Karelia-ammattikorkeakoulu
Tradenomi, Tietojenkäsittely (AMK)

Edistyneiden liikemekaniikkojen toteuttaminen verkkomoninpeliin

Onni Forsblom

Opinnäytetyö, Joulukuu 2023

www.karelia.fi

Karelia AMMATTIKORKEAKOULU

OPINNÄYTETYÖ
Joulukuu 2023
Tietojenkäsittelyn koulutus

Tikkarinne 9
80200 JOENSUU
+358 13 260 600

Tekijä
Onni Forsblom

Nimeke
Edistyneiden liikemekaniikkojen toteuttaminen verkkomoninpeliin

Tiivistelmä

Tämän opinnäytetyön tavoitteena oli toteuttaa neljä sulavasti toimivaa liikemekaniikkaa verkkomoninpeliin: teleportaatio, liikkeen suorittaminen takaperin määritellyltä ajanjaksolta sekä seinäjuoksu ja -hyppy. Työn tavoitteena oli myös tutkia ja vertailla kahta erilaista moninpeleissä hyödynnettyjä ohjelmistoarkkitehtuuria.

Työn kehitysympäristönä toimivat asiakas-palvelin-mallia hyödyntävä Unreal Engine 5 -pelimoottori sekä Epic Games -yrityksen tarjoama Lyra Starter Game -esimerkkiprojekti. Mekaniikat toteutettiin Unreal Engine -moottorin tarjoamalla hahmon liikkumiskomponentilla sekä Gameplay Ability -järjestelmällä. Kaikkia tuotoksia testattiin simuloimalla pelaamista verkossa asiakkaan roolissa verkkopakettien viiveen kanssa.

Yksinkertaisen teleportaatiotaidon toteuttaminen Gameplay Ability -järjestelmällä oli sekä helppoa että intuitiivista. Kuitenkin testatessa tällaisia toteutuksia suurella verkkopakettien viiveellä korjaustilanteita esiintyi enemmän verrattuna liikekomponenttia hyödyntävään toteutukseen, etenkin ajan kelaamisen kohdalla. Mekaniikkojen toteuttaminen hahmon liikekomponentilla oli taas monimutkaisempaa, mutta kokonaisuutena erittäin opettava kokemus. Seinäjuoksu- ja hyppymekaniikan toteuttamiselle ei tätä työtä varten riittänyt aikaa, mutta kyseisten mekaniikkojen luomiseen verkkomoninpeliä varten löytyi valmiita toteutuksia.

Kieli
suomi

Sivuja 62
Liitteet 0
Liitesivumäärä 0

Asiasanat
peliohjelmointi, moninpelit, verkkopelit, viive

Karelia AMMATTIKORKEAKOULU

THESIS
December 2023
Degree Programme in Business Information Technology

Tikkarinne 9
FI 80200 JOENSUU
FINLAND
Tel. +350 13 260 600

Author
Onni Forsblom

Title
Implementation of Advanced Movement Mechanics for Online Multiplayer

Abstract

The objective of this thesis was to implement four seamlessly functioning movement mechanics for an online multiplayer game: teleportation, executing movement backwards from a defined period of time, wall running, and wall jumping. Another academic goal of this thesis was to examine and compare two distinct software architectures frequently utilized in multiplayer games.

The development environment for this work comprised of the client-server model utilizing Unreal Engine 5 game engine, and the Lyra Starter Game example project provided by Epic Games. The mechanics were realized using the character movement component and the Gameplay Ability System provided by Unreal Engine. All implementations were tested by simulating online gameplay in the client role along with packet lag.

Implementing the straightforward teleportation ability with the Gameplay Ability system proved to be both easy and intuitive. However, when testing such implementations with significant packet lag, more instances of server-reconciliation occurred compared to implementations utilizing the movement component, especially in the case of time rewinding. In contrast, implementing the mechanics using the character movement component was more intricate but overall a very instructive experience. Time constraints prevented the development of wall run and wall jump mechanics for this thesis, but readily available implementations of these mechanics in the context of online multiplayer games were found.

Language	Pages 62
Finnish	Appendices 0
	Pages of Appendices 0

Keywords
game programming, multiplayer games, online games, latency

Sisältö

Sanasto

Etäproseduurikutsu Funktio jota kutsutaan lokaalisti, mutta suoritetaan etänä toisella koneella. (Epic Games 2023a.)

LineTrace-funktio Metodi jolla tarkastetaan mitä mahdollisia objekteja kahden pelimaailman pisteen välillä on. (Epic Games 2023b.)

Replikointi Prosessi jolla tietoja ja proseduurikutsuja synkronoidaan asiakkaan ja palvelimen välillä. (Epic Games 2023c.)

1 Johdanto

Tämän opinnäytetyön konkreettisena tavoitteena on toteuttaa sulavasti toimivia edistyneitä liikemekaniikkoja verkkomoninpeliin. Näihin mekaniikkoihin kuuluu pelaajahahmon teleportaatio, liikkeen suorittaminen takaperin määritellyltä ajanjaksolta sekä seinäjuoksu ja -hyppy. Kyseiset liikekyvyt tulee myös toteuttaa siten, että pelaaja ei koe syötteissään viivettä, ja liikkuminen voidaan laskea uudelleen, mikäli asiakkaan ja palvelimen välillä esiintyy epäsynkronointia.

Kehitystyö suoritetaan asiakas-palvelin-mallia hyödyntävällä Unreal Engine 5 -pelimoottorilla, ja työn pohjana käytetään Epic Gamesin tarjoamaan Lyra Starter Game -esimerkkiprojektia. Pelinkehityksen harjoittelu valmiin arkkitehtuurin pohjalta on myös yksi tämän opinnäytetyön tavoitteista.

Aihe perustuu työnhakuprosessissa annettuun tehtävänantoon. Merkittävin syy aihevalinnalle on nettipelien kehittämisen tietotaidon hyödyllisyys ja kysyntä pelialalla. Lisäksi mekaniikkojen kehittäminen toimintamoninpelejä varten on opinnäytetyön tekijälle uusi ja haasteellinen teema, jota hänen opetussuunnitelmassa ei olla käsitelty. Kirjoittaja kuitenkin omaa hieman kokemusta nettipelien kehittämisestä Game as a service -palvelujen toteuttamisen kautta.

Toinen työn oppimistavoitteista on tutkia ja vertailla kahta erilaista moninpeleissä hyödynnettyjä ohjelmistoarkkitehtuuria: vertaisverkkoa ja asiakas-palvelin-mallia. Tämä myös auttaa pohjustamaan varsinaista kehitystyötä. Asiakas-palvelin-arkkitehtuurin kohdalla lähempään tarkasteluun otetaan client-side prediction sekä server reconciliation -tekniikoiden osuus viiveen minimoimisessa ja liikkeen uusimisessa.

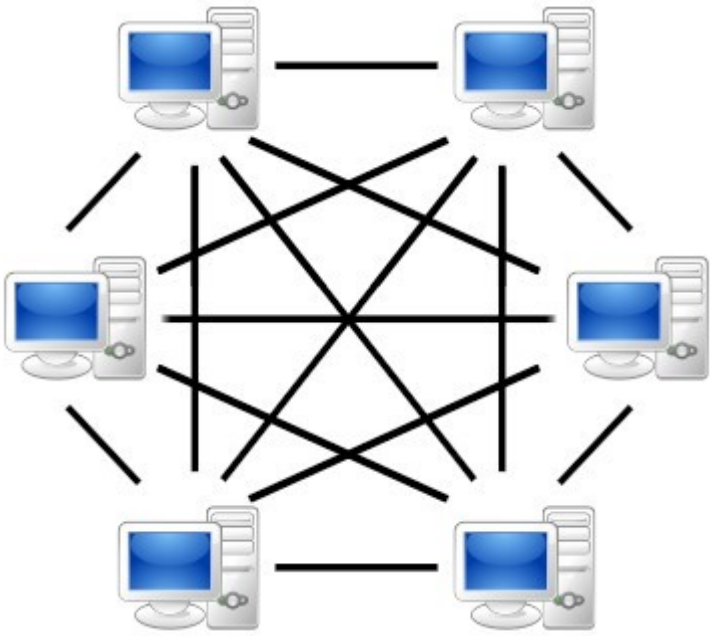
Liikemekaniikat ovat tarkoitus toteuttaa Unreal Engine -moottorin tarjoamalla hahmon liikkumiskomponentilla sekä mahdollisesti Gameplay Ability -järjestelmällä. Mekaniikkojen toimintalogiikka on tarkoitus suunnitella pitkälti itse

noudattamatta mitään tiettyä olemassa olevaa ratkaisua. Kaikkia tuotoksia lopulta testataan simuloimalla pelaamista verkossa asiakkaan roolissa verkkopakettien viiveen kanssa.

2 Verkkomoninpelien arkkitehtuurit

2.1 Vertaisverkko

Vertaisverkkoa eli peer-to-peer- tai P2P-verkkoa on hyödynnetty nopeatempoisissa ampumapeleissä, kuten id Softwaren Doom-pelissä, jo 90-luvun alkupuolella. Tällaisessa verkostossa jokaisen pelaajan tietokone vaihtaa tietoja keskenään ilman välikätenä toimivaa palvelinta (kuvio 1). Kun tietokone saa vastaanotettua muiden pelaajien syötteet, nämä komennot toteutetaan ja peli etenee. (van Waveren 2006, 2-3.)



Kuvio 1. Vertaisverkossa kaikki tietokoneet ovat suoraan yhteydessä toisiinsa (Kuvio: Mauro Bieg).

P2P-mallin etuja ovat sen yksinkertaisuus ja toteuttamisen helppous (van Waveren 2006, 3). Tämä järjestelmä myös välttää palvelimien ylläpitämiseen liittyvät kustannukset (Lincroft, 1999). Vaikka palvelimien perustamisen vastuun voisikin antaa pelaajille, tämä ei välttämättä aina ole mahdollista lisenssiin liittyvistä syistä (Lincroft, 1999).

Vertaisverkko on myös suhteellisen vankka verkosto, sillä siinä ei ole yksittäistä osaa, joka voisi itsestään lakkauttaa koko verkon toiminnan (Bettner, 2001). Lisäksi kommunikointi suoraan pelaajien koneiden kesken on nopeaa, koska dataa ei tarvitse siirtää palvelimen kautta (Bettner, 2001). Peer-to-peer-arkkitehtuurin suurin etu voi kuitenkin olla sen skaalautuvuus, koska kyseinen arkkitehtuuri hyödyntää jokaisen pelaajan laitteiston resursseja (Neumann, Prigent, Varvello & Suh 2007, 81).

Vertaisverkon hyödyntämisessä on myös huonoja puolia. Erityistä huolta täytyy pitää siitä, että jokaisen pelaajan pelin tila on täydellisesti synkronoitu; pienikin eroavuus voi johtaa pelaajat täysin erilaisiin tiloihin omissa peleissään. Näiden ristiriitaisuuksien välttämiseksi pelin tilaa tulee ylläpitää hyödyntämättä esimerkiksi ruudunpäivitysnopeutta tai laitteistosta riippuvia syötteitä. P2P-mallin tapaisissa verkostoissa pelaaminen eri pelikonsolien välillä ei myöskään ole välttämättä mahdollista laitteistojen aiheuttamien erojen vuoksi. (van Waveren 2006, 3-4.)

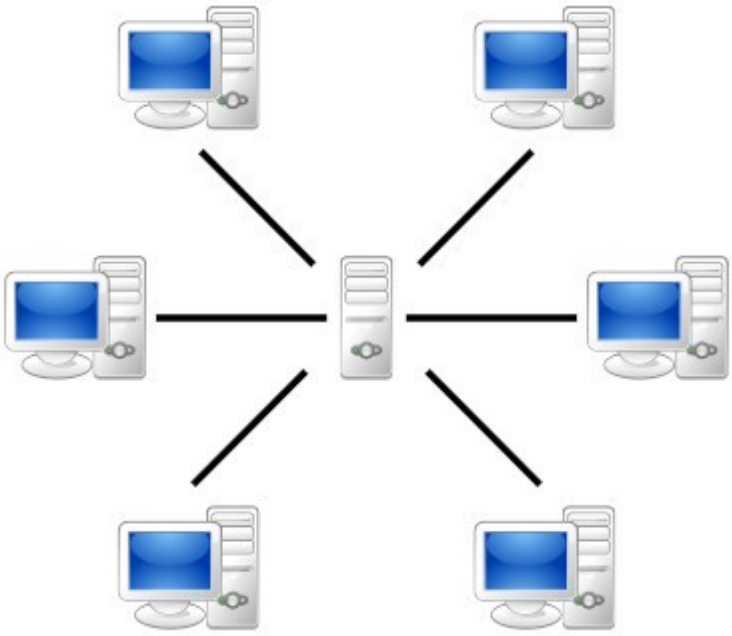
Täydellisen synkronoinnin pohjalta syntyy myös toinen ongelma: syöttöviive. Pelaajan syötteet tulee aina vastaanottaa muiden pelaajien päissä ennen kuin peli voi edetä, minkä seurauksena jokaisella pelaajalla on yhtä suuri latenssi kuin hitaimman yhteyden omaavalla. (Fiedler 2010.)

Lähiverkossa viive ei koidu ongelmaksi tietokoneiden välisen latenssin ollessa matala, mutta internetin varassa toimiessa se voi nousta huomattavasti. Lisäksi peer-to-peer-verkossa kaistanleveyden vaatimukset ovat suhteellisen korkeita ja ne kasvavat merkittävästi pelaajien määrän lisääntyessä. (van Waveren 2006, 4.)

Vertaisverkot myös mahdollistavat monia huijausmenetelmiä; koska jokaisen pelaajan kone on synkronoitu samalla tavalla koko pelin kanssa, pelaaja voi hyödyntää huijauksia esimerkiksi nähdäkseen asioita, joita hänen ei pitäisi pystyä näkemään (van Waveren 2006, 4). Hakkeri voi myös helpommin käyttää hyväkseen verkostoa, koska jokainen pelaaja on suoraan yhteydessä toisiinsa eikä vain luotettuun palvelimeen (Neumann ym. 2007, 81).

2.2 Asiakas-palvelin-arkkitehtuuri

id Softwaren vuonna 1996 ilmestynyt ensimmäisen persoonan ammuntapeli Quake oli ensimmäisiä merkittäviä pelejä, joka hyödynsi asiakas-palvelin-arkkitehtuuria. Tässä järjestelmässä pelaajien koneet eli asiakkaat lähettävät syötteensä pelkästään auktoritatiiviselle palvelimelle, joka ylläpitää itse peliä (kuvio 2). Asiakkaat vuorostaan saavat tiedot pelaajalle näytettävistä asioista palvelimelta, eli pelaajien tietokoneet toimivat eräänlaisina tyhminä päätteinä. (van Waveren 2006, 5.)



Kuvio 2. Asiakas-palvelin-arkkitehtuurissa kaikki asiakkaat ovat yhteydessä vain palvelimeen (Kuvio: Mauro Bieg).

Asiakas-palvelin-arkkitehtuurissa pelaajien kaistanleveyden vaatimus pysyy suhteellisen matalana pelaajamäärästä huolimatta (Glazer & Madhav 2016, 7). Lisäksi latenssin määrä ei riipu hitaimmasta verkkoyhteydestä, vaan se määräytyy asiakkaan ja palvelimen yhteyden nopeuden mukaan (Fiedler 2010). Epäsynkronointi ei myöskään ole ongelma tässä mallissa, sillä vain palvelin on vastuussa pelin tilasta (van Waveren 2006, 5). Tämän ansiosta lisää pelaajia voi liittyä keskeneräiseen peli-istuntoon (van Waveren 2006, 5). Lisäksi

palvelinta hyödyntäessä pelien ei tarvitse olla täysin deterministisiä (Fiedler 2010).

Asiakas-palvelin-malli auttaa myös estämään huijaamista moninpeleissä. Koska auktoritatiivinen palvelin suorittaa itse peliä ja pelaajat vain lähettävät syötteitä sekä vastaanottavat tietoa, asiakas ei voi helposti esimerkiksi muuttaa pelaajahahmon tilaa tai sijaintia saadakseen epäreilun edun. Hakkerit voivat kuitenkin yrittää hyödyntää palvelimen mahdollisia heikkouksia. (Gambetta 2023.)

Asiakas-palvelin-arkkitehtuurissa on kuitenkin heikkouksia verrattuna vertaisverkkoon. Keskuspalvelimet vaativat investointia prosessointitehoon, kaistanleveyteen sekä huoltoon. Lisäksi palvelimen ruuhkautuminen ja kaatuminen heijastuu siitä riippuvissa peleissä (Corman, Douglas, Schachte & Teague 2006, 1).

Ilman erityistoimenpiteitä asiakas-palvelin-malli kärsisi myös huomattavasta latenssin määrästä, koska syötteet pitäisi ensin lähettää palvelimelle ja tieto pelin tilasta lähettää takaisin asiakkaalle ennen kuin pelaaja saa palautetta päätteessään (Carmack 1996, 7). Latenssin vähentämiseksi QuakeWorld-peliä varten John Carmack (1996, 9) salli pelaajan tietokoneen simuloida pelaajahahmon liikettä odottaessa palvelimen vastausta. Palvelin on kuitenkin päättävä taho tilanteissa, joissa se on ristiriidassa asiakkaan puolen kanssa, jolloin asiakas simuloi uudelleen pelaajahahmon sijainnin viimeisestä oikeasta asemasta (Carmack 1996, 9). Näitä Carmackin client-side prediction ja server-reconciliation -tekniikoita hyödynnetään moderneissakin ammuntapeleissä (Fiedler 2010).

3 Verkkomoninpeli Unreal Enginessä

3.1 Unreal Engine -verkkopelin arkkitehtuuri

Verkkomoninpeleissä Unreal Engine hyödyntää asiakas-palvelin-arkkitehtuuria; yksi tietokone toimii pelissä palvelimena ja isäntänä, joka ylläpitää pelin tilaa. Muut koneet toimivat asiakkaina, joista jokainen ohjaa omistavaansa pelinappulaa. Palvelin jatkuvasti replikoi tietoa pelin tilasta ja objekteista jokaiselle asiakkaalle, jotka sitten simuloivat tilanteen omassa päässään. (Epic Games 2023d.)

Unreal Engine -moottorin hahmon liikekomponentti on suunniteltu yhteensopivaksi verkkopelien kanssa. Pelin aikana erikoistuneet funktiot kutsuvat komponentin PerformMovement-metodia, joka replikoi liikettä eri tavoin riippuen pelaajahahmon roolista verkostossa. Kyseisiä rooleja on kolme: asiakkaan paikallisesti ohjattu autonomous proxy, palvelimella toimiva authoritative actor sekä muiden asiakkaiden pelissä näkyvä etäältä ohjattu simulated proxy. (Epic Games 2023e.)

Asiakkaan paikallisesti ohjattu hahmo suorittaa liikelogiikkaa PerfomMovement-funktion avulla. Tämän lisäksi autonomous proxy säilöö dataa liikkumisestaan FSavedMove_Character-objekteina ja lähettää tiedot palvelimelle ServerMove-etäproseduurikutsulla. (Epic Games 2023e.)

