Zahvala

Zahvaljujem se svojoj mentorici Maji Matijašević za jako dobro mentorstvo tijekom mog preddiplomskog i diplomskog studija.

Zahvaljujem se i svojoj obitelji (otac Alen, majka Valentina, brat Tin, djed Stjepan, baka Milka, pokojni djed Branko i baka Zlata) na podršci tijekom mog cijelog školovanja.

Sadržaj

Uvod

Umrežene digitalne igre su u današnje vrijeme izrazito popularne. Interakcija između više igrača, gdje god oni geografski bili daje posebno iskustvo koje je teško postići bez umreženog aspekta igara. Problem kod umreženih igara koje koriste računalne mreže kao sredstvo komunikacije je u činjenici da računalne mreže nisu savršene. Preciznije, u računalnim mrežama su prisutni nepoželjni efekti poput mrežnog kašnjenja i gubitka paketa koje je ponekad potrebno kompenzirati između igrača kako bi se ti efekti ublažili.

Cilj ovoga rada je dizajnirati pokaznu igru u kojoj su implementirane određene tehnike mrežnog kašnjenja uz analizu uspješnosti određene metode s obzirom na mrežno kašnjenje.

Rad se sastoji od pet poglavlja u kojima se detaljnije opisuje problem, dizajn i razvoj igre, tehnike kompenzacija kašnjenja i analiza uspješnosti određene metode uz rezultate. U provom poglavlju su opisani osnovni pojmovi od koncepta igre, računalnih mreža, umreženih igara i kompenzacije kašnjenja. Drugo poglavlje opisuje dizajn igre, odnosno kako igra radi i kao izgleda. Treće poglavlje je proces analize kompenzacije kašnjenja, gdje se uvodi hipoteza, proces i sustav za mjerenje te metodologija. Četvrto poglavlje opisuje izradu igre tj. odabir tehnologija za izradu i njihovo korištenje. Peto poglavlje se svodi na mjerenja i njihovo interpretiranje (rezultate). Na kraju rada se može pronaći zaključak, popis literature, sažetak i popis skraćenica.

1. Specifikacija problema

Cilj rada je bio dizajnirati i programski implementirati pokaznu umreženu igru u kojoj se realiziraju određene tehnike kompenzacije kašnjenja. Kompenzacija kašnjenja je skup tehnika koje „ublažavaju“ nepoželjno mrežno kašnjenje. Uz samu realizaciju igre, odrađena je i detaljnija analiza određene tehnike kompenzacije kašnjenja s obzirom na različite iznose mrežnog kašnjenja.

1.1. Koncept igre

Ideja igre nazvane „Tank Fight“ je borba s tenkovima za do 4 igrača. Svaki igrač ima svoj tenk te je cilj uništiti tenkove drugih igrača. Mečevi su tipa FFA (engl. Free For All), što znači da je svaki igrač samostalan tj. nema timova. U igri je naglašen realizam, odnosno tenkovi su modelirani tako da imaju oklop, granate imaju paraboličnu putanju (kinetički projektili), gusjenice tenkova se savijaju po tlu i sl. Tenkovi također imaju specifičan model štete, nema klasičnih „health pointova“ nego su unutarnje komponente tenka (posada, motor, gusjenice, top, municija) posebno modelirani, a šeta se određuje simulacijom prilikom pogotka. Tenk se smatra mrtvim ako ima manje od dva živa člana posade, dok se ostale neljudske komponente mogu automatski popraviti tijekom igranja.

Kada određeni tenk „umre“, on se ponovo stvori u „izliječenom stanju“ tj. ponovo spreman za borbu nakon određenog vremena. Uz osnovni cilj uništavanja protivničkih tenkova, igrači su slobodni raditi što god žele.

Slične igre po opisanom konceptu su „Gunner, HEAT, PC!“ **Error! Reference source not found.** od tvrtke Radian Simulations LCC i „War Thunder“ [2] od tvrtke Gaijin Entertainment.

1.2. Računalne mreže

Izvori poglavlja: [3]

Igra je umreženog tipa, a to znači da zahtjeva mrežnu povezanost.

