intentar implementar el sistema d'interpolació personalitzat. És un apartat al qual se li ha dedicat massa temps, desorganitzant el cronograma i impedint que es complissin altres objectius. Hagués estat millor implementar un controlador de personatge que s'adaptés a les necessitats de la llibreria, i no haver-hi reciclat un controlador per intentar estalviar temps.

8. Referències

8.1 Bibliografia

Britannica, T. Editors of Encyclopaedia. (31 / Agost / 2018). *protocol.* Recollit de Encyclopedia Britannica: https://www.britannica.com/technology/protocol-computer-science

Carmack, J. (16 / August / 1996). *Fabien Sanglard's Website*. Recollit de QuakeWorld logs of august: https://fabiensanglard.net/quakeSource/johnc-log.aug.htm

CS as fast as possible - FNScene. (8 / Setembre / 2020). *The fastest way to BOOST (Dust2, Mirage, Cache, Inferno) | CS afap*. Recollit de Youtube: https://youtu.be/wIJFM2J88wI

DapperDino. (9 / Març / 2020). *How To Make A Multiplayer Game In Unity - Lobby - Readying Up*. Recollit de YouTube: https://youtu.be/WMJS7sVp2FQ

DapperDino. (13 / Març / 2020). *How To Make A Multiplayer Game In Unity - Spawning Players In*. Recollit de YouTube: https://youtu.be/s2ypWi553nY

Darby, J. (2012). *Wizards and Warriors: Massively Multiplayer Online Game Creation.* Cengage Learning.

Donovan, T. (2010). *Replay The History of Video Games.* Great Britain: Yellow Ant.

Dunn, C. (2012). Guild Wars 2: Programming the Next Generation Online World. *GDC.* ArenaNet.

Epic Games. (2022). *Download Unreal Engine*. Recollit de Unreal Engine: https://www.unrealengine.com/en-US/download

Epic Games. (2022). *Networking and Multiplayer*. Recollit de Unreal Engine: https://docs.unrealengine.com/5.0/en-US/networking-and-multiplayer-in-unreal-engine/

Fiedler, G. (24 / Febrer / 2010). *Networking for Game Programmers*. Recollit de Gaffer on Games: https://gafferongames.com/post/what_every_programmer_needs_to_know_about_game_networking/

Fiedler, G. (5 / Gener / 2015). *State Synchronization*. Recollit de Gaffer on Games: https://gafferongames.com/post/state_synchronization/

Glazer, J., & Madhav, S. (2015). *Multiplayer Game Programming.* Addison-Wesley.

Graetz, J. (1981). The origin of Spacewar. *Creative Computing*.

Hamari, J., & Sjöblom, M. (2017). What is eSports and why do people watch it? *Internet Research*, 211-232.

House, B. (8 / Setembre / 2020). *Choosing the right netcode for your game*. Recollit de Unity Blog: https://blog.unity.com/technology/choosing-the-right-netcode-for-your-game

IR Media. (2022). *What is network packet loss?* Recollit de IR: https://www.ir.com/guides/what-is-network-packet-loss

Kushner, D. (2003). *Masters of Doom.* New York: Random House.

Lewis, M., & Jacobson, J. (2002). Game Engines in scientific research. *Communications of the ACM*, 27-31.

Mikson, J.-S. (25 / Març / 2019). *Unity Networking Frameworks - Feature List.* Recollit de Google Drive: https://docs.google.com/document/d/1ufcl6_ylbzthfmzzZ7HPiwDLR0sr1N3UAunUG9rnisY/

Mikson, J.-S. (15 / Juliol / 2019). *Unity Networking Frameworks - Feature List Excel.* Recollit de Google Drive: https://docs.google.com/spreadsheets/d/100vNy3grUgLV5M7rIUKUtRqO4cmTewQI8P5uwf-Qb9k/

Mirror. (2022). *A history of Mirror*. Recollit de Mirror: https://mirror-networking.gitbook.io/docs/trivia/a-history-of-mirror

Mirror. (2022). *Mirror Networking Docs*. Recollit de Mirror: https://mirror-networking.gitbook.io/docs/

No Bugs Hare. (2015). *Development and deployment of Multiplayer Online Games Vol I.* Austria: ITHare.com Website GmbH.

Noll, A. M. (1966). Computers and the visual arts. *Design Quarterly*, 64-71.

Photon. (2022). *PUN Features*. Recollit de Photon: https://www.photonengine.com/pun

Photon. (2022). *RPC and RaiseEvent*. Recollit de Photon Engine: https://doc.photonengine.com/en-us/pun/current/gameplay/rpcsandraiseevent

Prabha, C. (2015). Login and Networking Services for Online Multiplayer Games. Tesi final de màster, Metropolia University of Applied Sciences, Hèlsinki, Finlàndia. Recollit de https://www.theseus.fi/bitstream/handle/10024/98794/PeachiMuthuChithra-Masters-IT-Thesis.pdf

Preston-Werner, T. (2013). *Semantic Versioning 2.0.0*. Recollit de Semantic Versioning: https://semver.org/spec/v2.0.0.html

Roelofs, G. R. (2002). *Estats Units d'Amèrica Patent núm. 6,475,090 B2.*

Salz, D. (4 / Juny / 2016). *Albion Online - Software Architecture of an MMO.* Recollit de https://www.slideshare.net/davidsalz54/albion-online-software-architecture-of-an-mmo-talk-at-quo-vadis-2016-berlin