Vastaanotettua asiakkaan lähettämät tiedot liikkumisesta, palvelimella toimiva authoritative actor -roolin omaava pelihahmo jäljentää pelaajan liikkeet PerformMovement-metodilla. Palvelin ilmoittaa asiakkaalle, jos kummassakin päässä hahmon viimeiset sijainnit täsmäävät. Muulloin palvelin lähettää korjauksen ClientAdjustPosition-etäproseduurikutsulla, jonka seurauksena asiakas jäljentää palvelimella toimivan hahmon liikkeet sekä laskee uuden viimeisen sijainnin tallennettujen syötteiden pohjalta. (Epic Games 2023e.)

Toisin kuin autonomous proxyt, authoritative actor -tyyppiset hahmot eivät käytä jatkuvasti aktivoituvaa TickComponent-metodia liikkeen määrittämisessä käytetyn delta time -muuttujan laskemiseen. Sen sijaan palvelin odottaa asiakkaan kutsuvan ServerMove-funktiota, jolloin palvelin laskee eron kahden viimeisimmän liikkeen aikaleimoissa. Koska tässä laskussa hyödynnetään enimmäkseen palvelimen aikaleimoja, asiakkaat eivät voi nopeuttaa itseään manipuloimalla heidän paikallisen pelin kelloa. (Epic Games 2023e.)

Unreal Engine -pelimoottorissa AGameNetworkManager-luokka suorittaa nettipeleissä huijausten havaitsemista muiden asioiden ohella. Jos esimerkiksi asiakkaan ajan manipuloinnin havaitseminen ja mitätöiminen halutaan ottaa käyttöön, objektin bMovementTimeDiscrepancyDetection- ja bMovementTimeDiscrepancyResolution-muuttujat tulee asettaa todeksi. Myös esimerkiksi aikaeron virhemarginaalia voidaan säätää luokan eri muuttujista. (Epic Games 2023f.)

Lopulta kaikkien muiden asiakkaiden simulated proxy -hahmot soveltavat suoraan palvelimelta saatuja liiketietoja interpoloimalla hahmojen sijaintia lähtöpaikkojensa ja kohdepaikkojensa välillä. Tätä network smoothing -tekniikkaa hyödynnetään tasoittaakseen liikkumisen visuaalisuutta, kun pelaajan virkistystaajuus on korkeampi kuin replikoidun liikkeen lähettämisen taajuus. (Epic Games 2023e.)

3.2 Mekaniikkojen lisääminen liikekomponenttiin

On useita tapoja lisätä uusia liikemekaniikkoja pelaajahahmolle, mutta jotkut menetelmät toimivat toisia paremmin verkkomoninpeleissä. Esimerkiksi teleportaatiotaidon voi lisätä yksinkertaisesti antamalla asiakkaalle kyvyn pyytää palvelinta käsittelemään objektin teleportaation. Koska tällaisessa ratkaisussa liikettä ei ensin simuloida asiakkaan puolella, toteutus tulee aiheuttamaan viivettä taidon aktivoinnissa. Lisäksi tämän aiheuttamat eroavaisuudet palvelimen ja asiakkaan välillä saa välittömän liikkeen näyttämään odottamattoman sulavalta. Vaikka tätä toteutusta korjaisikin simuloimalla

teleportaation heti asiakkaan puolella, teleportaation suorittaminen kumottaisiin mikäli palvelin joutuisi korjaamaan pelaajan liikettä. (Epic Games 2018.)

Yksi verkkopelaamisen kanssa yhteensopiva tapa toteuttaa uusia liiketaitoja on lisäämällä kyvyt pelihahmon liikekomponenttiin. Näin liike tulee suoritettua sekä paikallisesti että palvelimella, eikä syöte tule kumotuksi korjaustilanteissa. Tämän mahdollistamiseksi projektiin on luotava uudet hahmon liikekomponentista ja FSavedMove_Character-luokasta perivät luokat. Liikekomponenttiin määritetään uudelleen AllocateNewMove-metodi, joka on yhteensopiva uusien liikkeiden tallentamista varten luotujen objektien kanssa. Mukautettuun versioon FSavedMove_Character-luokasta ylikirjoitetaan GetCompressedFlags- ja UpdateFromCompressedFlags-funktiot, jotta nämä toimisivat uuden liikkumiskykyjen laukaisemisen kanssa. Näin liikkumiskykyjen aktivointi voidaan laskea uudelleen korjaustilanteiden jälkeen. (Epic Games 2018.)

3.3 Mekaniikkojen lisääminen Gameplay Ability -järjestelmällä

Toinen client-side prediction -tekniikkaa hyödyntävä tapa toteuttaa liikemekaniikat on käyttämällä Gameplay Ability System -liitännäisen taito-objekteja. Ability System -komponenttia hyödyntävälle peliobjektille voidaan antaa kykyjä GiveAbility- ja GiveAbilityAndActivateOnce-funktioilla. Taidon toiminnallisuus toteutetaan ActivateAbility-funktioon ja se voidaan aktivoida esimerkiksi CallActivateAbility- tai TryActivateAbility-metodilla. (Epic Games 2023g.)

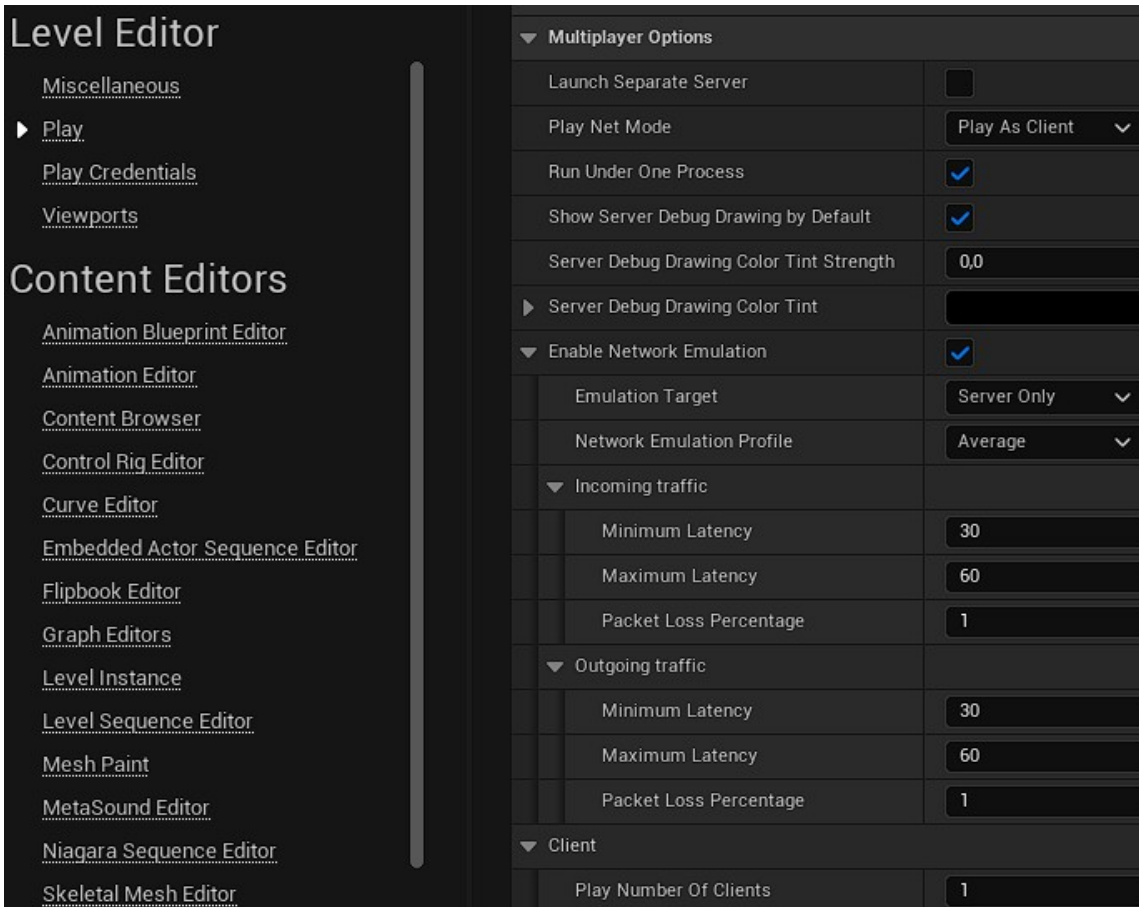
Kykyjä ei voida kuitenkaan aktivoida uudelleen kesken niiden suoritusta. Niiden päättämiseksi tulee hyödyntää EndAbility-funktiota tai CancelAbility-metodia, joista jälkimmäinen on taidon suorittamisen peruuttamista varten. Kykyjen välistä vuorovaikutusta voidaan myös hallita Gameplay-tagien avulla. Ability-objektille voi lisätä esimerkiksi BlockAbilitiesWithTag-säiliöön toisen taidon AbilityTags-säiliössä olevan tunnisteen, jotta tätä toista taitoa ei voi suorittaa ensimmäisen ollessa käynnissä. (Epic Games 2023g.)

Jotta peliobjekti voisi käyttää Gameplay Ability -järjestelmää, sen tulee toteuttaa IAbilitySystemInterface-rajapinta. Tästä rajapinnasta tulee ylikirjoittaa GetAbilitySystemComponent-funktio, joka palauttaa Ability System -komponentin. Kyseistä komponenttia yleensä säilytetään itse kykyjä käyttävässä peliobjektissa, mutta sen voi tarvittaessa kiinnittää myös muuhun objektiin. (Epic Games 2023h.)

Tarpeen mukaan kyvyn replikoinnin käsittelytyylin voi määrittää Gameplay Net Execution Policy -muuttujalla. Jos muuttujan arvoksi asetetaan ”Local Predicted”, taidon aktivointi simuloidaan ensin paikallisesti. (Epic Games 2023g.)

3.4 Verkkomoninpelin testaaminen

Unreal Engine -pelimoottori tarjoaa useita asetuksia ja vaihtoehtoja verkkomoninpelien testaamista varten (kuva 1). Editorin Play-asetuksista on mahdollista säätää esimerkiksi pelaajien määrää sekä Net Mode -muuttujaa (Epic Games 2023i). Riippuen Net Mode -muuttujan arvosta, peliä ajetaan yhdessä neljästä mahdollisesta eri tilasta (Epic Games 2023d).



Kuva 1. Moninpelien asetuksia Unreal Engine 5 -pelimoottorissa.

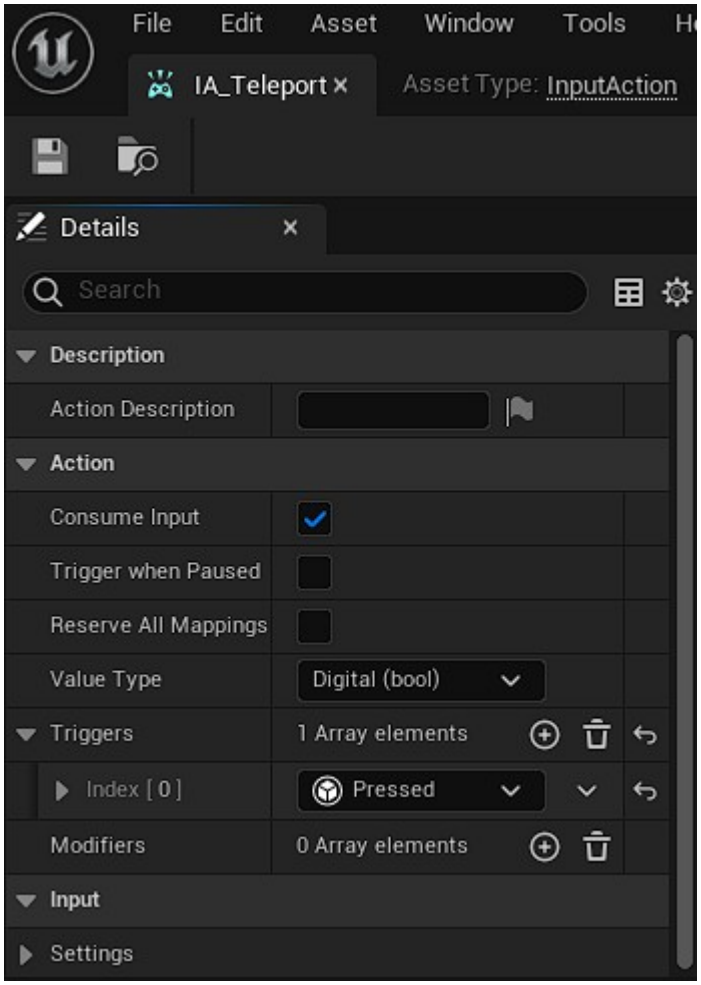
Standalone-tila on yksinpelejä ja paikallisia moninpelejä varten. Tässä tilassa peli toimii palvelimena, johon vain paikalliset asiakkaat voivat yhdistää. Client-asetuksella on mahdollista pelata pelkkänä palvelimeen yhdistettynä asiakkaana. Listen server -palvelimena toimiessa peli hyväksyy sekä lokaaleja että etänä toimivia asiakkaita. Turvallisempi ja laajamittaisempaan käyttöön sopivampi Dedicated Server -palvelin ei ota vastaan paikallisia asiakkaita ja toimiakseen tehokkaammin se ei esimerkiksi hahmonna grafiikkaa tai soita ääniä. (Epic Games 2023d.)

Lisäksi verkon emulaatiolla on mahdollista simuloida tietoliikenteen viivettä ja verkkopakettien häviötä, joita yleensä esiintyy todellisissa pelitilanteissa. Emulaatiota voi säätää pelimoottorin Play-asetuksista, konsolista, DefaultEngine.ini-tiedostosta tai komentokehotteesta. Konsolista haluttua asetusta voi muuttaa kirjoittamalla asetuksen nimen sekä uuden halutun arvon. (Epic Games 2023j.)

4 Kehittämisasetelma

4.1 Lyra Starter Game -esimerkkiprojekti

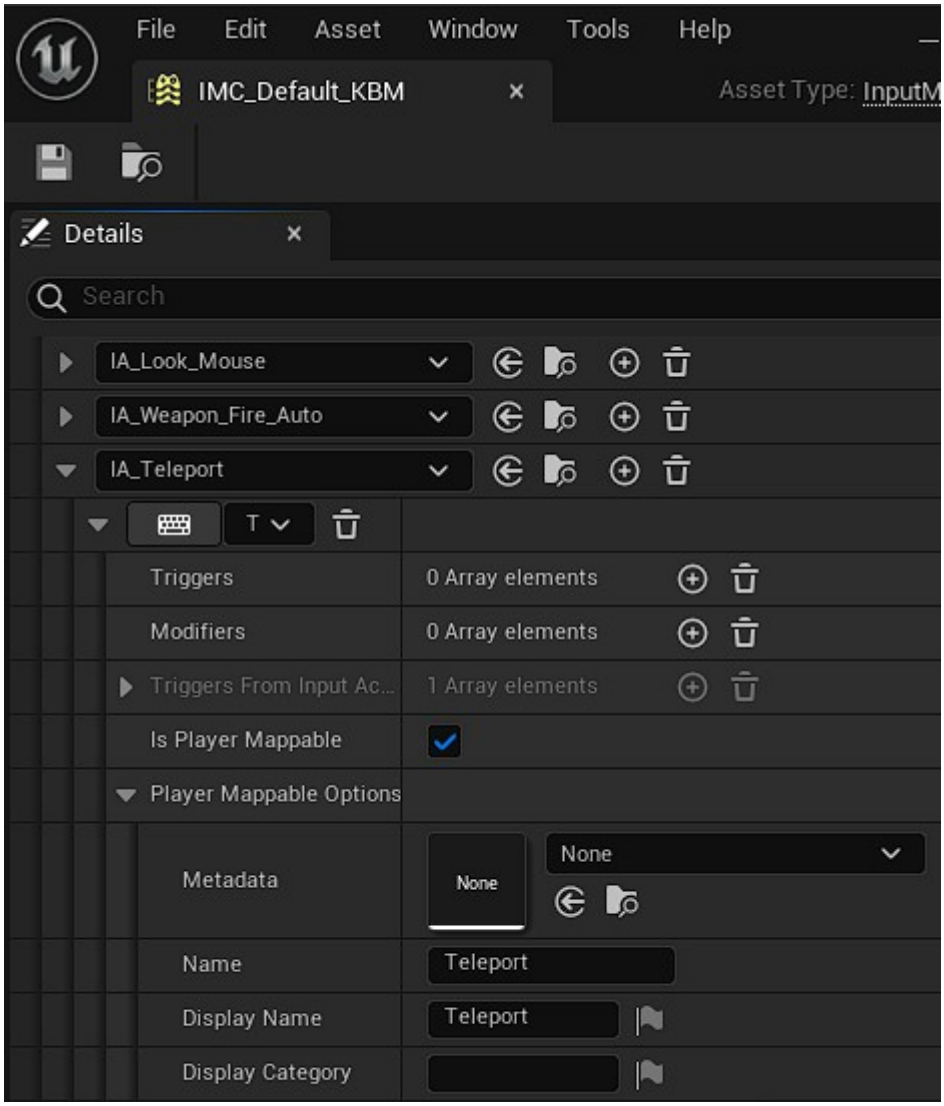
Liikemekaniikat toteutetaan Epic Gamesin tarjoamaan ilmaiseen Lyra Starter Game -projektiin. Syötteitä varten Lyra hyödyntää Enhanced Input -järjestelmää, jossa jokaiselle syötteellä aktivoitavalle toiminnalle luodaan Input Action -resurssi. Kyseiselle objektille voi esimerkiksi määrittää, millainen syöte laukaisee toiminnon, ja millainen arvo syötteestä saadaan (kuva 2).



Kuva 2. Input Action -resurssi ampumiselle.