Računalna mreža je skup računala ili drugih uređaja koji međusobno komuniciraju kako bi dijelili informacije, resurse ili usluge. Ovisno o veličini i opsegu, računalne mreže mogu biti

lokalne (LAN – Local Area Network), šire (WAN – Wide Area Network) i globalne (Internet). U kontekstu umreženih igara, najčešće se koristi Internet, no ponekad i LAN (koji može biti i virtualni što je često preko interneta).

Gotovo svaki čvor u mreži (privatnoj ili javnoj) ima svoju jedinstvenu IP (engl. Internet Protocol) adresu. Česta je pojava da računala u različitim privatnim mrežama imaju iste IP adrese, no to je u redu zato što se između mreža često odrađuje preslikavanje mrežnih IP adresa NAT-om (engl. Network Address Translation).

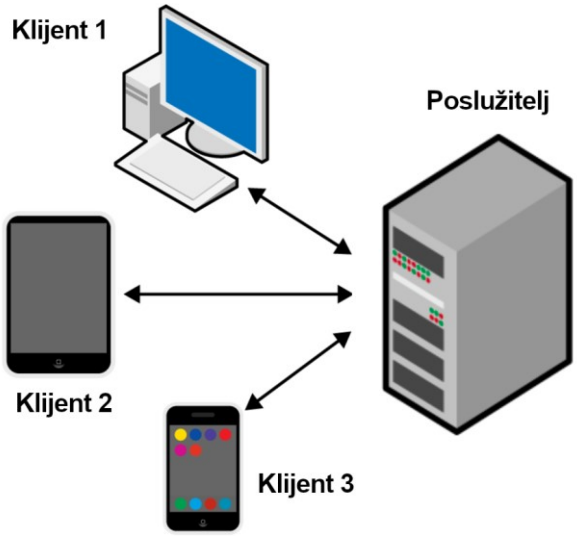
Jedan od problema u računalnim mrežama je kašnjenje, odnosno, vrijeme prijenosa paketa nije obavljeno „odmah“, nego se inače mjeri u milisekundama. Za računalne igre je preferirano što manje kašnjenje. Bitno je napomenuti da je često potrebno za svaki poslani paket dobiti i potvrdu da je paket primljen. To znači da se pod kašnjenje često također računa i vrijeme dobivanja potvrde (RTT – Round Trip Time). Paketi koji se šalju se također mogu izgubiti tijekom slanja, gubitak paketa (PL – Packet Loss) se mjeri postocima (postotak izgubljenih paketa).

Detaljnije o računalnim mrežama se može pročitati u izvorima [4] i [5].

1.2.1. Klijent Poslužitelj topologija

Kod umreženih igara, programske instance najčešće komuniciraju putem dvaju arhitektura: klijent-poslužitelj i peer-to-peer. Igra navedena u radu koristi model klijent-poslužitelj, odnosno podtip tog modela gdje jedan od igrača uzima ulogu poslužitelja (engl. Host).

U topologiji/arhitekturi klijent-poslužitelj [6], klijenti su spojeni na poslužitelj koji im služi kao točka za pristup i dijeljenje informacija (s poslužiteljem, pa tako i s drugim klijentima, slika 1.1). Alternativa za arhitekturu klijent-poslužitelj je arhitektura peer-to-peer, zainteresirani čitatelj o njoj može više pročitati u izvoru [7].