Sara. (4 / Maig / 2022). *Migrating from UNet to Netcode for GameObjects*. Recollit de Unity Multiplayer Networking: https://docs-multiplayer.unity3d.com/netcode/current/migration/migratingtonetcode

Sara. (18 / Abril / 2022). *Transports*. Recollit de Unity Multiplayer Networking: https://docs-multiplayer.unity3d.com/netcode/current/advanced-topics/transports

Smith, E. J. (1999). *Estats Units d'Amèrica Patent núm. 5,899,810.*

Terrano, M., & Bettner, P. (22 / Març / 2001). *1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond.* Recollit de Gamasutra: https://www.gamasutra.com/view/feature/3094/1500_archers_on_a_288_network_.php

Unity Technologies. (2020). *Choosing the right netcode*. Recollit de Unity: https://create.unity.com/form-netcode-report

Unity Technologies. (8 / Abril / 2022). *About Netcode for GameObjects*. Recollit de Unity Multiplayer Networking: https://docs-multiplayer.unity3d.com/netcode/current/about

Unity Technologies. (2022). *Unity Multiplayer Networking*. Recollit de Unity Multiplayer Networking: https://docs-multiplayer.unity3d.com/

Valve. (29 / Agost / 2021). *Valve Developer Community*. Recollit de Prediction: https://developer.valvesoftware.com/wiki/Prediction

Valve. (sense data). *Valve Developer Community*. Recollit de Source Multiplayer Networking: https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking

Wasserman, K., & Stryker, T. (1980). Multimachine Games. *Byte*, 20.

8.2 Ludografia

The Behemoth (2013). Battleblock Theater (PC, Xbox, MacOS i Linux) [Videojoc]. San Diego, CA: Microsoft Studio.

Valve Corporation (2012). Counter-Strike Global Offensive (Windows, macOS, Linux, PlayStation 3, Xbox 360) [Videojoc]. Bellevue, Washington: Valve Corporation.

Centres universitaris adscrits a la

Grau en Disseny i Producció de Videojocs

DESENVOLUPAMENT D’UN PROTOTIP D’UN VIDEOJOC EN XARXA

Annexes

Jordi Alba Franco
Tutor: Jordi Arnal Montoya
Curs: 2021 - 2022

Índex

Annexes dins Google Drive. 1

Material de tercers 3

Software de tercers 5

Annexes dins Google Drive.

A continuació s'explica la ruta on estan els elements annexats digitals del treball.

El projecte es pot descomprimir i obrir utilitzant *Unity Hub* o la versió ***2020.3.14f1*** de Unity. L'escena inicial es diu "00_Launcher", situada dins "Assets/Scenes/Game/00_Launcher.scene"

Les *builds* de Unity són versions que s'han fet per les proves de xarxa. Cada *".zip*" conté un executable amb la icona de Unity per jugar. Addicionalment, també té un fitxer anomenat "changelog.txt" que conté el diari de desenvolupament.

1. **PUN / Photon**
   a. **Codi font:** Annexes / PUN / PUN_Projecte.zip
   b. **Executables:** Annexes / PUN / Builds /...
2. **Mirror**
   a. **Codi font:** Annexes / Mirror / Mirror_Projecte.zip
   b. **Executables:** Annexes / Mirror / Builds /...

També es pot trobar el vídeo demostratiu de la versió 0.0.8.1 Mirror. La ruta és:

- Annexos / Alba_Franco_VideoDemostratiu_TFG.mp4

Dins la carpeta d'annexes també es pot trobar les taules comparatives originals i les usades d'exemple per crear les comparatives. Es poden trobar en:

- Annexos / TaulesComparatives.xlsx

Material de tercers

Per a l'elaboració del prototip han estat utilitzats els següents assets.

Clean Settings UI: Conjunt de *sprites* i recursos per crear UI minimalista. Publicat a la Unity Asset Store per "Landan Lloyd". Obtingut de: https://assetstore.unity.com/packages/tools/gui/clean-settings-ui-65588

RPG/FPS Game Assets (Industrial Set v2.0): Conjunt de models industrials utilitzats dins la escena "Game". Publicat a la Unity Asset Store per "Dimitrii Kutsenko". Obtingut de: https://assetstore.unity.com/packages/3d/environments/industrial/rpg-fps-game-assets-for-pc-mobile-industrial-set-v2-0-86679

Cinemachine: Càmera interactiva, configurable amb extensions. Integrat amb Package Manager de Unity.

Models dels personatges cedits per Sergi Segarra Garrucho. (@ssegarra3D)

Software de tercers

S'ha utilitzat les següents llibreries per desenvolupar el projecte.

Rider: IDE Multiplataforma feta per JetBrains. Versió 2019.3.1. Obtingut de: https://www.jetbrains.com/rider/download/

PUN: Llibreria multijugador per Unity. Fet per Exit Games / Photon. Versió 2. Obtingut de: https://assetstore.unity.com/packages/tools/network/pun-2-free-119922.

Mirror: Llibreria multijugador per Unity. Fet per Mirror Networking. Obtingut de: https://mirror-networking.com/

ModelStitching: Llibreria per unir les peces de roba individuals al model. Fet per l'usuari "masterprompt" a GitHub. Obtingut de: https://github.com/masterprompt/ModelStitching