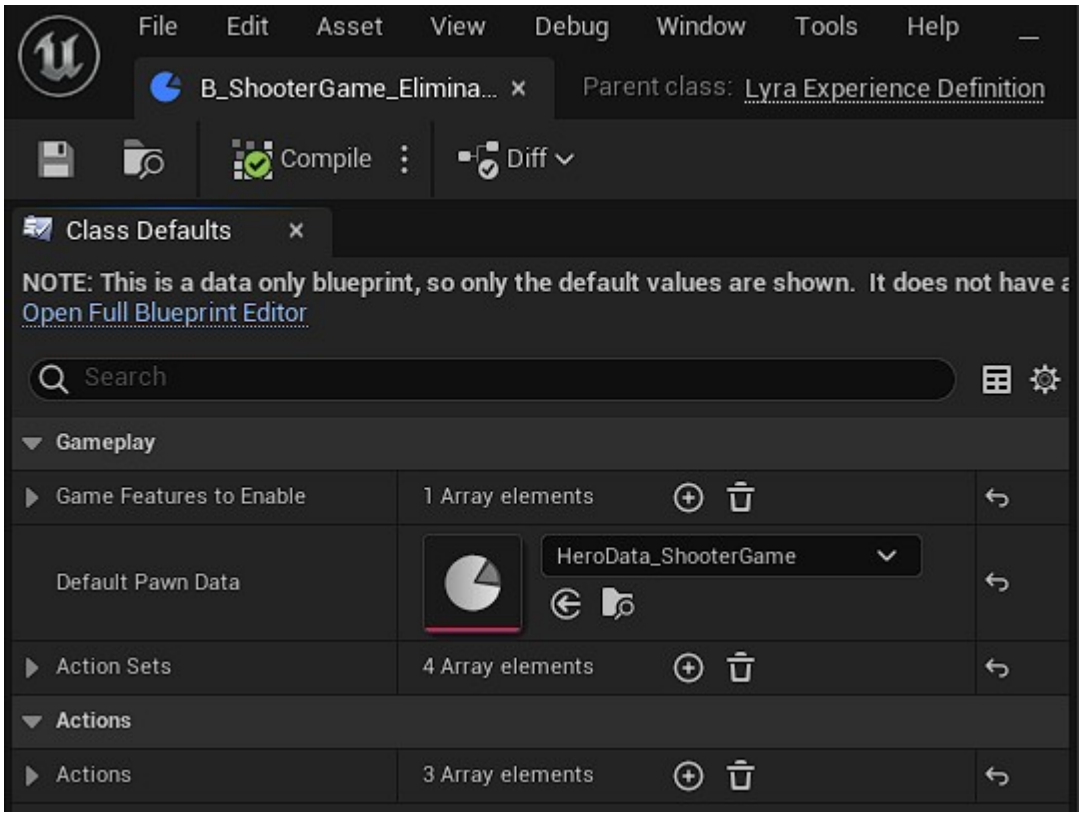
Luodut Input Action -resurssit lisätään Input Mapping Context -objektiin, jossa toiminnon aktivointi voidaan sitoa konkreettisen syöttölaitteen mekanismiin

(kuva 3). Toiminnot voi tässä tietorakenteessa myös sallia pelaajan määritettäväksi, jolloin niille tulee määrittää Name- ja DisplayName-muuttujat.



Kuva 3. Input Mapping Context hiirtä ja näppäimistöä varten.

Perinteisen GameMode-luokan lisäksi Lyra Starter Game hyödyntää Gameplay Experience -objekteja pelin sääntöjen määrittämiseksi (kuva 4). Tähän kuuluu myös pelaajan pelinappulan ominaisuuksien määrittäminen. Syötteet ja taidot tulee lisätä osaksi Experience-resurssin käyttämän pelinappulan data-objektin Input Config -tietorakennetta. Esimerkiksi ammuskelutasossa käytetty B_ShooterGame_Elimination-Gameplay Experience hyödyntää pelinappuloita varten HeroData_ShooterGame-luokkaa, joka sisältää InputData_Hero-tietorakenteen.

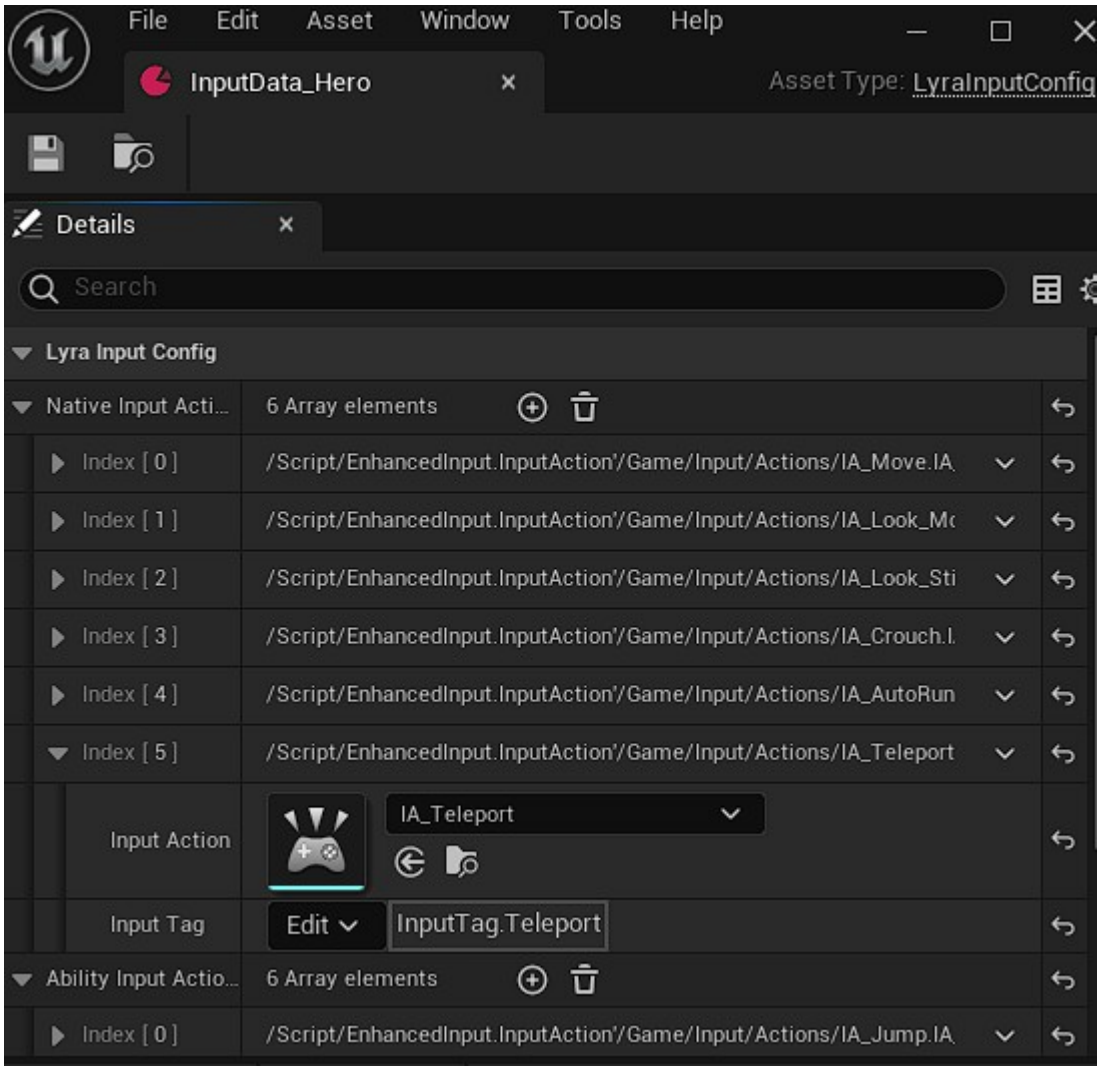


Kuva 4. Ammuskelutason B_ShooterGame_Elimination-Gameplay Experience.

Lopuksi Lyra-projektissa pelaajan mahdollisesti mukautetut ampujahahmon syötteet sidotaan LyraHeroComponent-komponentin InitializePlayerInput-funktiossa toimintoihin niiden Gameplay-tagin perusteella (listaus 1). Kyseiset tagit voidaan määrittää LyraGameplayTags-nimiavaruudessa ja liittää Input Action -resurssiin InputData_Hero-tietorakenteessa (kuva 5).

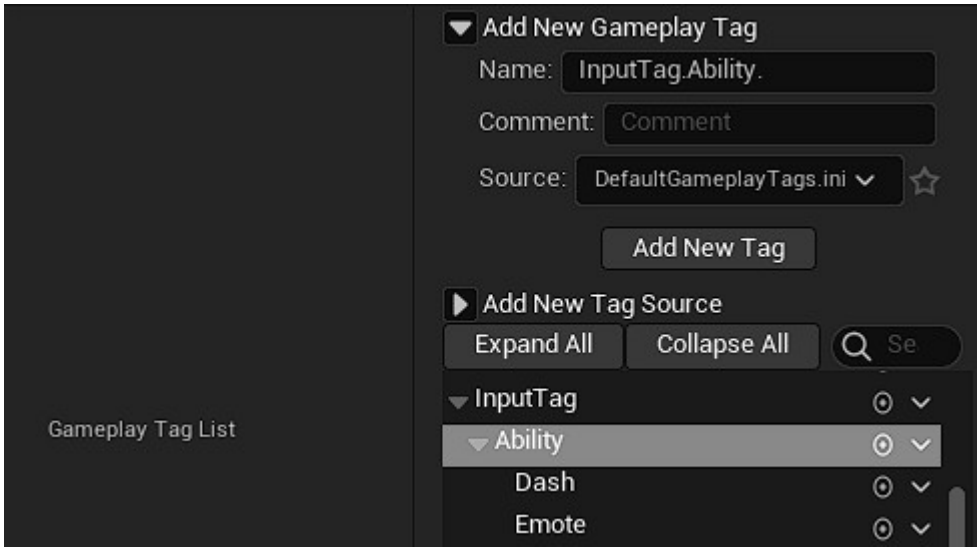
```
LyraIC->BindAbilityActions(InputConfig, this,
&ThisClass::Input_AbilityInputTagPressed,
&ThisClass::Input_AbilityInputTagReleased, /*out*/ BindHandles);

LyraIC->BindNativeAction(InputConfig, LyraGameplayTags::InputTag_Move,
ETriggerEvent::Triggered, this, &ThisClass::Input_Move, /*bLogIfNotFound=*/
false);
LyraIC->BindNativeAction(InputConfig, LyraGameplayTags::InputTag_Look_Mouse,
ETriggerEvent::Triggered, this, &ThisClass::Input_LookMouse,
/*bLogIfNotFound=*/ false);
LyraIC->BindNativeAction(InputConfig, LyraGameplayTags::InputTag_Look_Stick,
ETriggerEvent::Triggered, this, &ThisClass::Input_LookStick,
/*bLogIfNotFound=*/ false);
LyraIC->BindNativeAction(InputConfig, LyraGameplayTags::InputTag_Crouch,
ETriggerEvent::Triggered, this, &ThisClass::Input_Crouch,
/*bLogIfNotFound=*/ false);
LyraIC->BindNativeAction(InputConfig, LyraGameplayTags::InputTag_AutoRun,
ETriggerEvent::Triggered, this, &ThisClass::Input_AutoRun,
/*bLogIfNotFound=*/ false);
LyraIC->BindNativeAction(InputConfig, LyraGameplayTags::InputTag_Teleport,
ETriggerEvent::Triggered, this, &ThisClass::Input_Teleport,
/*bLogIfNotFound=*/ false);
```

Listaus 1. Syötteet liitetään funktioihin InitializePlayerInput-metodissa.

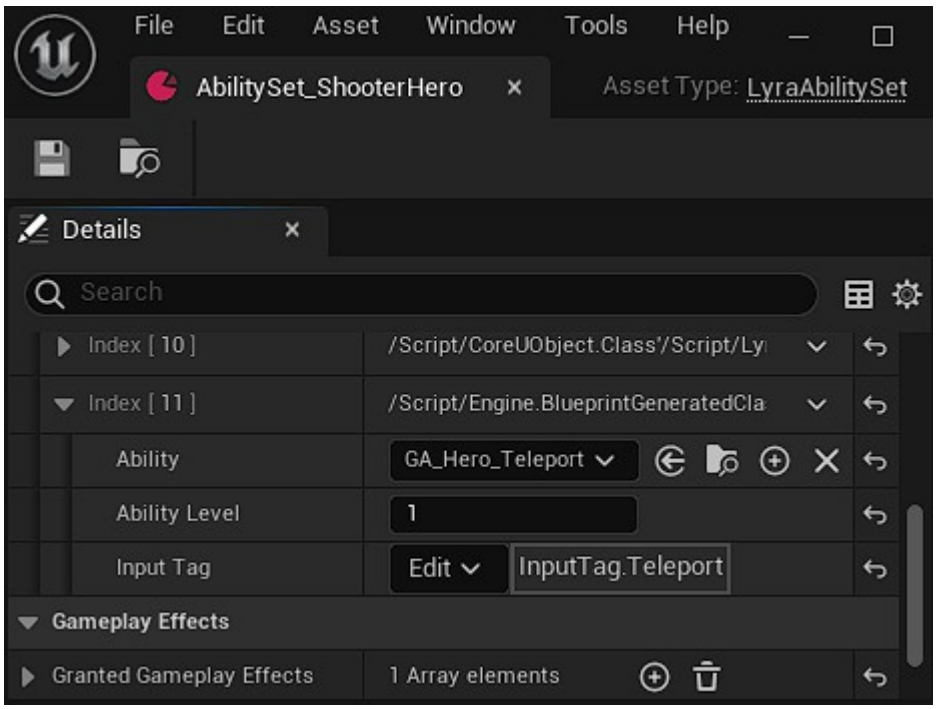


Kuva 5. InputData_Hero-tietorakenne.

Gameplay Ability -taitojen Input-tagit on mahdollista lisätä projektin asetusten kautta DefaultGameplayTags.ini-tiedostoon (kuva 6). Luotujen kykyjen blueprint-skriptit tulee periä LyraGameplayAbility-luokasta ja taidot on lisättävä tason Gameplay Experiencen käyttämän PawnData-tietorakenteen AbilitySet-objektiin tageineen (kuva 7).

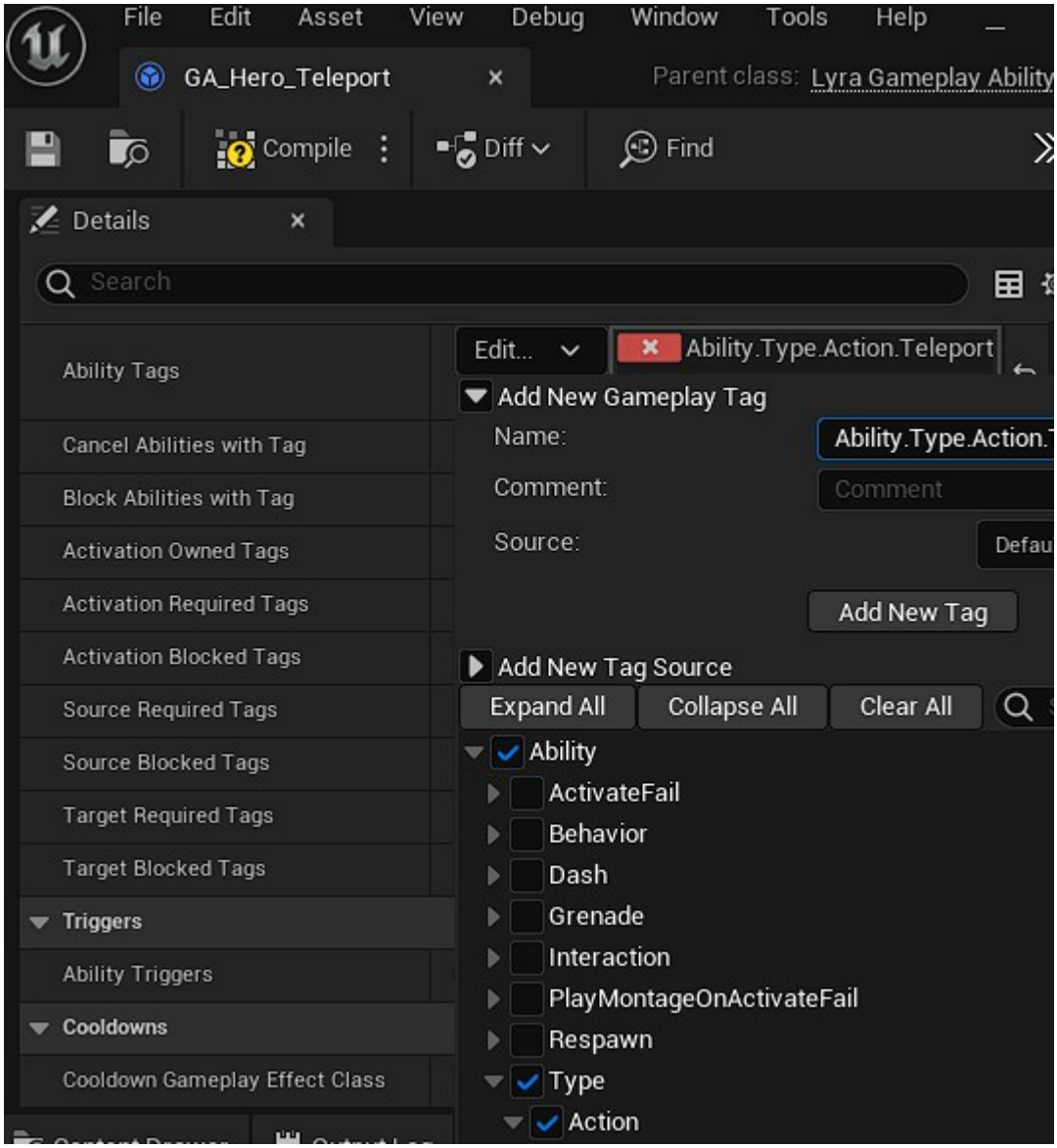


Kuva 6. Projektin GameplayTags-asetuksia.



Kuva 7. AbilitySet_ShooterHero-data-assetti.

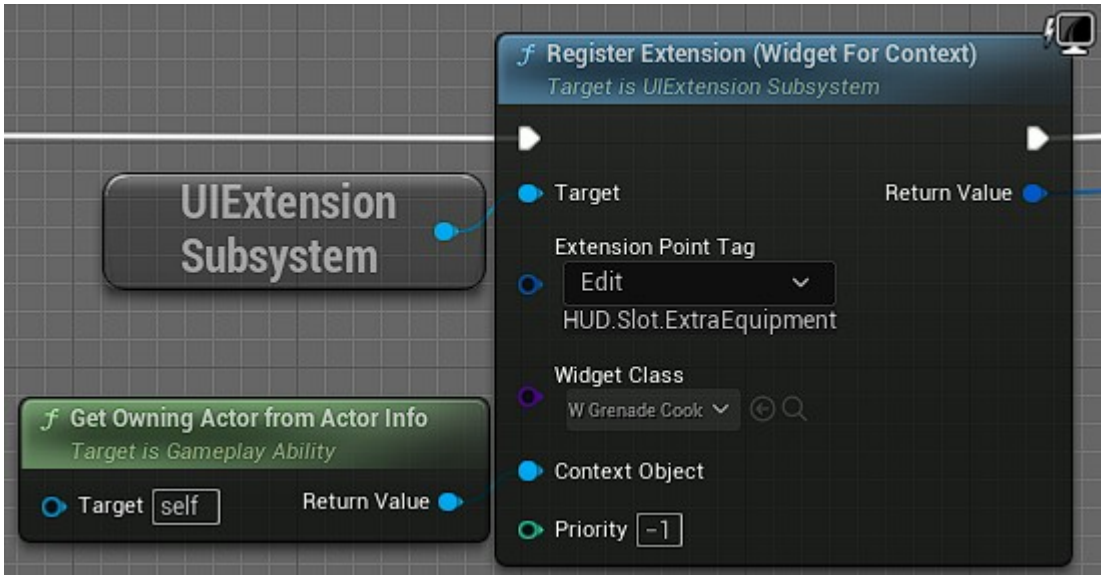
Jotta uudet Gameplay Ability -kyvyt toimisivat ongelmitta vanhojen rinnalla, niille on parasta lisätä Ability.Type.Action-tyyppinen Ability-tagi (kuva 8). Esimerkiksi kuoleminen käsitellään Lyra-projektissa pelihahmon taitona, joka voi estää muiden kykyjen suorittamisen tagien perusteella. Epätäydellisellä toteutuksella pelaaja voisi vaikka liikuttaa kuollutta hahmoaan.