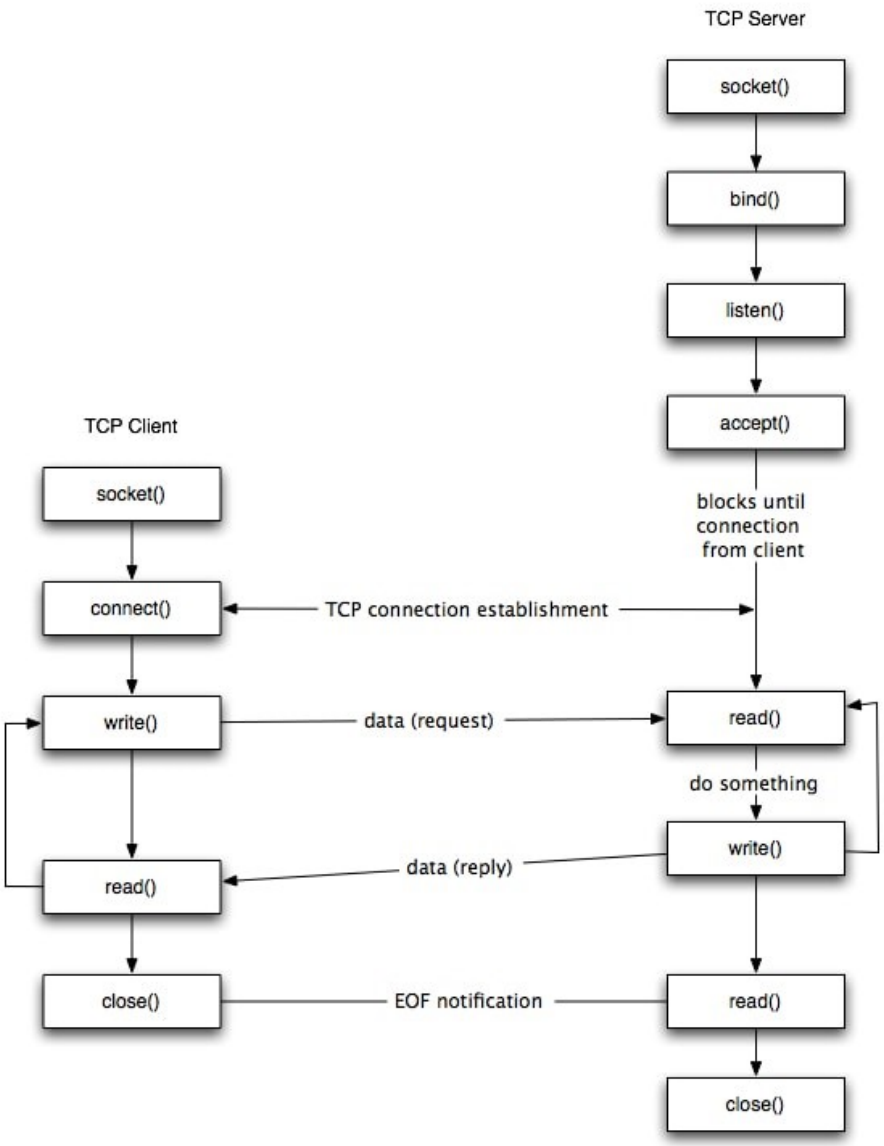
Slika 1.1 Topologija klijent poslužitelj [8]

Prednost korištenja takve arhitekture (u igrama, ali i općenito) je u tome što poslužitelj ima autoritet nad podacima i događajima, čime se znatno otežava varanje. Jedan od nedostataka ove arhitekture je cijena posjedovanja i održavanja poslužitelja – nije besplatno.

1.2.2. Programsko sučelje

Za komunikaciju između mrežnih aplikacija na programskoj razini, često se koristi Socket API (engl. Application Programming Interface), korištenjem njega se uspostavlja veza između dva čvora (programa/aplikacija) u mreži, što mogu biti npr. klijent i poslužitelj. Socket apstrahira mrežu između klijenta i poslužitelja u smislu da, osim IP adrese, porta i potrebnog komunikacijskog protokola (UDP (engl. User Datagram Protocol) ili TCP (engl. Transmission Control Protocol)), nije potrebno znati ništa drugo o mreži. Čitanje ili slanje podataka preko socketa je slično čitanju i pisanju u/iz datoteke. Gotovo svaka mrežna aplikacija koristi sockete za komunikaciju preko mreže.

Primjer uspostave i korištenja komunikacije uz pomoć socketa prikazan je na slici 1.2.:



Slika 1.2 Komunikacija preko socketa [9]

Iako se socketi gotovo uvijek koriste u mrežnoj komunikaciji, dobar dio programskih biblioteka apstrahira njihovo korištenje uz pomoć posebnih funkcija (biblioteke visoke razine) u smislu da programer ne mora ni razmišljati o tome što se događa na socketu, a to olakšava razvoj aplikacija. Više o socketima se može pročitati u knjizi L. Kalita „Socket Programming“ [10].