Kuva 8. Gameplay Ability -taidon blueprint-skripti, johon lisätään Ability-tagia.

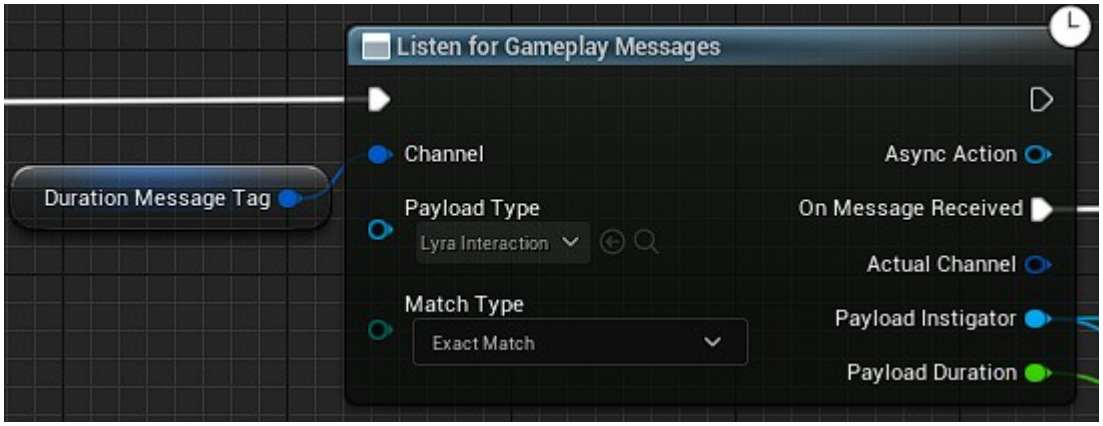
Pelaajan käyttöliittymän muokkaamiseksi Lyra-projektissa käyttöliittymäelementit tulee lisätä UIExtension Subsystem -alijärjestelmän Register Extension -funktiolla (kuva 9). Kyseinen funktio ottaa parametrina

Gameplay Tag -struktuurin, jonka avulla on mahdollista määrittää, mihin osaan käyttöliittymää elementti sijoitetaan.

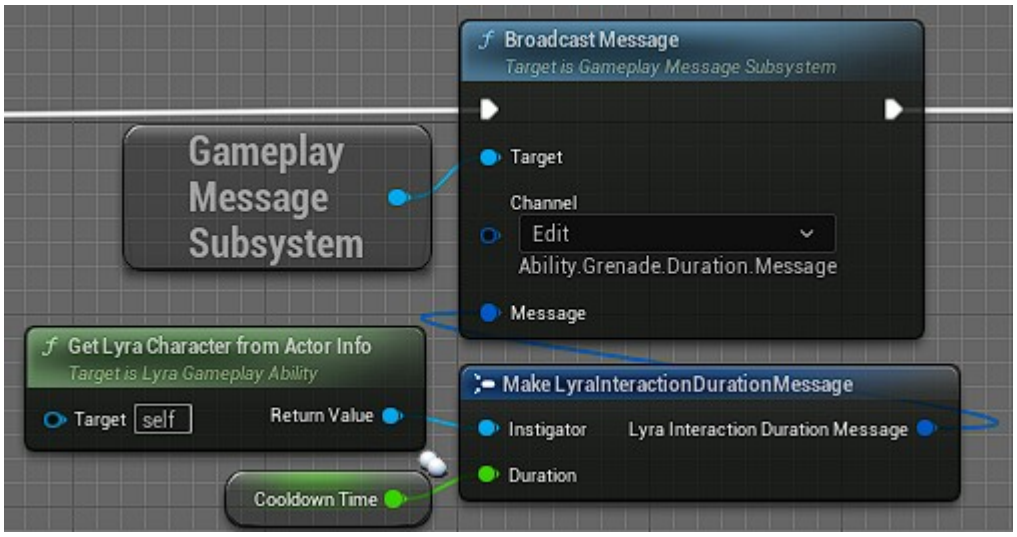


Kuva 9. Käyttöliittymäelementti liitetään pelaajan käyttöliittymään UIExtension-järjestelmällä.

Jotta käyttöliittymäelementti päivittyisi määriteltyjen tapahtumien pohjalta, objektin toiminnallisuutta on sidottava viesteihin Listen for Gameplay Messages -funktiolla (kuva 10). Näitä viestejä voi lähettää Gameplay Message Subsystem -järjestelmän Broadcast Message -metodilla (kuva 11). Jokainen viesti sisältää itse viestin datan tietostruktuurissa sekä tagin, joka määrittää käytetyn kommunikaatiokanavan. Jotta kuuntelijat voisivat onnistuneesti ottaa vastaan lähetetyn tiedon, niiden on käytettävä oikeaa kanavaa ja tietorakennetta.



Kuva 10. Toiminnallisuutta sidotaan viestin saapumiseen käyttöliittymäelementissä.



Kuva 11. Liukulukumuuttuja lähetetään viestissä Gameplay Message Subsystem -alijärjestelmällä.

Vaikka kyseisen projektin arkkitehtuuri voi perehtymättömälle olla monimutkainen, Lyra Starter Game hyödyntää Epic Gamesin uusimpia ja parhaita kehityskäytäntöjä. Näiden valmiiden kehitysmallien seuraaminen on myös tärkeää siistin ja ylläpidettävän koodin luomiseksi. Lisäksi seuraamalla kyseistä arkkitehtuuria pelaajien on mahdollista määritellä uudelleen syötteiden asetukset itse pelissä.

4.2 Liikemekaniikkojen toteutusvaatimukset

Opinnäytetyön inspiroimassa alkuperäisessä tehtävänannossa vaaditaan neljän eri liiketaidon toteuttamista esimerkkiprojektiin. Ensimmäinen näistä on hahmon teleportaatio kymmenen metriä eteenpäin. Etäisyyden asettaminen on yksinkertainen prosessi, sillä pelimoottorin projektin asetuksista pituuden mittayksikkö on asetettu senttimetreiksi. Hahmo tulee siis etenemään 1000 yksikköä.

Suoraan eteen liikkumisen voi tulkita liikkumiseksi pitkin hahmon kehosta eteenpäin menevää vektoria, mutta tällä toteutuksella pelaaja ei etäsiirry katsomansa suunnan mukaan. Tämä voi tuntua vähemmän intuitiiviselta kolmannen persoonan ammuntapelissä, ja lisäksi pelaajahahmo ei voi tällaisella teleportaatiolla liikkua ylemmäs tai alemmas ilman koko kehonsa kiertämistä.

On siis syytä toteuttaa kyseinen liikemekaniikka hyödyntämällä esimerkiksi Control Rotation -muuttujaa tai pelihahmon kameraa.

Toinen huomioitava kohta etäsiirtymisen toteutuksessa on kiinteiden esineiden sisälle ilmestyminen. Unreal Engine kuitenkin tarjoaa valmiita toiminnallisuutta juuri tämän välttämiseksi. Mikäli nämä työkalut eivät kuitenkaan tuota täysin haluttua lopputulosta, esineihin törmäämisen voi tarkastaa muilla funktioilla ja siirtymisen kohdetta säätää tämän pohjalta. Lopuksi on tärkeää varmistaa, että pelaaja ei voi kaukosiirtyä esimerkiksi pelin tason alapuolelle. Tämän mahdollistamiseksi tulisi esimerkiksi tarkastaa onko teleportaatiokohteen alla kiinteää objektia.

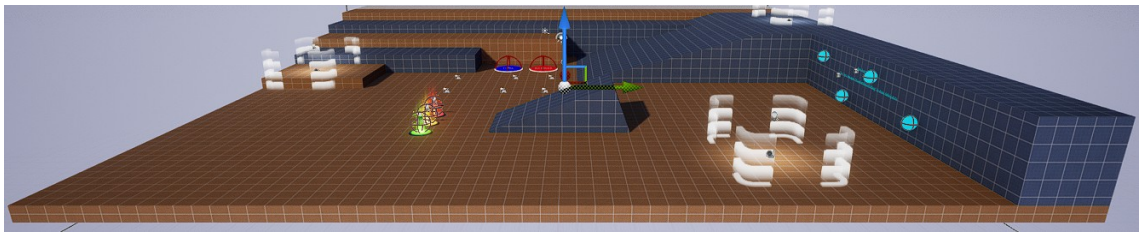
Toinen vaadittava liikemekaniikka on pelaajahahmon tilan, kuten sijainnin ja jopa elinpisteiden kelaaminen taaksepäin ajassa tiettyyn hetkeen asti. Kelauksen tulee toimia vaikuttamatta muuhun ympäristöön tai muihin pelaajiin. Aikamatkustuksen kohdehetken tulee olla vain muutama sekunti menneisyydessä, mutta kyvyn uudelleenaktivointiin vaadittu ajanjakso voi olla pidempi. Lisäksi tietoa kelaustaidon tilasta on pystyttävä näyttämään pelaajan ruudulla.

Perustapa toteuttaa kyseinen mekaniikka on tallentamalla pelaajahahmon tila tai muutokset heidän tilassa tihein väliajoin. Tämä aikaväli voisi pohjautua ruudunpäivitysnopeuteen, mutta tämä arvo voi vaihdella pelin aikana sekä laitteistojen ja asetusten välillä. Tilan tallentaminen kannattaa siis käsitellä kiinteällä sekunteihin perustuvalla ajastimella, ja myöhemmin tilojen välillä voisi myös interpoloida.

Lopuksi peliin tulisi toteuttaa seinäjuoksu- ja hyppymekaniikat. Pelaajan on kyettävä juoksemaan seinällä määritellyn ajan hyppäämällä seinään päin ja pitämällä hyppypainiketta alhaalla. Seinähypyllä hahmo voisi saada lisää korkeutta kimpoamalla seinästä. Näiden taitojen toteuttaminen on odotettavasti monimutkaisempaa, mikäli toteutuksen halutaan pelaajan päässä tuntuvan ja näyttävän suhteellisen luontaiselta.

Olisi suotavaa, että näiden mekaniikkojen toteuttaminen ei vaatisi ominaisuuksien lisäämistä pelin kenttien seinille. Sen sijaan juoksemiselle sekä hyppäämiselle otolliset pinnat ja kulmat voisi tarkastaa esimerkiksi LineTrace-funktiolla. Lisäksi seinillä liikkumista varten tulisi tarkastaa onko pelaaja ilmassa ja liikkuuko hän tarpeeksi nopeasti. Pelihahmon ei myöskään tulisi pystyä seinäjuoksemaan ja -hyppimään esimerkiksi muiden pelaajien kautta.

Mekaniikkojen testaamiseksi hyödynnetään Lyra-projektin tarjoamaa L_ShooterGym-tasoa (kuva 12). Tämä kenttä on varsin sopiva liikkumisen tutkimiseen sillä se sisältää seiniä, eri korkuisia alustoja ja kaltevia pintoja. Jokaisen mekaniikan kokeilua varten ei ole kehitetty tarkempaa suunnitelmaa ja tuloksia myös analysoidaan enemmän kvalitatiivisesti kuin kvantitatiivisesti.



Kuva 12. Lyra-projektin sisältämä L_ShooterGym-kenttä.

Lopuksi client-side prediction ja server-reconciliation -tekniikoiden testaamiseksi peliä pelataan asiakkaan roolissa simuloiden verkkopakettien liikennettä 500 millisekunnin viiveellä sekä 30% vaihtelulla. Nämä arvot asetetaan yksinkertaisesti editorin komentokehotteesta. Mikäli liikkumiskyvyt eivät heti aktivoidu syötteestä asiakkaan puolella tai niiden suorittaminen kumotaan, liikemekaniikkojen toteuttamisen katsotaan olevan virheellinen.

Työn yleisenä teemana ja tavoitteena on siis hyödyntää tietämystä verkkomoninpelien arkkitehtuureista responsiivisten liikemekaniikkojen luomiseksi. Sulavan nettitoiminnallisuuden toteuttamisen lisäksi työssä kuitenkin painotetaan myös itse näiden mekaniikkojen tuntua pelissä sekä kehitystyön suorittamista monimutkaisen arkkitehtuurin käytäntöjen mukaisesti.

5 Liikemekaniikkojen toteutus

5.1 Teleportaatio Gameplay Ability -järjestelmällä

Lyra-projektissa lopulliset pelaajahahmon käyttämät Gameplay Ability -taidot ovat toteutettu blueprint-skripteinä, mutta niiden on mahdollista periä C++-luokista. Opinnäytetyötä varten luodun teleportaatiotaidon toiminnallisuus toteutetaankin pääosin C++-kielellä sen siisteyden sekä sen tarjoaman FindTeleportSpot-funktion vuoksi.

Teleportaation suorittaminen alkaa ActivateAbility-metodissa kohdesijainnin laskemisella (listaus 2). Lähtökohtana käytetään itse pelaajahahmon sijaintia, mutta suunnan määrittämisessä haetaan pelaajan näkökulma hahmoa ohjaavan controller-objektin GetPlayerViewPoint-metodilla. Näin pelaaja pystyy etäsiirtymään katsomansa suunnan mukaan.

```
// Get the player view point for line-trace later
// and the rotation for the direction to the teleport target
FVector ViewPointLocation;
FRotator ViewPointRotation;
GetControllerFromActorInfo()->GetPlayerViewPoint(ViewPointLocation,
ViewPointRotation);

// Calculate the (initial) teleport destination
FVector TeleportDestination =
            ActorInfo->AvatarActor->GetActorLocation() +
            ViewPointRotation.Vector() * TeleportDistance;
```

Listaus 2. Teleportaation kohdesijainti lasketaan ActivateAbility-metodissa.

Loppusijainnin laskettua on mahdollista tarkistaa, onko objektin etäsiirtäminen mahdollista siten että se ei ilmestyisi kiinteän esineen sisälle (listaus 3). Tätä varten Unreal Engine tarjoaa FindTeleportSpot-funktion, joka voi myös tehdä pieniä säätöjä kohdevektoriin. Samalla koodissa myös luodaan jana pelaajan kamerasta näkökulman suuntaan, jotta poikkeustilanteissa hahmo käyttäisi vaihtoehtoista loppusijaintia, eli ilmestyisi pelaajan tähtäämän kiinteän esineen lähelle.

```
// Check if the teleport destination works and possibly adjust it
bool bCanTeleportToLocation = GetWorld()->FindTeleportSpot(ActorInfo-
>AvatarActor.Get(), TeleportDestination, ActorInfo->AvatarActor-
>GetActorRotation());

// Do a line-trace from the player's viewpoint to the direction of the
camera
FHitResult StraightMoveResult;
const uint16 LineTraceDistance = 1500;
FVector LineTraceEnd = ViewPointLocation + ViewPointRotation.Vector() *
LineTraceDistance;
GetWorld()->LineTraceSingleByChannel(
            StraightMoveResult, ViewPointLocation, LineTraceEnd,
ECollisionChannel::ECC_Visibility);
```

Listaus 3. Teleportaation kohdevektori tarkistetaan ja vaihtoehtoista loppusijaintia varten luodaan jana alkaen pelaaja kamerasta.

Mikäli etäsiirtymisen kohdesijainti ei ollut pätevä, pelaaja yritetään siirtää vaihtoehtoiseen sijaintiin (listaus 4). Lopullinen teleportaatio tapahtuu taitoa varten luodun kyvyn Teleport-metodissa, jossa myös kyvyn suorittaminen asetetaan päättyneeksi (listaus 5). Funktion voi laittaa etäsiirtämään objektin suoraan tai pienellä säädöllä riippuen siitä onko kohdesijainti linetrace-funktion osumapiste. Linetrace-funktiota käyttäessä teleportaatiokohdetta asetetaan alemmas pelaajahahmon keskipisteen ja hänen silmänsä korkeuden erotuksen verran, mikäli kohde on pelihahmoa korkeammalla (listaus 6). Näin pelaaja ei päädy yllättävän korkealla etäsiirtyessä hänen edessään olevan seinän viereen.

```
// If the teleport destination did not work,
// try to teleport to the impact point of the previous line-trace (with some
possible adjustments)
if (!bCanTeleportToLocation) {
            Teleport(Handle, ActorInfo, ActivationInfo,
StraightMoveResult.ImpactPoint, true);
            return;
}
```

Listaus 4. Hahmo yritetään siirtää vaihtoehtoiseen sijaintiin, jos ensimmäinen teleportaation kohde ei ollut kelpaava.

```
void ULyraGameplayAbility_Teleport::Teleport(const
FGameplayAbilitySpecHandle Handle, const FGameplayAbilityActorInfo*
ActorInfo, const FGameplayAbilityActivationInfo ActivationInfo, FVector
DestLocation, bool bUsedLineTrace)
{
            // If the teleport destination is the impact point of a line
trace,
            // account for the difference in the actor's eye location
            if (bUsedLineTrace) {
                        AdjustImpactPoint(DestLocation.Z, ActorInfo-
>AvatarActor.Get());
            }
            // Teleport to the destination with given parameters and actor's
current rotation
            FRotator DestRotation = ActorInfo->AvatarActor-
>GetActorRotation();
            const bool bIsATest = false;
            ActorInfo->AvatarActor->TeleportTo(DestLocation, DestRotation,
bIsATest, !bUsedLineTrace);

            // Finally end the ability
            const bool bWasCancelled = false;
            const bool bReplicateEndAbility = true;
            EndAbility(Handle, ActorInfo, ActivationInfo,
bReplicateEndAbility, bWasCancelled);
}
```

Listaus 5. Teleportaatiotaidon luokkaan luotu lopullinen Teleport-metodi.

```
void ULyraGameplayAbility_Teleport::AdjustImpactPoint(double& Z, AActor*
ActorToTeleport)
{
            // If the line-trace impact point is above the player's location
            // account for the difference in eye location
            float ActorZLocation = ActorToTeleport->GetActorLocation().Z;
            if (Z > ActorZLocation) {
                        FVector ActorEyeLocation;
                        FRotator ActorEyeRotation;
                        ActorToTeleport-
>GetActorEyesViewPoint(ActorEyeLocation, ActorEyeRotation);

                        float ActorEyeLocationZDifference =
(ActorEyeLocation.Z - ActorZLocation);
                        Z = Z - ActorEyeLocationZDifference;
            }
}
```

Listaus 6. Ero pelaajan keskipisteen ja silmien sijainnissa huomioidaan AdjustImpactPoint-funktiossa.

Jos pelaajahahmosta on näkölinja määränpäähän teleportaatioetäisyyden sisällä, hahmon sallitaan siirtyvän suoraan sijaintiin (listaus 7). Näin pelaaja pystyy tahallisesti etäsiirtymään esimerkiksi kuilun yläpuolelle vain, jos hän pystyy ensin näkemään teleportaatiokohteensa.

```
// If the previous line trace did not hit anything,
// just teleport to the destination
if (!StraightMoveResult.bBlockingHit) {
            Teleport(Handle, ActorInfo, ActivationInfo,
TeleportDestination);
            return;
}

// If the previous line trace did hit something
// but outside the teleport range,
// just teleport to the destination
float DistanceToHit = FVector::Distance(ActorInfo->AvatarActor-
>GetActorLocation(), StraightMoveResult.ImpactPoint);
if (DistanceToHit >= TeleportDistance) {
            Teleport(Handle, ActorInfo, ActivationInfo,
TeleportDestination);
            return;
}
```

Listaus 7. Hahmo siirretään suoraan näkemäänsä teleportaatiokohteeseensa.

Muissa tilanteissa on tarkastettava, että kohdesijainnin alla on maata, johon pelaaja voi laskeutua. Tätä varten teleportaatiosijainnista piirretään jana määritellyn etäisyyden päähän suoraan alaspäin ja tarkistetaan osuiko viiva mihinkään kiinteään (listaus 8). Tähän käytetty etäisyys voisi olla vain mahdollisimman suuri luku, mutta tällöin pelaaja voisi vahingossa etäsiirtyä sijainteihin joista on erittäin pitkä matka maahan sekä putoamisvahingon vaara.

```
// Do a line-trace to check if there is ground beneath the teleport spot
FHitResult LandBelowResult;
FVector LandCheckEndLocation = TeleportDestination + FVector::DownVector *
LandCheckDistance;
GetWorld()->LineTraceSingleByChannel(LandBelowResult, TeleportDestination,
LandCheckEndLocation, ECollisionChannel::ECC_Visibility);

// If land was found, teleport
if (LandBelowResult.bBlockingHit) {
            Teleport(Handle, ActorInfo, ActivationInfo,
TeleportDestination);
            return;
}

// If all else fails, teleport the player next to the solid object ahead
Teleport(Handle, ActorInfo, ActivationInfo, StraightMoveResult.ImpactPoint,
true);
```

Listaus 8. Pelaaja etäsiirretään joko suoraan sijaintiin josta voi turvallisesti laskeutua tai edellä määritettyyn vaihtoehtoiseen kohteeseen.

Joka tapauksessa hahmo voi etäsiirtyä suoraan vain, jos jotain kiinteää maata löydetään. Muulloin käytetään edellä määritettyä vaihtoehtoista etäsiirtymäsijaintia. Näin pelaaja ei voi kaukosiirtyä esimerkiksi lattian tai seinien läpi kuilun yläpuolelle.

5.2 Teleportaatio liikekomponentilla

Jotta uusia liikemekaniikkoja voisi toteuttaa käyttämällä liikekomponenttia Gameplay Ability -järjestelmän sijasta, peliin on luotava uusi hahmon liikekomponentista perivä komponentti ja siihen FSavedMove_Character-tietorakenteesta perivä luokka. Tämä jälkimmäinen objekti käsittelee tarvittavaa tietoa hahmon liikkumisesta.

Teleportaatiota varten FSavedMove_Character-luokasta perivään FSavedMove_Lyra-nimiseen tietorakenteeseen lisätään bSavedWantsToTeleport-niminen kokonaislukumuuttuja, joka kertoo yrittääkö hahmo etäsiirtyä, sekä SavedTeleportDestination-vektori joka sisältää etäsiirtymisen kohdesijainnin (listaus 9). Hahmon liikekomponenttiin luodaan myös vastaavat TeleportDestination- ja bWantsToTeleport-muuttujat, joista jälkimmäinen voi olla pelkkä totuusarvo.

```
class FSavedMove_Lyra : public FSavedMove_Character {

            typedef FSavedMove_Character Super;

            uint8 bSavedWantsToTeleport : 1;
            FVector SavedTeleportDestination;

            virtual bool CanCombineWith(const FSavedMovePtr& NewMove,
ACharacter* InCharacter, float MaxDelta) const override;
            virtual void Clear() override;
            virtual uint8 GetCompressedFlags() const override;
            virtual void SetMoveFor(ACharacter* C, float InDeltaTime,
FVector const& NewAccel, class FNetworkPredictionData_Client_Character&
ClientData) override;
            virtual void PrepMoveFor(class ACharacter* C) override;
};
```

Listaus 9. FSavedMove_Character-luokasta perivä FSavedMove_Lyra, joka sisältää uusia liikemekaniikkoja varten tarvittavaa dataa.

Jotta FSavedMove_Lyra-rakennetta voitaisiin hyödyntää liikekomponentissa, komponenttiin tulee luoda asiakkaalle verkkotietoja edustavasta FNetworkPredictionData_Client_Character-luokasta perivä luokka (listaus 10). Yliluokkaan viittaavan konstruktorin lisäksi tähän määritetään uudelleen

AllocateNewMove-funktio, joka palauttaa osoittimen määritellyn tyyppiselle tallennettavalle liikkeelle (listaus 11).

```
class FNetworkPredictionData_Client_Lyra : public
FNetworkPredictionData_Client_Character
{
public:
            FNetworkPredictionData_Client_Lyra(const
UCharacterMovementComponent& ClientMovement);

            typedef FNetworkPredictionData_Client_Character Super;

            virtual FSavedMovePtr AllocateNewMove() override;
};
```

Listaus 10. FNetworkPredictionData_Client_Lyra-luokka lisättynä mukautetun liikekomponentin otsikkotiedostoon.

```
ULyraCharacterMovementComponent::FNetworkPredictionData_Client_Lyra::FNetwor
kPredictionData_Client_Lyra(const UCharacterMovementComponent&
ClientMovement)
            : Super(ClientMovement)
{
}

FSavedMovePtr
ULyraCharacterMovementComponent::FNetworkPredictionData_Client_Lyra::Allocat
eNewMove()
{
            return FSavedMovePtr(new FSavedMove_Lyra());
}
```

Listaus 11. FNetworkPredictionData_Client_Lyra-luokan konstruktori, ja ylikirjoitettu FSavedMove_Lyra-luokkaa hyödyntävä AllocateNewMove-metodi.

Tämä uusi FNetworkPredictionData_Client_Lyra-objekti voidaan hakea liikekomponentista GetPredictionData_Client-funktiolla, mikäli liikekomponentilla on pätevä omistaja (listaus 12). Jos asiakkaan ennustetiedoille ei ole vielä luotu objektia ajon aikana, ennustetieto asetetaan halutun tyyppiseksi objektiksi, jonka funktio lopuksi palauttaa.

```
FNetworkPredictionData_Client*
ULyraCharacterMovementComponent::GetPredictionData_Client() const
{
            check(PawnOwner != nullptr);

            if (ClientPredictionData == nullptr)
            {
                        ULyraCharacterMovementComponent* MutableThis =
const_cast<ULyraCharacterMovementComponent*>(this);
                        MutableThis->ClientPredictionData = new
FNetworkPredictionData_Client_Lyra(*this);
            }

            return ClientPredictionData;
}
```

Listaus 12. GetPredictionData_Client-metodi, jossa liikekomponentin asiakkaan puolen ennustetiedot määritetään.

FSavedMove_Lyra-luokasta on myös ylikirjoitettava metodeja, jotta tiedon replikointi toimisi oikein. SetMoveFor-metodia käytetään liikekomponentin arvojen kopioimiseksi FsavedMoved_Lyra-luokkaan, ja tätä tietoa vuorostaan käytetään korjausten tekemiseen (listaus 13).

```
void
ULyraCharacterMovementComponent::FSavedMove_Lyra::SetMoveFor(ACharacter* C,
float InDeltaTime, FVector const& NewAccel, class
FNetworkPredictionData_Client_Character& ClientData)
{
            // Set the saved move with the variables from the movement
component
            Super::SetMoveFor(C, InDeltaTime, NewAccel, ClientData);
            ULyraCharacterMovementComponent* CharacterMovement =
Cast<ULyraCharacterMovementComponent>(C->GetCharacterMovement());

            // Include teleportation
            bSavedWantsToTeleport = CharacterMovement->bWantsToTeleport;
            SavedTeleportDestination = CharacterMovement-
>TeleportDestination;
}
```

Listaus 13. Tallennetun liikkeen muuttujia asetetaan liikekomponentin mukaan SetMoveFor-funktioissa.

GetCompressedFlags on melko yksinkertainen get-funktio, joka palauttaa liikkumiseen liittyvät binäärimuuttujat tavun sisällä (listaus 14). Etäsiirtymisen kohdalla palautettavaan tavuun asetetaan myös teleportaatiota varten varattu binäärimuuttuja, mikäli hahmo yrittää etäsiirtyä.

```
uint8 ULyraCharacterMovementComponent::FSavedMove_Lyra::GetCompressedFlags()
const
{
            // Set the regular compressed flags
            uint8 Result = Super::GetCompressedFlags();

            // If we want to teleport, set the custom flag for teleporting
            if (bSavedWantsToTeleport) {
                        Result |= FLAG_Custom_0;
            }

            // Return the flags
            return Result;
}
```

Listaus 14. GetCompressedFlags-metodi palauttaa tietoa maksimissaan kahdeksasta eri liiketaidosta binäärimuuttujien muodossa.

UpdateFromCompressedFlags-metodi päivittää liikekomponentin tilan pelin vastaanotettua tallennetun liikkeen (listaus 15). Tätä funktiota hyödynnetään jatkuvasti palvelimella, mutta asiakkaalla kyseiselle metodille on tarvetta vain korjaustilanteissa. Tässä tapauksessa liikekomponentin bWantsToTeleport-muuttujan arvo asetetaan teleportaatiota varten varatun binäärimuuttujan mukaan, joka on parametrina saadun tavun sisällä.

```
void ULyraCharacterMovementComponent::UpdateFromCompressedFlags(uint8 Flags)
{
            // Update movement statuses from flags
            Super::UpdateFromCompressedFlags(Flags);

            // Include teleportion
            bWantsToTeleport = (Flags & FSavedMove_Character::FLAG_Custom_0)
!= 0;
}
```

Listaus 15. Liikekomponentin tila päivitetään UpdateFromCompressedFlags-funktiolla.

Koska vektorin tapaisia muuttujia ei voi säilöä binäärimuuttujina, teleportaation kohdesijainti tulee lähettää parametrina funktiossa, jota asiakas voi kutsua, mutta jonka vain palvelin suorittaa (listaus 16). Näin voidaan myös varmistaa, että asiakkaalla ja palvelimella on varmasti sama kohdesijainti. Tällaisen funktion luomiseksi otsikkotiedoston metodiin on lisättävä UFUNCTION-makro, joka sisältää Server-tarkenteen (listaus 17). Tällöin funktion C++-toteutukseen on lisättävä _Implementation-pääte. Makroon kannattaa lisätä myös Reliable-tarkenne, jotta funktio onnistuisi huolimatta kaistanleveydestä tai

verkkovirheistä. Etäsiirtymisen voisi suorittaa tällaisen metodin sisällä, mutta tällöin teleportaation suorittaminen kumottaisiin korjaustilanteissa.

```
void
ULyraCharacterMovementComponent::Server_SendTeleportDestination_Implementati
on(FVector SentTeleportDestination)
{
            // Set the teleport destination on the server
            TeleportDestination = SentTeleportDestination;
}
```

Listaus 16. C++-tiedostoon toteutettu metodi jolla teleportaatiokohde lähetetään palvelimelle.

```
UFUNCTION(Server, Reliable, WithValidation)
void Server_SendTeleportDestination(FVector SentTeleportDestination);
```

Listaus 17. Otsikkotiedostoon määritetty metodi, jolla teleportaatiokohde lähetetään palvelimelle.

Lisäksi UFUNCTION-makroon voidaan liittää WithValidation-määrite, jos lähetetyn teleportaatiokohteen pätevyys halutaan tarkastaa. Tätä varten metodista tulee luoda C++-tiedostoon toinen _Validate-loppuinen määritelmä (listaus 18).

```
bool
ULyraCharacterMovementComponent::Server_SendTeleportDestination_Validate(FVe
ctor SentTeleportDestination)
{
            // Checks for validating teleport destination can be added here
            return true;
}
```

Listaus 18. Funktio jolla palvelimelle lähetetyn teleportaation kohteen sijainnin pätevyys voidaan tarkastaa.

Teleportaatiokohteen sijainnin määrittäminen toimii samoin kuin Gameplay Ability -järjestelmän toteutuksessa pieniä eroja lukuun ottamatta; ActorInfo-parametrin sijasta liikekomponentissa viittauksen pelihahmoon voi saada GetOwner-metodilla. Kun pelaajahahmo viimein halutaan etäsiirtää, kohdesijainti lähetetään palvelimelle asiakkaalta ja liikekomponentin bWantsToTeleport-muuttuja asetetaan todeksi (listaus 19).

```
void ULyraCharacterMovementComponent::Teleport(bool bNoCheck)
{
	// If the teleport destination has not been checked,
	// check (and possibly adjust) it now
	if (!bNoCheck) {
		bool bFoundTeleportSpot = GetWorld()-
>FindTeleportSpot(GetOwner(), TeleportDestination, GetOwner()-
>GetActorRotation());
		// If no valid teleport spot was found,
		// do not teleport
		if (!bFoundTeleportSpot)
			return;
	}
	// Only send the destination to the server if this is the client
	if (PawnOwner->GetLocalRole() == ENetRole::ROLE_AutonomousProxy)
		Server_SendTeleportDestination(TeleportDestination);
	bWantsToTeleport = true;
}
```

Listaus 19. Metodi jolla lopullinen teleportaatiokohde lähetetään asiakkaalta palvelimelle ja ilmoitetaan halusta etäsiirtyä.

Itse teleportaatio suoritetaan OnMovementUpdated-funktiossa, kun bWantsToTeleport-muuttuja on asetettu todeksi (listaus 20). Kyseinen funktio laukaistaan jokaisen liikkeen päivityksen lopussa sekä asiakkaan että palvelimen puolella. Samalla kun pelaajalle asetetaan uusi sijainti, bWantsToTeleport-boolean asetetaan jälleen epätodeksi, jotta etäsiirtyminen suoritettaisiin vain yhdesti. Teleportaation kohdesijainnin voisi laskea sekä asiakkaan että palvelimen puolella tässä funktiossa, mutta nämä eri osapuolet saattavat tulla eri tuloksiin mikä johtaisi liikkeen korjaamiseen pelaajan puolella.

```
void ULyraCharacterMovementComponent::OnMovementUpdated(float DeltaTime,
const FVector& OldLocation, const FVector& OldVelocity)
{
	Super::OnMovementUpdated(DeltaTime, OldLocation, OldVelocity);
	if (bWantsToTeleport) {
		bWantsToTeleport = false;
		// Teleport to the destination with given parameters
		const bool bSweep = false;
		FHitResult* OutSweepHitResult = nullptr;
		GetOwner()->SetActorLocation(TeleportDestination,
bSweep, OutSweepHitResult, ETeleportType::TeleportPhysics);
	}
}
```

Listaus 20. OnMovementUpdated-metodi, jossa teleportaation tapaiset luodut liikkeet lopulta suoritetaan.

Teleportaation aloittavaan funktioon viitataan LyraCharacter-luokkaan luodussa metodissa (listaus 21), johon vuorostaan viitataan LyraHeroComponent-komponentissa projektin arkkitehtuurin mukaisesti (listaus 22).

```
void ALyraCharacter::Teleport()
{
	ULyraCharacterMovementComponent* LyraMoveComp =
CastChecked<ULyraCharacterMovementComponent>(GetCharacterMovement());
	LyraMoveComp->StartTeleport();
}
```

Listaus 21. Funktio LyraCharacter-luokassa joka aloittaa etäsiirtymisen liikekomponentilla.

```
void ULyraHeroComponent::Input_Teleport(const FInputActionValue&
InputActionValue)
{
	if (ALyraCharacter* Character = GetPawn<ALyraCharacter>())
	{
		Character->Teleport();
	}
}
```

Listaus 22. Teleportaatio aktivoidaan syötteeseen sidotussa metodissa LyraHeroComponent-komponentissa.

PrepMoveFor-metodia voidaan hyödyntää korjausten tekemisessä asettamalla liikekomponentin arvot tallennetun liikkeen mukaan (listaus 23). Etäsiirtymistä varten komponentin bWantsToTeleport-totuusarvo asetetaan tallennetun bSavedWantsToTeleport-muuttujan mukaan.

```
void
ULyraCharacterMovementComponent::FSavedMove_Lyra::PrepMoveFor(ACharacter* C)
{
	// Set the character movement component with variables from
saved move
	Super::PrepMoveFor(C);
	ULyraCharacterMovementComponent* CharacterMovement =
Cast<ULyraCharacterMovementComponent>(C->GetCharacterMovement());

	// Include teleportation
	CharacterMovement->bWantsToTeleport = bSavedWantsToTeleport;
	CharacterMovement->TeleportDestination =
SavedTeleportDestination;
}
```

Listaus 23. Liikekomponentin ja tallennetun liikkeen muuttujia asetetaan toistensa mukaan SetMoveFor- ja PrepMoveFor-funktioissa.

Eräät tallennettujen liikkeiden ylikirjoitettavat metodit tarjoavat tapoja kaistanleveyden säästämiseen. Clear-funktio yksinkertaisesti nollaa FSavedMove_Lyra-objektin datan siten, että objektia on mahdollista käyttää uudelleen. CanCombineWith tarkastaa onko nykyisen liikkeen ja uuden liikkeen

tiedot yhdistettävissä. Etäsiirtymisen kohdalla tässä tapauksessa käsitellään vain binääristä bSavedWantsToTeleport-muuttujaa (listaus 24).

```
bool ULyraCharacterMovementComponent::FSavedMove_Lyra::CanCombineWith(const
FSavedMovePtr& NewMove, ACharacter* InCharacter, float MaxDelta) const
{
            // Check if the current move and the next move can be combined
to save bandwidth
            FSavedMove_Lyra* NewLyraMove =
static_cast<FSavedMove_Lyra*>(NewMove.Get());

            // If both the current and new move do not have the same
teleport status,
            // the moves cannot be combined
            if (bSavedWantsToTeleport != NewLyraMove->bSavedWantsToTeleport)
            {
                        return false;
            }
            return Super::CanCombineWith(NewMove, InCharacter, MaxDelta);
}

void ULyraCharacterMovementComponent::FSavedMove_Lyra::Clear()
{
            // Reset all variables including teleporting
            Super::Clear();
            bSavedWantsToTeleport = 0;
            SavedTeleportDestination = FVector::ZeroVector;
}
```

Listaus 24. Kaistanleveyttä on mahdollista säästää yhdistämällä ja uudelleen käyttämällä dataa CanCombineWith- ja Clear-funktioilla.

Kaikki edellä mainitut funktiot eivät ole pakollisia liikkeiden replikoimiseksi asiakkaalta palvelimelle, mutta ne voivat kuitenkin tehdä pelikokemuksesta sulavamman mahdollistamalla liikkeiden korjaamisen sekä säästämällä kaistanleveyttä.