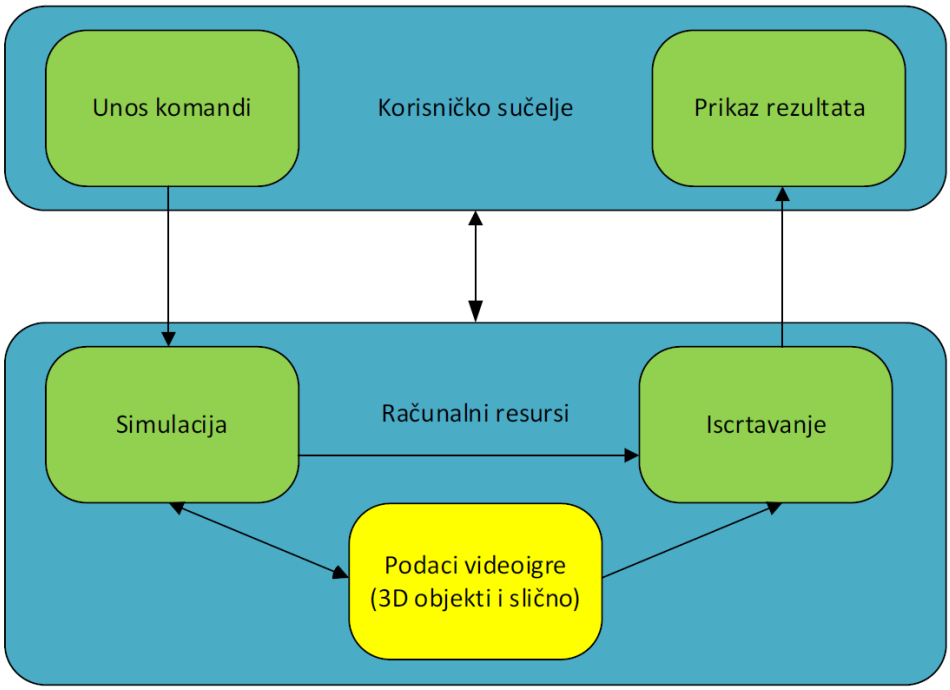
1.3. Umrežene igre

Umrežene igre omogućavaju da igrači na različitim računalima i na raznim dijelovima svijeta mogu prisustvovati u istom meču te imati međusobnu interakciju. Problem je u tome što u računalnoj mreži uvijek postoji, između ostalog, određeno kašnjenje (često prikazano u milisekundama) i određeni gubitak paketa (prikazan u postocima) koji mogu narušiti iskustvo igraća. Istraživanje od L. Shengei i M. Claypool [11] je pokazalo da iskustvo igraća (u FPS žanru koji je podosta sličan igri u ovome radu) linearno pada kako kašnjenje raste počevši od 25 ms. U određenim radovima [12][13] je pokazano da kašnjenje od 50 ms (ili više) mogu imati utjecaj na rad s mišem od računala [14]. Za gubitak paketa (a i kvalitetu veze općenito) je pokazano da igrači često odustanu od igre/meča jer je njihovo iskustvo narušeno [18]. Postotak gubitka paketa kod kojeg igrači imaju narušeno iskustvo varira o igri/tehnologiji za umrežavanje. Za igru „Halo“, igračima je iskustvo narušeno s gubitkom paketa od 4% (ili više), dok za igru „Quake III“ (donekle) prihvatljiv gubitak paketa iznosi do 35% [15]. Ovisno o lokaciji poslužitelja i vremenu igranja igara (tj. koliko igrača je trenutno spojeno na poslužitelj), gubitak paketa može varirati od 0% (kada nema mnogo igrača) do 25% (kada je broj igraća pri vrhu) [16].