5.3 Ajan kelaaminen taaksepäin

Ajan kelaamisen mahdollistamiseksi pelissä on ensiksi säilöttävä kelattavan objektin tila tiheään tahtiin. Tämän voi toteuttaa asettamalla esimerkiksi hahmon liikekomponentin InitializeComponent-funktiossa ajastimen, joka määritetyn väliajoin aktivoi pelaajan tilan tallentavan funktion (listaus 25). Mikäli hahmon muita ominaisuuksia kuten elinpisteitä halutaan tallentaa, ajastin on silloin parempi laittaa pelihahmon luokan sisälle BeginPlay-tapahtumaan.

```
void ULyraCharacterMovementComponent::InitializeComponent()
{
            Super::InitializeComponent();

            // Set a timer to capture data for rewinding
            bool InbLoop = true;
            float InRate = (1.f / RewindCapturesPerSecond);
            int8 InFirstDelay = -1;
            GetOwner()->GetWorldTimerManager().SetTimer(RewindCaptureTimer,
this, &ULyraCharacterMovementComponent::CaptureRewindData, InRate, InbLoop,
InFirstDelay);
}
```

Listaus 25. Objektin tilaa jatkuvasti tallentava ajastin luodaan liikekomponentin InitializeComponent-metodissa.

Ajastimen käyttämä metodi lisää pelihahmon nykyisen sijainnin taulukkoon sekä poistaa vanhimman sijainnin, mikäli halutulle kestolle on jo tarpeeksi dataa (listaus 26). Tässä esimerkissä RewindCapturesPerSecond-luku määrittää montako kertaa sekunnissa tila tallennetaan, ja RewindLengthSeconds-muuttuja kertoo montako sekuntia menneisyyteen pelaaja voi kelata. RewindCapturesPerSecond-muuttujan arvo voidaan asettaa suuremmaksi, jos halutaan tarkempaa dataa, tai pienemmäksi, jos muistia ja prosessointitehoa halutaan säästää.

```
void ULyraCharacterMovementComponent::CaptureRewindData()
{
            // If there is already rewind data for the whole length of the
rewind,
            // remove the first index
            if (RewindTargetLocations.Num() >= (RewindLengthSeconds *
RewindCapturesPerSecond)) {
                        RewindTargetLocations.RemoveAt(0);
            }

            // Add the actor's current data to the rewind data array
            RewindTargetLocations.Add(GetOwner()->GetActorLocation());
}
```

Listaus 26. CaptureRewindData-funktio lisää pelihahmon sijainnin taulukkoon ja hävittää vanhimman sijainnin, jos dataa on tarpeeksi täyteen kelaukseen.

Kun kelaus halutaan käynnistää, ensiksi tarkastetaan, ettei kelaus ole jo käynnissä, ja että data kelausta varten on kelvollista (listaus 27). Dataa tallentava ajastin on myös pysäytettävä, jotta kelautuminen ei jatkuisi loputtomasti. Lopuksi liikekomponentin bWantsToRewind-muuttuja asetetaan todeksi, jotta kelausta suoritettaisiin OnMovementUpdated-metodissa.

```
void ULyraCharacterMovementComponent::StartRewind()
{
            // Do not rewind if already rewinding or if there is no rewind
data
            if (bWantsToRewind
                        || RewindTargetLocations.IsEmpty()) {
                        return;
            }

            // Pause capturing data for rewinding
            GetOwner()-
>GetWorldTimerManager().PauseTimer(RewindCaptureTimer);

            // Set the flag for wanting to rewind to true
            bWantsToRewind = true;
}
```

Listaus 27. Ajan taaksepäin kelaaminen käynnistetään StartRewind-funktiossa.

Itse hahmon sijainnin asettava koodi OnMovementUpdated-funktiossa on pitkälti samanlainen kuin teleportaation toteutus. Tässä voisikin suoraan hyödyntää TeleportDestination-muuttujaa sekä Server_SendTeleportDestination-metodia, mutta tällöin kelauksen tallentamien tietojen sekä niiden todentamisen mukauttaminen tulevaisuudessa voi olla monimutkaisempaa. Jos kelaamisen dataan halutaan sisällyttää muuta tietoa pelaajahahmon sijainnin lisäksi, ja tieto halutaan validoida eri tavalla kuin etäsiirtyessä, kelaukselle on luotava erilliset funktiot siistin koodin ylläpitämiseksi.

Kun objektin tilaa halutaan kelata taaksepäin, pelihahmon kohdesijainti asetetaan paikallisesti SetTargetRewind-funktiolla, ja asiakas lähettää palvelimelle tuotetun arvon etäproseduurikutsulla (listaus 28). Paikallisesti ohjattu pelihahmo voi olla asiakkaan, mutta myös suoraan listen server -palvelimella olevan pelaajan hallitsema objekti. Lopulta kohdesijaintiin siirrytään sekä lokaalisti että palvelimella.

```
// If the player wants to rewind, set the target location if locally
controlled
// and send the location to server if this is the client.
// Finally, move the owning actor to the target location
if (bWantsToRewind) {
            if (PawnOwner->IsLocallyControlled()) {
                        SetTargetRewindLocation();
                        if (PawnOwner->GetLocalRole() ==
ENetRole::ROLE_AutonomousProxy) {

            Server_SendRewindData(RewindTargetLocation);
                        }
            }
            GetOwner()->SetActorLocation(RewindTargetLocation, bSweep,
OutSweepHitResult, ETeleportType::TeleportPhysics);
}
```

Listaus 28. OnMovementUpdated-metodiin lisätty osio, joka suorittaa hahmon tilan kelaamisen.

Kelauksen kohdesijainnin laskemiseksi hyödynnetään RewindProgress-nimistä liukulukua; pyöristämällä alas kelaussijaintien taulukon viimeisen indeksin ja RewindProgress-muuttujan erotus saadaan ensimmäisen interpoloitavan elementin indeksi, jota edeltävää elementtiä käytetään toisena interpoloinnissa (listaus 29). Lisäksi kyseisen liukuluvun desimaalilukuja hyödynnetään kohdevektorin laskun alfa-arvona.

```
// Get the indexes the elements of which will be interpolated between
            int16 LerpIndexA = FMath::CeilToInt(LastRewindIndex -
RewindProgress);

            // Get the interpolation alpha value (decimals of Progress)
            float Alpha = RewindProgress - FMath::Floor(RewindProgress);

            // Set the rewind target location by interpolating between the
elements in the set indexes
            RewindTargetLocation =
FMath::Lerp(RewindTargetLocations[LerpIndexA],
RewindTargetLocations[LerpIndexA - 1], Alpha);
```

Listaus 29. Osuus SetRewindTargetLocation-metodista, jossa lasketaan kelauksen kohdesijainti.

RewindProgress-luku alkaa nollasta, ja siihen lisätään jokaisen päivityksen lopussa aika edelliseen päivitykseen kerrottuna tallennettavien tietojen maksimimäärällä, joka on vielä jaettu halutulla kokonaisen kelauksen vievällä ajalla (listaus 30). Tässä hyödynnetään globaalia DeltaTime-muuttujaa OnMovementUpdated-funktion tarjoaman DeltaTime-parametrin sijasta, koska tämä jälkimmäinen arvo eriää normaalista virkistysajasta asiakkaan puolella.

Mikäli esimerkiksi liikekomponentille asetettaisiin oma päivitystaajuus, tätä uutta arvoa tulisi hyödyntää globaalin DeltaTime-arvon sijasta. Loppujen lopuksi tämän ruudunpäivitysten väliseen aikaan perustuvan interpoloinnin ansiosta ajan kelaamisen suorittamisen tulisi toimia yhtä nopeasti sekä näyttää mahdollisimman sulavalta päivitysnopeudesta huolimatta.

```
// Increment RewindProgress so that a full rewind is performed in the
desired time
RewindProgress += GetWorld()->GetDeltaSeconds() * (RewindCapturesPerSecond *
RewindCaptureLengthSeconds) / RewindFullPerformLengthSeconds;
```

Listaus 30. RewindProgress-muuttujaa kasvatetaan SetRewindTargetLocation-funktion lopussa.

Mikäli kelaustaulukon viimeinen indeksi on saavutettu tai ohitettu, viimeinen kohdesijainti asetetaan suoraan ja ajan kelaaminen lopetetaan (listaus 31). Lisäksi RewindProgress-muuttuja asetetaan takaisin nollaksi, vanhat kelaustiedot poistetaan ja uusia tietoja aletaan jälleen tallentamaan. Koska tätä funktiota saatetaan kutsua uudelleen asiakkaan korjatessa liikkeitään, viimeinen kohdesijainti tulee asettaa vain, jos kelaustaulukko vielä sisältää elementtejä. Vaihtoehtoisesti kaikki sijainnit kelausta varten voitaisiin tallentaa FSavedMove_Lyra-objektissa, mutta tämä veisi enemmän muistia.

```
void ULyraCharacterMovementComponent::SetTargetRewindLocation(float
DeltaTime)
{
            // Get the last index of the rewind target locations
            int16 LastRewindIndex = RewindTargetLocations.Num() - 1;

            // If the current rewind index is greater or equal to the last
rewind data index
            // finish rewinding properly
            if (RewindProgress >= LastRewindIndex)
            {
                        // set the last rewind data (if not reconciliating)
                        if (!RewindTargetLocations.IsEmpty()) {
                                    RewindTargetLocation =
RewindTargetLocations[0];
                        }

                        // reset rewind progress to zero and stop rewinding
                        RewindProgress = 0;
                        bWantsToRewind = false;

                        // empty the rewind data array so that the same
rewind cannot be performed again
                        RewindTargetLocations.Empty();

                        // resume capturing new rewind data
                        GetOwner()-
>GetWorldTimerManager().UnPauseTimer(RewindCaptureTimer);

                        return;
            }
```

Listaus 31. Kohta SetTargetRewindLocation-funktiosta, jossa kelaus päätetään.

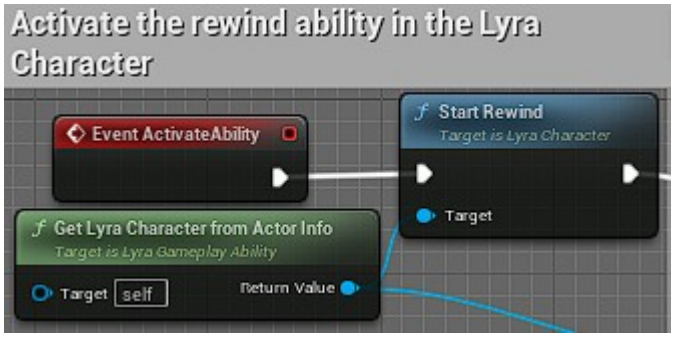
Samoin kuin etäsiirtymisen kanssa, FSavedMode_Lyra-luokkaan luodaan liikekomponentin bWantsToRewind-totuusarvoa vastaava bSavedWantsToRewind-tavumuuttuja, jota käytetään esimerkiksi SetMoveFor- ja PrepMoveFor-funktioissa. Muuttujia varten on myös varattu binääriluku, jota hyödynnetään GetCompressedFlags- ja UpdateFromCompressedFlags-metodeissa. Lisäksi FSavedMode_Lyra-luokkaan tallennetaan hahmon liikekomponentin RewindProgress-liukuluku, jotta kelauksen liike olisi sulavaa myös korjaustilanteissa.

Tällaista jatkuvaa liikettä sisältävää liikemekaniikkaa on hankala saada toimimaan sulavasti pelkällä Gameplay Ability -järjestelmällä, koska tällöin toiminnallisuutta ei voi helposti synkronoida asiakkaan ja palvelimen välillä OnMovementUpdated-funktion kaltaisessa metodissa. Gameplay Ability -taitoja voi kuitenkin hyödyntää yhdessä liikekomponentin kanssa, ja näin ajan kelaamisen vuorovaikuttamisen muiden kykyjen sekä käyttöliittymän kanssa voidaan määrittää helposti Lyra-projektissa.

Tässä esimerkissä hahmon liikekomponentin luokkaan on määritetty BlueprintPure-tarkenteella blueprint-skriptistä kutsuttava funktio, joka palauttaa kelauksen vievän ajan huomioiden kerätyn kelausdatan määrän (listaus 32). Kelauksen aktivointia varten on luotu BlueprintCallable-tarkenteen omaava metodi, joka laukaistaan taidon aktivoidessa (kuva 13).

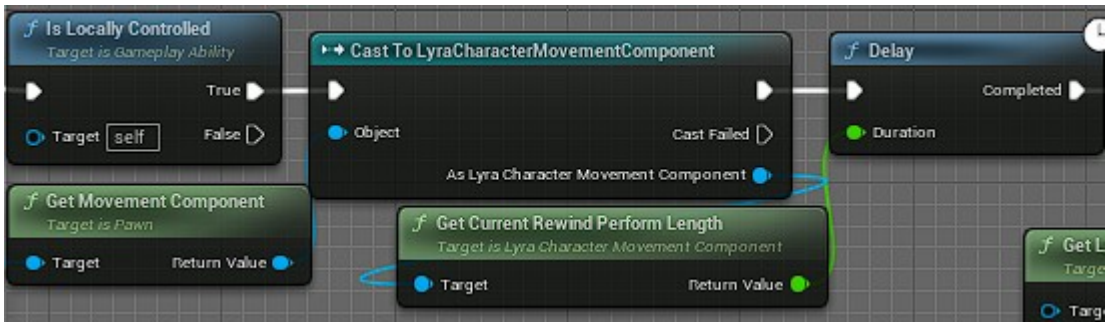
```
float ULyraCharacterMovementComponent::GetCurrentRewindPerformLength()
{
            // Return the amount of time it takes to perform a rewind
            // with current amount of captured rewind data
            float RewindDataCapturedRatio = (RewindTargetLocations.Num() /
(RewindCaptureLengthSeconds * RewindCapturesPerSecond));
            return RewindDataCapturedRatio * RewindFullPerformLengthSeconds;
}
```

Listaus 32. GetCurrentRewindPerformLength-funktio laskee kelauksen vievän ajan nykyisellä datan määrällä ja palauttaa tuloksen.

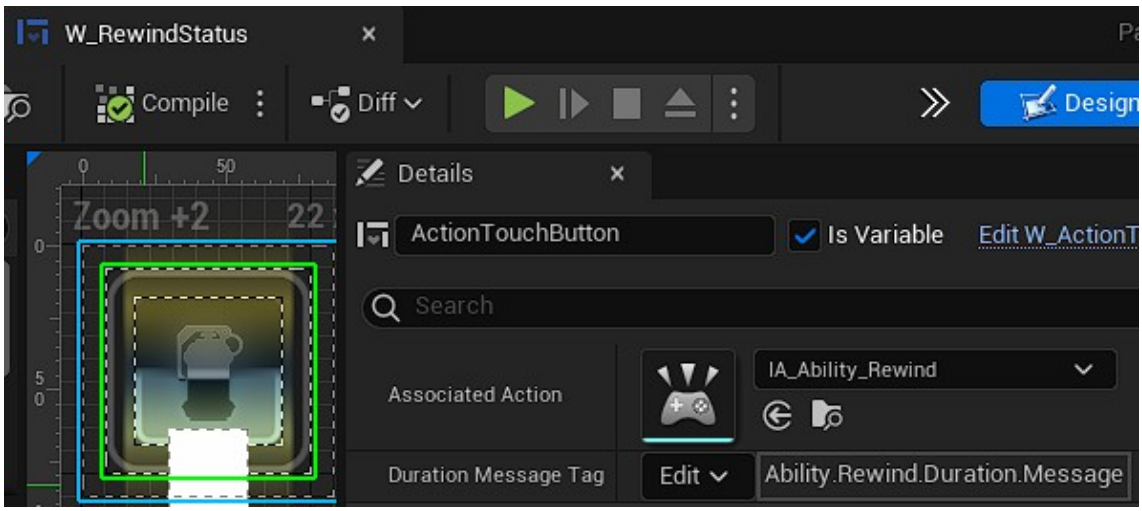


Kuva 13. C++-koodissa olevaa kelausfunktiota kutsutaan kelaustaidon blueprint-skriptin ActivateAbility-tapahtumassa.

Kun oikea aika on kulunut (kuva 14), taidon suorittaminen merkitään päättyneeksi, ja kelaamista koskeva käyttäliittymäefekti aktivoidaan. Tässä esimerkissä hyödynnetään mukautettua versiota kranaatille tarkoitetusta käyttöliittymäelementistä (kuva 15) joka kiinnitetään näytön vasemmassa alakulmassa varattuun tilaan taidon OnPawnAvatarSet-metodissa. Kyvyn päättäminen tietyn ajan kuluttua suoritetaan pelkästään lokaalisti, koska liikekomponentissa tallennetusta kelausdatasta pidetään tarkkaa lukua vain paikallisesti.



Kuva 14. Kelauksen loppumista odotetaan lokaalisti blueprint-skriptissä.



Kuva 15. Valmiista tuotoksesta kopioitu ajan kelauksen hyödyntämä käyttöliittymäelementti.

Hyödyntämällä Gameplay Ability -järjestelmää tällä tavalla, pelaaja voi nähdä miten pitkälle tietoa on tallennettu kelaamista varten, ja häntä myös estetään aktivoimasta muita taitoja kelauksen ollessa käynnissä.

6 Tulokset

6.1 Teleportaatio

Suhteellisen yksinkertaisen teleportaatiotaidon toteuttaminen Gameplay Ability -järjestelmällä oli sekä helppoa että intuitiivista. Tarkemmin sanottuna tämän kyvyn lisääminen verkkomoninpeliin ei tässä tapauksessa eronnut yksinpeliin toteuttamisesta yhtään. Kuitenkin testatessa tätä toteutusta suurella verkkopakettien viiveellä ja hahmon liikkuessa, korjaustilanteita esiintyi enemmän verrattuna liikekomponenttia hyödyntävään toteutukseen. Tämä todennäköisesti johtuu siitä, että hahmon liikekomponentissa etäsiirtyminen suoritetaan synkronoidussa OnMovementUpdated-funktiossa, jossa palvelin myös hyödyntää asiakkaan lähettämää kohdesijaintia. Lisäksi tässä teleportaation kohdevektori tallennetaan erikseen korjaustilanteita varten.

Vektorin voisi replikoida asiakkaalta palvelimelle myös Gameplay Ability -järjestelmällä. Tämän mahdollistamiseksi taidon ReplicationPolicy-muuttujan arvoksi tulee asettaa ReplicateYes. Lyhyen kokeilun perusteella vektorin lähettäminen ei kuitenkaan vähentänyt korjaustilanteiden määrää huomattavasti. Tulee kuitenkin ottaa huomioon, että suurta verkkopakettien viiveettä simuloidessa tällaista epäsynkronointia tapahtuu esimerkiksi myös Lyra-projektin omassa syöksymistaidossa.

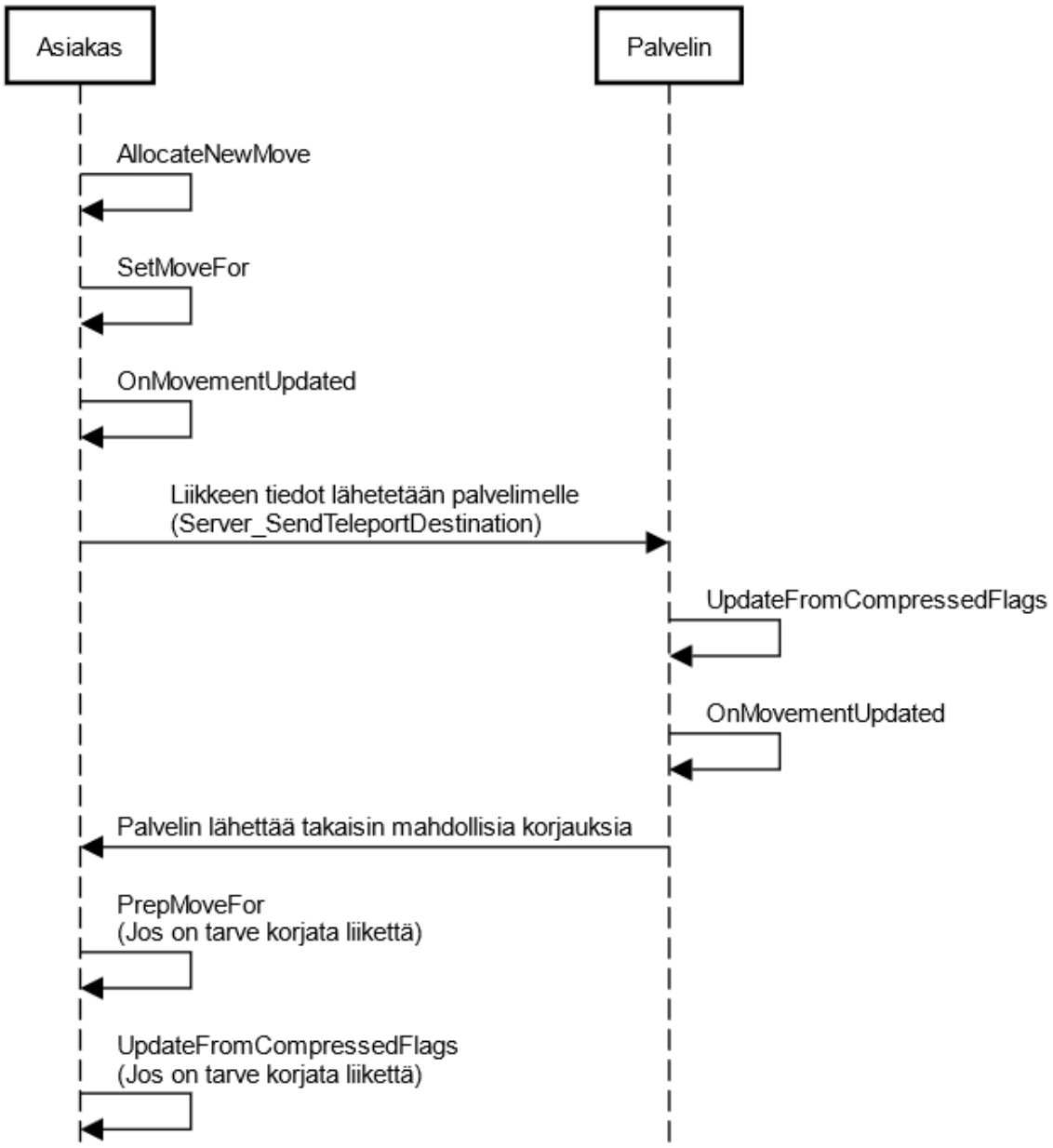
Joka tapauksessa teleportaatiomekaniikan toimintalogiikka tuntuu varsin luontevalta. Pelaajan etäsiirtymissijaintia rajoitetaan vain, jos hän ei suoraan näe kuilun yllä olevaa kohdesijaintia, tai jos ilmestymissijainti on kiinteän objektin sisällä. Tämä ei kuitenkaan tunnu liian rajoittavalta, koska pelaaja pystyy silti etäsiirtymään näkemiinsä sijainteihin niiden vaarallisuudesta huolimatta. Samalla estetään kuitenkin tilanteet, jossa pelihahmo ilmestyisi seinän tai lattian läpi tyhjyyteen, tai jossa hän jäisi jumiin tason geometriaan. Ja koska kohdesijainti perustuu kameran eikä pelaajahahmon orientaatioon, pelaaja pystyy intuitiivisesti etäsiirtymään katsomaansa suuntaan.

Kolmannen persoonan peleissä teleportaatiota objektien läpi ei voi kuitenkin rakentaa samalla tavalla kuin esimerkiksi aseella tähtäämistä, koska etäsiirtymisen kohde voi olla pelaajan näkemän objektin takana. Tällainen mekaniikka olisi siis yksinkertaisempaa toteuttaa ensimmäisen persoonan peliin.

C++-koodi osoittautui tämän mekaniikan rakentamisessa varsin sopivaksi valinnaksi. Tämä ei johdu ainoastaan sen siisteydestä, vaan myös sen tarjoamasta FindTeleportSpot-funktiosta, jolle ei löydy vastinetta blueprint-skripteistä. Kyseistä metodia hyödynnetään myös TeleportTo-funktiossa. Unreal Engine siis itsestään tarjoaa tapoja etäsiirtymiseen jäämättä kiinni objekteihin, mikä tekee tämän mekaniikan toteuttamisesta varsin kätevää. Muulloin koodissa pitäisi itse ottaa huomioon pelihahmon kiinteiden osien laajuudet, ja säätää kohdesijainti tämän mukaan.

Teleportaation toteutuksessa on kuitenkin paranneltavaa. Kehittäessä teleportaatiota ilmestyi tilanteita, joissa pelaajahahmo pystyi jäämään jumiin geometriaan tai jopa ilmestymään lattian läpi tasossa olevien pienien rakojen takia. Koodiin lisättiin tarkistus tämän estämistä varten, jonka seurauksena teleportaatio ei aktivoidu ollenkaan tähdätessä tiettyjen objektien välisiin rakoihin. Tämän voi nähdä ongelmana kenttäsuunnittelussa, mutta kyseinen ongelma voidaan mahdollisesti ratkaista myös lisäämällä koodia tätä tapausta varten.

Vaikka tämä teleportaatiomekaniikka on suhteellisen yksinkertainen, sen toteuttaminen liikekomponenttiin verkkopeliä varten oli aluksi erittäin monimutkaista, mutta kokonaisuutena erittäin opettava kokemus. Tämän mekaniikan rakentaminen vaati monia lisäaskeleita verrattuna Gameplay Ability -järjestelmän hyödyntämiseen, ja koodin ymmärtäminen vaatii syvempää ymmärrystä Unreal Engine -moottorin verkkopelien arkkitehtuurista (kuvio 3). Lopputulos oli kuitenkin vakaampi verkkopakettien viivettäkin simuloidessa, ja esimerkiksi liikkeen korjaamistapaa voi helposti mukauttaa tällä menetelmällä.



Kuvio 3. Kaavio liikkumisen toiminnasta Unreal Engine -pelimoottorin verkkopelissä.

6.2 Ajan kelaaminen taaksepäin

Jo alussa oli hyvin ilmeistä, että Gameplay Ability -järjestelmä ei soveltunut hyvin jatkuvan liikkeen toteuttamista varten; nettipelissä lopputulos oli erittäin epätasainen, vaikka toteutus olisikin melkein identtinen liikekomponentin kanssa. Ja vaikka ajan kelaamisesta on monia esimerkkitoteutuksia yksinpelejä varten, moninpeleissä tällainen mekaniikka on harvinaisempi. Ajan kelaamisen

hienosäätäminen vei siis aikansa, vaikka Unreal Enginen verkkoarkkitehtuuri oli suhteellisen tuttu jo tässä vaiheessa.

Tämä mekaniikka oli hankalampi toteuttaa siististi koska asiakas ja palvelin pystyivät helposti epäsynkronoimaan, mikä näyttäytyisi erittäin epäsulavalta liikkeeltä pelaajan päässä. Tässä tapauksessa oli erityisen tärkeää, että asiakas ja palvelin varmasti käyttäisivät identtistä dataa liikkumiseen, sekä liikkuisivat synkronoidusti esimerkiksi OnMovementUpdated-funktion avulla.

Tärkeimmät valinnat liittyen ajan kelaamisen toteuttamiseen liittyivät tallennettaviin muuttujiin sekä asiakkaalta palvelimelle lähetettävään dataan. bWantsToRewind-muuttujan lisäksi SavedRewindProgress-muuttujan säilöminen liikkeiden uusimista varten oli paras valinta sulavuuden varmistamiseksi. Kelauksen kohdesijainnin säilöminen ei testaamisen pohjalta vaikuttanut liikkeen tasaisuuteen korjaustilanteissa.

Yksittäisen sijainnin lähettämisen sijaan asiakas voisi lähettää kaikki nauhoittamansa kelauksen tiedot ja antaa palvelimen käydä läpi tätä dataa erikseen. Tällainen toteutus voi kuitenkin helposti lisätä korjaustilanteiden määrää moninkertaisesti.

Vaikka lopuksi Gameplay Ability -järjestelmä ei toiminut tarpeeksi sulavasti tällaisen mekaniikan toteuttamiseksi, se oli silti mahdollista yhdistää liikekomponentin toiminnallisuuden kanssa taitojen välisen vuorovaikutuksen määrittämiseksi sekä käyttöliittymäefektien näyttämiseksi. Näin pelaaja ei voi esimerkiksi hyökätä, syöksyä tai etäsiirtyä kelauksen ollessa käynnissä, ja käyttöliittymässä näkyy, kuinka suuri osa kelausdatasta on kerätty verrattuna maksimiin. Tämä efekti kuitenkin päivittyy vain kelauksen suoritettua eikä taidon aktivoitua, joten käyttöliittymän palkki voi täydentyä, vaikka kelaus olisi aktivoitu.

Koska taidon suoritus suoraan hyödyntää mukautettua liikekomponenttia, tämä kyky toimii vain niillä peliobjekteilla jotka omistavat kyseisen komponentin. Yksi Gameplay Ability -järjestelmän eduista mitätöityy, kun mikä tahansa hahmo ei voi suorittaa taitoa.

Kelaukseen, kuten oikeastaan kaikkeen jatkuvaan liikkeeseen, vaikuttaa suuresti viive. Simuloidessa 500 millisekunnin viivettä selviä korjaustilanteita ja poikkeavuuksia esiintyi aika-ajoin. Tämä viive on kuitenkin erittäin korkea ja vähentäessä viiveen esimerkiksi sataan millisekuntiin, mikä on vieläkin suhteellisen paljon, nämä selvät poikkeukset vaikuttavat katoavan kokonaan.

Vaikka interpolointi tallennettujen kelaussijaintien välillä tekee liikkeestä sulavaa, nykyisellä toteutuksella tämä aiheuttaa pienen ristiriidan teleportaation kanssa. Kelatessa hahmo voi siis ilmestyä etäsiirtymisen lähtö- ja loppukohdan väliin, vaikka hän ei koskaan ollut tässä sijainnissa. Tämä kuitenkin tapahtuu niin nopeasti, että sitä ei voi visuaalisesti havaita, mutta pelaajahahmo voi esimerkiksi aktivoida sijaintiin sidottuja tapahtumia. Kyseinen virhe voi kuitenkin vaikuttaa vain erittäin tarkoissa olosuhteissa, joten sen korjaaminen ei ole tälle opinnäytetyölle prioriteetti.

Lopuksi testauksessa esiintyi yksi harvinainen virhe, mikäli kyvyn lopetus oli sidottu liikekomponentissa aktivoitavaan funktiodelegaattiin: joskus kelaus ei päättynyt kunnollisesti, jonka seurauksena Gameplay Ability -taitoa ei voinut aktivoida uudelleen. Tätä vikaa ei pystytty säännöllisesti toistamaan, joten paras tapa toistaa kyseinen tapaus oli yksinkertaisesti aktivoimalla kelaustaito jatkuvasti, kunnes vika ilmeni. Taidon päättämisen sitominen ajan kulumiseen kuitenkin korjasi ongelman.

6.3 Seinäjuoksu- ja hyppy

Seinäjuoksu- ja hyppymekaniikan toteuttamiselle ei tätä työtä varten riittänyt aikaa, mutta kyseisten mekaniikkojen luomiseen verkkomoninpeliä varten löytyy helpommin lähteitä internetistä verrattuna esimerkiksi ajan kelaamiseen. Seinillä juokseminen esiintyykin monissa verkkoammuntapeleissä kuten Titanfall 2, Call of Duty: Black Ops III ja Warframe-peleissä.

Yksi tapa toteuttaa seinäjuoksu on luomalla liikekomponenttiin mukautettu movement mode -tila. Hahmon tila tarkastetaan ylikirjoitetussa UpdateCharacterStateBeforeMovement-funktiossa, ja mikäli hän on putoamassa, seinällä juokseminen yritetään aktivoida. Kyseisen liikkeen aktivoimiselle voidaan määrittää esimerkiksi minimi vaadittu nopeus ja etäisyys maasta, jotta seinäjuoksu ei aktivoituisi liian herkästi. (delgoodie 2023.)

Yrittäessä aloittaa seinällä liikkumista, liikekomponentissa tarkistetaan linetrace-funktion avulla, onko pelaajan kummankaan puolen lähellä seinää (listaus 33). Mikäli pinta löytyy ja peliobjekti menee sitä kohti, liikekomponenttiin luodun totuusarvomuuttujan arvo asetetaan sen mukaan, kummalla puolella hahmoa seinä on. Tälle muuttujalle luodaan myös binäärimuuttuja tallennettavia liikkeitä varten. (delgoodie 2023.)

```
// Left Cast
GetWorld()->LineTraceSingleByProfile(WallHit, Start, LeftEnd, "BlockAll",
Params);
if (WallHit.IsValidBlockingHit() && (Velocity | WallHit.Normal) < 0)
{
            Safe_bWallRunIsRight = false;
}
// Right Cast
else
{
            GetWorld()->LineTraceSingleByProfile(WallHit, Start, RightEnd,
"BlockAll", Params);
            if (WallHit.IsValidBlockingHit() && (Velocity | WallHit.Normal)
< 0)
            {
                        Safe_bWallRunIsRight = true;
            }
            else
            {
                        return false;
            }
}
```

Listaus 33. Pintaa yritetään löytää pelaajahahmon vasemmalta ja oikealta puolelta hyödyntäen linetrace-funktiota (Listaus: delgoodie).

Tämän jälkeen pelaajan nopeus projisoituna tason normaaliin tarkastetaan, ja jos tämä projisoitu nopeus on tarpeeksi suuri, se asetetaan pelihahmon uudeksi velocity-arvoksi, ja movement mode -tila muutetaan vastamaan seinäjuoksua (listaus 34). Tässä kohdassa on myös mahdollista rajata objektin vauhtia Z-akselilla, mikäli pelaajan ei haluta tippuvan tai nousevan liian nopeasti. (delgoodie 2023.)

```
FVector ProjectedVelocity = FVector::VectorPlaneProject(Velocity,
WallHit.Normal);
if (ProjectedVelocity.SizeSquared2D() < pow(MinWallRunSpeed, 2)) return
false;

// Passed all conditions
Velocity = ProjectedVelocity;
Velocity.Z = FMath::Clamp(Velocity.Z, 0.f, MaxVerticalWallRunSpeed);
SetMovementMode(MOVE_Custom, CMOVE_WallRun);
```

Listaus 34. Seinäjuoksu aloitetaan, jos hahmon seinälle projisoitu nopeus on tarpeeksi suuri (Listaus: delgoodie).

Seinällä juoksua päivitetään ylikirjoitetun PhysCustom-fysiikkafunktion kautta, kun mukautettu movement mode -tila on seinäjuoksutilassa. Kyseinen metodi ottaa parametrina deltaTime-liukulukumuuttujan sekä Iterations-kokonaislukumuuttujan. Tässä voi käyttää mallina esimerkiksi hahmon liikekomponentissa valmiina oleva PhysWalking-funktiota (listaus 35). (delgoodie 2023.)

```
float remainingTime = deltaTime;

// Perform the move
while ( (remainingTime >= MIN_TICK_TIME) && (Iterations <
MaxSimulationIterations) && CharacterOwner && (CharacterOwner->Controller ||
bRunPhysicsWithNoController || HasAnimRootMotion() ||
CurrentRootMotion.HasOverrideVelocity() || (CharacterOwner->GetLocalRole()
== ROLE_SimulatedProxy)) )
{
            Iterations++;
            bJustTeleported = false;
            const float timeTick = GetSimulationTimeStep(remainingTime,
Iterations);
            remainingTime -= timeTick;
```

Listaus 35. Liikettä laskevan while-silmukan alku PhysWalking-fysiikkafunktiossa (Listaus: Epic Games).

Ennen liikkeen päivittämistä on hyvä tarkastaa, että saatu deltaTime on minimiä suurempi, ja että liikekomponentin omistama hahmo on pätevä. Tämän jälkeen liikettä lasketaan while-silmukan sisällä, jossa ensin määritetään time step -arvo deltaTime-muuttujan sekä iteraatioiden määrän perusteella. Jokaisen silmukan alussa deltaTime-luvusta vähennetään time step -arvo, ja jos tämä erotus alittaa minimimäärän tai iteraatioiden määrä ylittää maksimin, silmukan suoritus päättyy. (delgoodie 2023.)

Päivityksen alussa ensin tarkastetaan, onko seinää vielä pelaajan lähellä, ja yrittääkö hän vetää siitä pois (listaus 36). Tämä jälkimmäinen voidaan tarkistaa hyödyntämällä liikekomponentin kiihdytyksen arvoa. Seinäjuoksun lopettamiseksi movement mode -tilalle asetetaan arvo MOVE_Falling, jolloin pelihahmo alkaa tippua. Mikäli seinällä juoksemista jatketaan, pelaajahahmon kiihtyvyys sekä nopeus asetetaan jälleen projisoinnin avulla, ja mikäli nopeus on liian pieni, putoaminen aloitetaan. (delgoodie 2023.)

```
bool bWantsToPullAway = WallHit.IsValidBlockingHit() && !
Acceleration.IsNearlyZero() && (Acceleration.GetSafeNormal() |
WallHit.Normal) > SinPullAwayAngle;
if (!WallHit.IsValidBlockingHit() || bWantsToPullAway)
{
            SetMovementMode(MOVE_Falling);
            StartNewPhysics(remainingTime, Iterations);
            return;
}
```

Listaus 36. Seinäjuoksu muuttuu putoamiseksi, mikäli pintaa ei enää ole lähettyvillä, tai jos pelaaja vetää siitä pois (Listaus: delgoodie).