1.4. Umrežavanje digitalnih igara

Primarni izvor poglavlja: [17]

Umrežena digitalna igra je igra koja se igra putem računalne mreže. Gotovo svaka igra ima dvije vrste informacija: promjenjive i nepromjenjive. Promjenjive informacije mogu biti pozicije i rotacije objekata, health bodovi likova od igrača i sl. Nepromjenjive informacije su u osnovi konstante tj. ne mijenjaju se tijekom igranja, to mogu biti pozicije i rotacije stacionarnih (fiksnih) objekata, izgledi 3D objekata i sl. Generalni cilj kod umrežavanja videoigara je, na prikladan način, sinkronizirati potrebne promjenjive informacije uz pomoć mreže. Skup promjenjivih informacija zove se dinamičko stanje. Gotovo svaka digitalna igra koristi sustav prikazivanja baziran na otkucajima (tj. prikazivanju određenog broja sličica u sekundi, slika 1.3). Korisnik ima svoje sučelje koje se sastoji od uređaja za unos (npr. tipkovnica, miš, kontroler, zaslon na dodir) i uređaja za prikaz (npr. monitor, TV, zvučnik, VR (engl. Virtual Reality) headset).

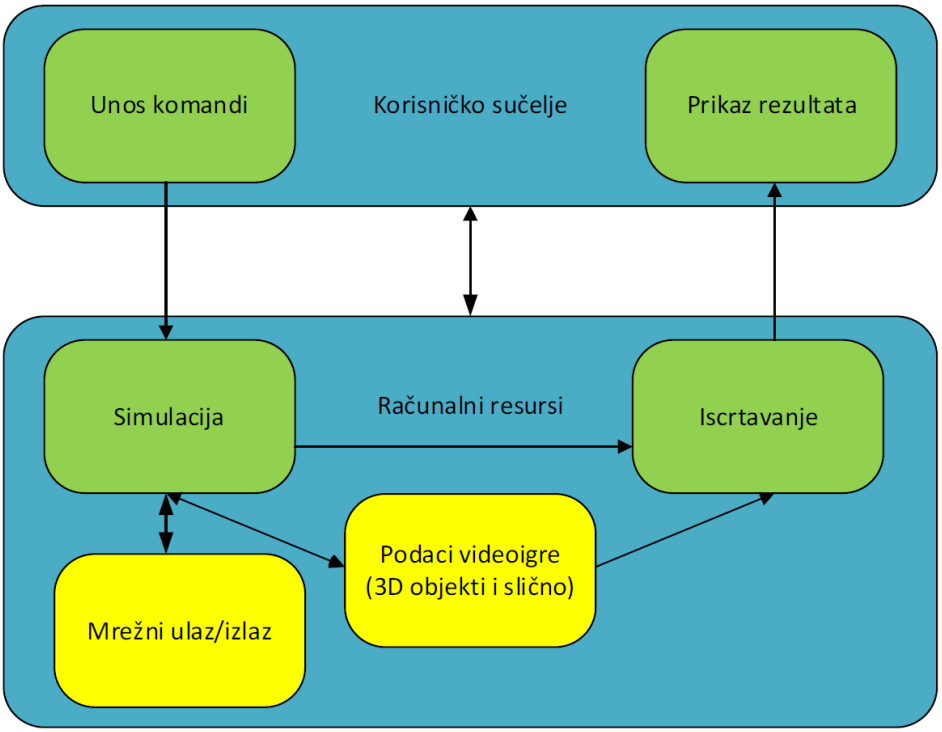


Slika 1.3 Klasični sustav rada (neumreženih) digitalnih igara [17]

1.4.1. Umrežena simulacija

Svaki igrač (ili korisnik) na svom računalu ima pokrenutu svoju instancu igre. Radi mrežnog kašnjenja, gotovo je nemoguće je da svatko vidi (ili ima) isto stanje u istom vremenskom trenutku. Jedna od bitnih karakteristika kod umreženih igara je konzistentnost simulacije koja određuje mjeru usklađenosti dinamičkih stanja u video igri (umreženoj simulaciji). Općenito, postoje dvije vrste konzistentnosti: jaka i slaba. Kod jake konzistentnosti, stanje umrežene simulacije je isto za svakog korisnika u svakom trenutku. Problem je u tome što informacija mora stići do svih korisnika u mreži prije nego što može doći do promjene u simulaciji. Iz toga proizlazi činjenica da je brzina simulacije (tj. promjena) uvjetovana kvalitetom veze najsporijeg korisnika što može imati loš utjecaj na iskustvo ostalih. Slaba konzistentnost nema taj problem, no kod nje nema garancije da će svi imati isto stanje igre koje je onda potrebno naknadno sinkronizirati. Igra u radu ima slabu konzistentnost tj. kvaliteta simulacije ovisi o kvaliteti veze za određenog igraća.

Neovisno o tipu konzistentnosti, dinamičko stanje igre se osvježava (tj. obnavlja) u određenim vremenskim trenucima odnosno mrežnim otkucajima (engl. tick). Broj otkucaja u sekundi je također fiksan, a određen je mjerom zvanom brzina otkucaja (engl. tick-rate). Mrežni otkucaji su često neovisni o običnim otkucajima u igri koji se izvršavaju prije crtanja slike na zaslon. Takav sustav je prikazan na slici 1.4.:



Slika 1.4 Sustav rada umreženih digitalnih igara [17]

Za slabu konzistentnost potrebno je imati i sustav autoriteta nad (mrežnim) objektima. Mrežni objekti su objekti „bitni“ za mrežnu simulaciju tj. njihove promjenjive informacije se moraju sinkronizirati. Sustav autoriteta je potreban zato što dva igrača mogu u istom vremenskom trenutku u stvarnom svijetu odraditi različite promjene nad nekim istim objektom i iz toga nastaje „kolizija“ ili konflikt. Pitanje je koja promjena od kojeg igrača je valjana? Bez sustava autoriteta nije moguće dati dobar i funkcionalan odgovor. Ideja sustava autoriteta je da svaki mrežni objekt ima svog vlasnika, a taj vlasnik ima autoritet nad tim objektom. S autoritetom nad određenim objektom, moguće je raditi promjene na tom objektu. Može postojati više vrsta autoriteta tj. autoritet nad stanjem (engl. State Authority) i autoritet nad ulazom (engl. Input Authority). Autoritet nad stanjem se često prepušta samo poslužitelju, dok svaki igrač dobije svoj objekt nad kojim ima autoritet za ulaz. Dodatna prednost sustava autoriteta je prevencija varanja u igrama.

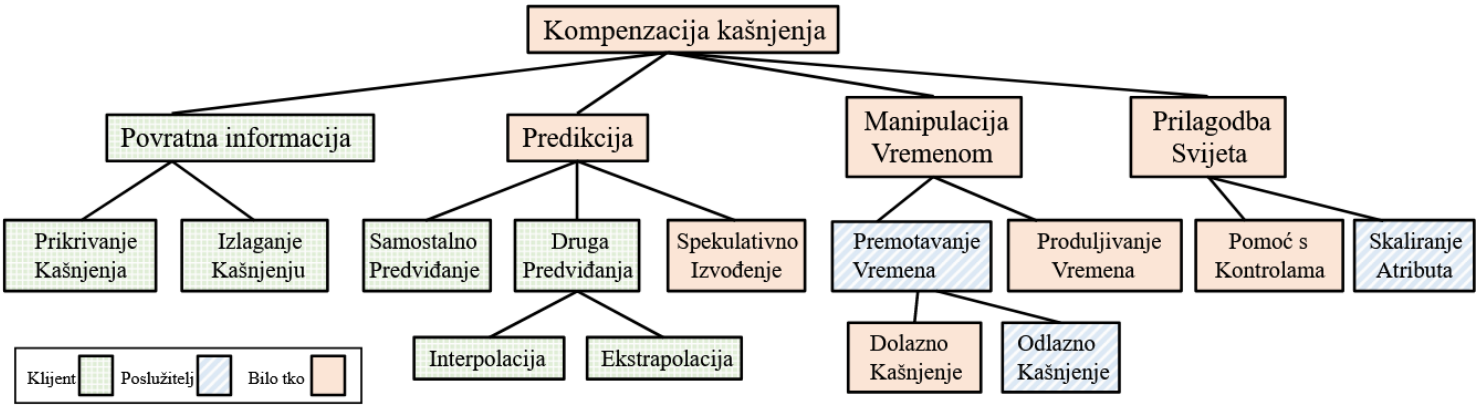
Za obnavljanje stanja kod slabe konzistentnosti, koristi se interpolacija i/ili ekstrapolacija (predikcija). Spomenute dvije metode se obavljaju iz snimaka (engl. Snapshots) stanja igre.

1.4.2. Kompenzacija kašnjenja

Primarni izvor poglavlja: [14].