Itse liikkuminen tapahtuu SafeMoveUpdatedComponent-funktion avulla, jonka Delta-nimiseksi parametriksi asetetaan time step -arvo kerrottuna liikekomponentin nopeudella (listaus 37). Mikäli tämän tulon arvo on liian pieni, tai jos komponentti ei liikkumisyrityksen jälkeen siirtynyt vanhasta paikastaan, silmukan suorittaminen lopetetaan. Hahmoa voi myös varoilta liikuttaa kohti seinää käyttämällä SafeMoveUpdatedComponent-metodia siten, että Delta-parametri asetetaan seinän normaalin mukaan. Silmukan lopussa komponentin nopeuden arvoksi asetetaan vanhan ja uuden sijainnin erotus jaettuna time step -arvolla. Silmukan jälkeen seinäjuoksemisen ehdot tarkastetaan vielä kerran, ja mikäli niitä ei täytetä, hahmo alkaa putoamaan. (delgoodie 2023.)

```
// Compute move parameters
const FVector Delta = timeTick * Velocity; // dx = v * dt
const bool bZeroDelta = Delta.IsNearlyZero();
if (bZeroDelta)
{
            remainingTime = 0.f;
}
else
{
            FHitResult Hit;
            SafeMoveUpdatedComponent(Delta, UpdatedComponent-
>GetComponentQuat(), true, Hit);
            FVector WallAttractionDelta = -WallHit.Normal *
WallAttractionForce * timeTick;
            SafeMoveUpdatedComponent(WallAttractionDelta, UpdatedComponent-
>GetComponentQuat(), true, Hit);
}
if (UpdatedComponent->GetComponentLocation() == OldLocation)
{
            remainingTime = 0.f;
            break;
}
Velocity = (UpdatedComponent->GetComponentLocation() - OldLocation) /
timeTick;
```

Listaus 37. Liikekomponenttia liikutetaan, ja sen nopeus asetetaan time step -arvon avulla (Listaus: delgoodie).

Jotta seinällä liikkumisesta pystyisi hyppäämään pois, CanAttemptJump-metodi on ylikirjoitettava siten, että se palauttaa true-arvon, mikäli hahmo juoksee seinällä (listaus 38). Ylikirjoittamalla DoJump-funktion, hypylle voidaan määrittää seinällä juostessa oma nopeusvektori pinnan normaalin mukaan. (delgoodie 2023.)

```
bool UZippyCharacterMovementComponent::CanAttemptJump() const
{
            return Super::CanAttemptJump() || IsWallRunning();
}

bool UZippyCharacterMovementComponent::DoJump(bool bReplayingMoves)
{
            bool bWasWallRunning = IsWallRunning();
            if (Super::DoJump(bReplayingMoves))
            {
                        if (bWasWallRunning)
                        {
                                    FVector Start = UpdatedComponent-
>GetComponentLocation();
                                    FVector CastDelta = UpdatedComponent-
>GetRightVector() * CapR() * 2;
                                    FVector End = Safe_bWallRunIsRight ?
Start + CastDelta : Start - CastDelta;
                                    auto Params = ZippyCharacterOwner-
>GetIgnoreCharacterParams();
                                    FHitResult WallHit;
                                    GetWorld()-
>LineTraceSingleByProfile(WallHit, Start, End, "BlockAll", Params);
                                    Velocity += WallHit.Normal *
WallJumpOffForce;
                        }
                        return true;
            }
            return false;
}
```

Listaus 38. Hyppääminen pois pinnasta seinäjuostessa mahdollistetaan ylikirjoittamalla CanAttemptJump- ja DoJump-funktiot (Listaus: delgoodie).

Mukautettuja Movement Mode -tiloja varten tulee myös ylikirjoittaa GetMaxSpeed- ja GetMaxBrakingDeceleration-metodit, joista ensimmäinen palauttaa tilan maksimivauhdin ja jälkimmäinen maksimihidastuksen. Lopuksi OnMovementModeChanged-funktiossa seinän sijainnin ilmaiseva totuusarvo asetetaan SimulatedProxy-tyyppisiä hahmoja varten (listaus 39). Näin kyseistä totuusarvoa ei tarvitse lähettää jokaiselle pelaajalle, koska movement mode -tilan muuttuminen replikoidaan kaikille muutenkin. (delgoodie 2023.)

```
if (IsWallRunning() && GetOwnerRole() == ROLE_SimulatedProxy)
{
            FVector Start = UpdatedComponent->GetComponentLocation();
            FVector End = Start + UpdatedComponent->GetRightVector() *
CapR() * 2;
            auto Params = ZippyCharacterOwner->GetIgnoreCharacterParams();
            FHitResult WallHit;
            Safe_bWallRunIsRight = GetWorld()-
>LineTraceSingleByProfile(WallHit, Start, End, "BlockAll", Params);
}
```

Listaus 39. Seinän sijainnin ilmaiseva totuusarvo asetetaan SimulatedProxy-hahmoja varten (Listaus: delgoodie).

Viimeinen huomioitava kohta olisi seinällä juoksemisen visualisointi. Tässä esimerkissä pelihahmoa ei esimerkiksi kierretä pois pinnasta päin C++-koodilla, vaan tällaiselle liikkumiselle on oma animaatio, joka aktivoidaan animation blueprint -skriptissä (delgoodie 2023). Liikemekaniikkojen animointi ei kuitenkaan kuulu tämän opinnäytetyön tavoitteisiin.

7 Pohdinta

7.1 Tulokset suhteessa tietoperustaan

Toteuttaessa verkkomoninpeliä Unreal Engine -moottorilla on tietenkin tärkeää ymmärtää asiakas-palvelin-arkkitehtuurissa hyödynnettäviä tekniikoita. Mutta ymmärtääkseen tätä mallia parhaiten, sitä on kuitenkin syytä vertailla vanhempaan vertaisverkkoarkkitehtuuriin. Näin asiakas-palvelin-mallin tarjoamien ominaisuuksien edellytykset ja vahvuudet tulevat selväksi; hyödyntämällä auktoritatiivista palvelinta, jokaisen asiakkaan pelin tilan ei tarvitse olla aina täysin synkronoitu. Näin viivettä on mahdollista eliminoida simuloimalla toimintoja heti asiakkaan puolella, ja liikkeitä voi laskea uudelleen kun epäsynkronointia tapahtuu. Kyseisistä arkkitehtuureista sekä niiden historiasta löytyykin melko helposti paljon dokumentaatiota.

Unreal Engine -pelimoottorin tarjoamalla Gameplay Ability -järjestelmällä liikemekaniikkojen toteuttaminen on melko helppoa ja intuitiivista. Tähän liittyvä virallinen dokumentaatio on myös melko selvää. Käytännön toteutuksesta tulee kuitenkin ilmi, että tämä järjestelmä ei välttämättä sovellu monimutkaisemman jatkuvan liikkeen rakentamiseen. Liikekomponentin hyödyntäminen tarjoaakin enemmän tapoja hallinnoida liikkeiden suorittamista sekä niiden korjaamista, minkä seurauksena lopputuloksesta on helpompi saada sulavampi.

Uuden toiminnallisuuden lisääminen oikeaoppisesti osaksi hahmon liikekomponenttia on kuitenkin monimutkainen prosessi, jota voi olla vaikea ymmärtää virallisenkin dokumentaation kautta. Liikkumisessa hyödynnetään monia funktiota, joista voi olla vaikeaa erottaa ne, jotka tulisi ylikirjoittaa. Liikekomponenttiin perehtymiseen kannattaa siksi käyttää paljon aikaa lukemalla lähdekoodia sekä tutustumalla yksityiskohtaisiin ulkoisiin oppaisiin.

Mekaniikkojen testaaminen kuitenkin toimii kuten Epic Gamesin tarjoamissa ohjeissa on kuvattu. Simuloidessa viivettä asiakkaan puolella ero oikea- ja vääräoppisesti rakennettujen toiminnallisuuksien ero tulee hyvin selväksi, sillä

ilman client-side prediction -tekniikkaa syötteissä esiintyy selvää viivettä. Lisäksi korjaustilanteissa liikkeen tietojen tallennuksen puute saa liikkeiden uusimisen näyttämään enemmän epätasaiselta. Emulaatioasetuksia säätäessä tulee ottaa huomioon, että ne nollautuvat kun pelimoottori sammutetaan. Kyseiset asetukset tulee siis määrittää uudelleen aina kun moottori käynnistetään.

7.2 Tulokset suhteessa asetettuihin tavoitteisiin

Kaikkia konkreettisia ohjelmointitehtäviä, tarkalleen ottaen seinäjuoksun ja -hypyn toteuttamista ei tämän opinnäytetyön puitteissa ehditty suorittamaan. Lisäksi ajan kelaamisen tilan päivittäminen käyttöliittymässä jäi jokseenkin keskeneräiseksi, eikä kelaukseen sisällytetty esimerkiksi elinpisteiden muuttamista. Tuloksia tarkastellessa tulee myös ottaa huomioon, että mekaniikkoja ei testattu tarkan ennalta määritetyn suunnitelman mukaan, ja tuloksetkin ovat kvalitatiivisia eikä kvantitatiivisia. Vaikka jatkokehitykselle on selvästi varaa, työ on silti varsin onnistunut ottaen huomioon suuremmat oppimistavoitteet.

Jo pelkkä teleportaation toteuttaminen antoi paljon tietämystä sulavien liikemekaniikkojen rakentamisesta asiakas-palvelin-arkkitehtuuria hyödyntäviin verkkopeleihin. Ajan kelaaminen toi omat haasteensa, vaikka sen lisääminen käytti monia samoja periaatteita kuin etäsiirtyminen. Lopuksi eri kehitysratkaisujen, kuten Gameplay Ability -järjestelmän hyödyntämisen ja erilaisten tietojen replikointitapojen vahvuudet ja heikkoudet tulivat hyvin selviksi. Ja vaikka seinäjuoksun ja -hypyn toteuttamiseen ei tätä työtä varten riittänyt aikaa, valmiin oppaan läpikäynti on tuottanut monia ideoita, kuten mukautettujen movement mode -tilojen hyödyntämisen sekä fysiikka- ja hyppyfunktioiden ylikirjoittamisen. Tulevaisuudessa näiden mekaniikkojen lisäämisen tulisi olla suhteellisen helppoa.

Liittyen ajan kelaamiseen, kyseisen mekaniikan voisi kenties saamaan toimimaan vielä sulavammin hyödyntämällä movement mode -tiloja sekä

fysiikkafunktioita. Lisäksi pelaajahahmon nopeusvektoria voisi muuttaa sijainnin asettamisen sijasta, kunhan hahmo ei vain ole etäsiirtynyt.

Esimerkiksi hyödyntämällä mukautettua movement mode -tilaa, pelaajaa voisi estää yrittämästä liikkua kelaamisen aikana, minkä loogisesti pitäisi vähentää prosessoinnin, käytetyn laajakaistan sekä korjaustilanteiden määrää. Tämän voisi myös saavuttaa estämällä syötteet kokonaan kelausta suorittaessa, mutta tällöin pelaaja ei voisi samalla esimerkiksi katsella ympärilleen. Liikkumisen voisi myös evätä vaikka liikekomponentin ylikirjoitettavassa AddInputVector-metodissa, kun bWantsToRewind-muuttujan arvo on tosi.

Joka tapauksessa työn ensimmäisiä jatkokehityksen kohteita olisi todennäköisesti kelauksen interpolaation poisto, kun hahmo on etäsiirtynyt. Yksi yksinkertainen tapa toteuttaa tämä olisi lisäämällä jokaisen tallennettavaan kelausdataan totuusarvo, joka kertoo, mikäli pelaaja etäsiirtyi edellisen tiedon tallennuksen jälkeen. Aikaa kelatessa tätä totuusarvoa käytettäisiin päättämään, pitäisikö interpoloida edellisen sijaintiin vai siirtyä suoraan.

Koska dataa kelausta varten säilötään nykyisessä toteutuksessa 30 kertaa sekunnissa riippumatta ruudunpäivitysnopeudesta, kelaaminen ei välttämättä täysin vastaa pelaajan aiemmin suoritettua liikettä. Tämä voi olla täysin sallittava uhraus suuren muistimäärän säästämiseksi, mutta häviöttömiä tallennustapoja kelausta varten on silti hyvä harkita.

Tietoa pelihahmon tilasta voisi esimerkiksi säilöä jokaisen ruudunpäivitysten kohdalla siten, että pelin aikaleima tallennetaan myös niiden mukana. Aikaleiman voisi saada esimerkiksi UWorld-luokan GetTimeSeconds-funktiolla. Kun uusimman ja vanhimman tallennetun liikkeen aikaleiman ero ylittää kelauksen halutun keston, vanhaa tietoa alettaisiin poistamaan uuden tilalta. Lisäksi mikäli pelihahmon tilassa ei tapahdu muutosta verrattuna edelliseen tallennukseen, uutta tietoa ei tarvitse tallentaa, koska aikaleimat ilmaisevat miten pitkään mikäkin tila on kestänyt.

Kelauksen kohdalla viimeinen jatkokehityksessä käsiteltävä asia olisi käyttöliittymän parantaminen. Toistaiseksi pelaajan näytölle on vain palkki, joka määritetyssä ajassa muuttuu tyhjästä täydeksi pelin alettua sekä kelaamisen loputtua. Tätä käyttöliittymäelementtiä voisi parantaa siten, että palkki alkaisi tyhjentyä kelauksen laukaistua. Lyra-projektissa ei kuitenkaan ole suoraan näin toimivaa elementtiä, mutta tämä on silti luotavissa yksinkertaisella käyttöliittymäohjelmoinnilla.

Lopuksi tällaisia mekaniikkoja luodessa verkkomoninpelejä varten tulee huomioida huijaamisen esto. Eräs huijaustapa olisi teleportaation tai kelauksen kohdesijainnin muuttaminen esimerkiksi ulkoisella ohjelmalla. Asiakkaan lähettämien muuttujien kelpous voidaan kuitenkin vahvistaa palvelimen puolella kun tiedot ovat vastaanotettu, mutta tämä vaatisi vielä oman validointilogiikan luomista.

Etäsiirtymisen kohdalla validoinnin pitäisi olla suhteellisen yksinkertaista; kohdesijainnin tulee ensinnäkin olla määritellyn etäisyyden sisällä pelihahmosta, tosin pienelle virhemarginaalille voi tehdä tilaa. Lisäksi pelaajan tulee mahtua etäsiirtymisen kohteeseen. Vektorin tarkastaminen ajan kelausta varten olisi kuitenkin monimutkaisempaa ainakin nykyisellä toteutuksella. Yksi vaihtoehto olisi katsoa onko lähetetty kelaamisen kohdesijainti tarpeeksi lähellä mitään palvelimen puolella tallennettua sijaintia.

Huijaamisen estämistä varten tulee vielä huomioida, että Lyra-projektissa kaikkien Gameplay Ability -taitojen NetSecurityPolicy-muuttujan arvo on ClientOrServer. Tämä tarkoittaa, että kykyjen aloittaminen ja lopettaminen voidaan vapaasti laukaista asiakkaan tai palvelimen puolella. Asiakas voisi mahdollisesti huijata aktivoimalla tai päättämällä kyvyn silloin, kun hänen ei pitäisi pystyä tekemään näin. Tämän estämiseksi NetSecurityPolicy-muuttujan arvoksi voisi asettaa ServerOnly, ServerOnlyExecution tai ServerOnlyTermination riippuen tarpeesta. Kaiken kaikkiaan huijaamisen esto on kuitenkin niin laaja aihe, että se ei mahdu tämän työn laajuuteen.

8 Lähteet

Bettner, P. 2001. 1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond. Game Developer. https://www.gamedeveloper.com/programming/1500-archers-on-a-28-8-network-programming-in-age-of-empires-and-beyond. 30.8.2023.
Carmack, J. 1996. .plan. https://fabiensanglard.net/fd_proxy/doom3/pdfs/johnc-plan_1996.pdf. 30.8.2023.
Corman, A., Douglas, S., Schachte, P., & Teague, V. 2006. A Secure Event Agreement (SEA) protocol for peer-to-peer games. IEEE. https://www.comp.nus.edu.sg/~cs4344/0607s1/sea.pdf. 30.8.2023.
delgoodie. 2023. [Unreal Engine] Wall Run | Character Movement Component In-Depth. YouTube-video. https://youtu.be/F2IOnzzOCT0?list=PLXJlkahwiwPmeABEhjwIALvxRSZkzoQpk. 30.11.2023.
Epic Games. 2018. Character Movement Component. https://web.archive.org/web/20181010012919/https://docs.unrealengine.com/en-us/Gameplay/Networking/CharacterMovementComponent#advancedtopic:addingnewmovementabilitiestocharactermovement. 30.8.2023.
Epic Games. 2023a. RPCs. https://docs.unrealengine.com/5.3/en-US/rpcs-in-unreal-engine/. 9.12.2023.
Epic Games. 2023b. Traces Overview https://docs.unrealengine.com/5.3/en-US/traces-in-unreal-engine---overview/. 9.12.2023.
Epic Games. 2023c. Networking and Multiplayer. https://docs.unrealengine.com/5.3/en-US/networking-and-multiplayer-in-unreal-engine/. 9.12.2023.
Epic Games. 2023d. Networking Overview. https://docs.unrealengine.com/5.2/en-US/networking-overview-for-unreal-engine/. 30.8.2023.
Epic Games. 2023e. Networked Movement in the Character Movement Component. https://docs.unrealengine.com/5.2/en-US/understanding-networked-movement-in-the-character-movement-component-for-unreal-engine/. 30.8.2023.
Epic Games. 2023f. AGameNetworkManager. https://docs.unrealengine.com/5.2/en-US/API/Runtime/Engine/GameFramework/AGameNetworkManager/. 6.11.2023.
Epic Games. 2023g. Gameplay Ability. https://docs.unrealengine.com/5.2/en-US/using-gameplay-abilities-in-unreal-engine/. 30.8.2023.
Epic Games. 2023h. Ability System Component And Attributes. https://docs.unrealengine.com/5.3/en-US/gameplay-ability-system-component-and-gameplay-attributes-in-unreal-engine/. 1.11.2023.
Epic Games. 2023i. Testing and Debugging Networked Games. https://docs.unrealengine.com/5.0/en-US/testing-and-debugging-networked-games-in-unreal-engine/. 30.8.2023.
Epic Games. 2023j. Using Network Emulation. https://docs.unrealengine.com/5.0/en-US/using-network-emulation-in-unreal-engine/. 30.8.2023.
Fiedler, G. 2010. What Every Programmer Needs To Know About Game Networking. Gaffer On Games.

https://www.gafferongames.com/post/what_every_programmer_needs_to_know_about_game_networking/. 30.8.2023.
Gambetta, G. 2023. Client-Server Game Architecture. Fast-Paced Multiplayer. https://www.gabrielgambetta.com/client-server-game-architecture.html. 30.8.2023.
Glazer, J. & Madhav, S. 2016. Multiplayer Game Programming. Addison-Wesley. http://ptgmedia.pearsoncmg.com/images/9780134034300/samplepages/9780134034300.pdf. 30.8.2023.
Lincroft, P. 1999. The Internet Sucks: Or, What I Learned Coding X-Wing vs. TIE Fighter. Game Developer. https://www.gamedeveloper.com/design/the-internet-sucks-or-what-i-learned-coding-x-wing-vs-tie-fighter. 30.8.2023.
Neumann, C., Prigent, N., Varvello, M. & Suh, K. Challenges in Peer-to-Peer Gaming. ACM SIGCOMM. http://ccr.sigcomm.org/online/files/p79-v37n1p-neumann.pdf. 30.8.2023.
van Waveren, J. 2006. The DOOM III Network Architecture. id Software. https://www.mrelusive.com/publications/papers/The-DOOM-III-Network-Architecture.pdf. 30.8.2023.