Zbog spomenutih mrežnih problema (kašnjenje), u umreženim igrama se često događa situacija gdje, radi kašnjenja, igrači „ne vide“ isto stanje igre. Kompenzacija kašnjenja je skup tehnika uz pomoć kojih se ublažava učinak kašnjenja na iskustvo igranja u smislu da igra bude generalno pravednija i u određenim slučajevima „glađa“, inače bi igrači s boljim mrežnim uvjetima imali prednost nad igračima s lošijim mrežnim uvjetima. Kao što je prije rečeno, istraživanje od L. Shengei i M. Claypool [11] je pokazalo da iskustvo igraća linearno pada kako kašnjenje raste počevši od 25 ms, ali je i pokazalo da se uporabom prikladnih metoda kompenzacije kašnjenja to može poboljšati. Isto vrijedi i za odustajanje igrača kod gubitka paketa (i općenite kvalitete veze) , ali je uz to i pokazano da se primjenom prikladnih tehnika kompenzacije kašnjenja vjerojatnost odustajanja smanjuje [18].

Postoje razne vrste i načini kompenzacije kašnjenja prikazanih slikom 1.5 koja je preuzeta iz izvora [14].



Slika 1.5 Vrste kompenzacije kašnjenja – prevedeno od izvora [14]

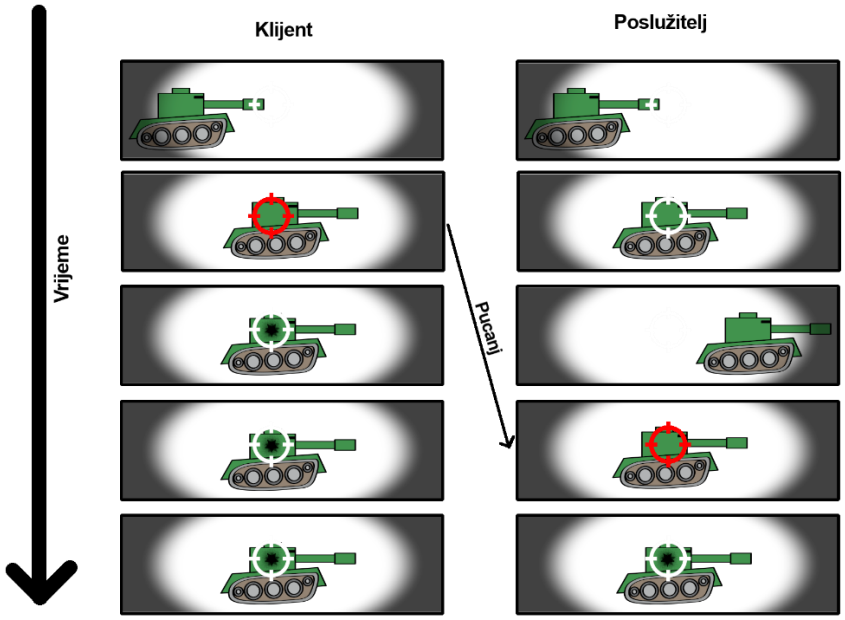
U radu "A Survey and Taxonomy of Latency Compensation Techniques for Network Computer Games." [14], autori su analizirali 85 drugih radova o kompenzaciji kašnjenja. U tom radu je obrađeno 11 tipova kompenzacija kašnjenja te ih podijelili u 4 skupine. Ovisno o tipu, kompenzacija kašnjenja se može izvoditi na klijentu, poslužitelju ili obostrano. Metode tipa povratna informacija ne mijenjaju stanje svijeta, već samo daju grafički ili zvučni efekt da se određen događaj odradio. Metode predikcije procjenjuju stanje igre na klijentu bez posjedovanja informacija o stanju svijeta od poslužitelja. Manipulacija vremenom uključuje izmjenjivanje vremena svijeta u igri za izračun stanja svijeta i/ili rješavanje radnji igrača. Prilagodba svijeta mijenja stanje igre kako bi se smanjila težina igre igraća s visokim kašnjenjem na razinu igraća s nižim kašnjenjem.

Korištene tehnike kompenzacije kašnjenja u ovom radu su premotavanje vremena (engl. Time Wrap, ova tehnika je često poznata kao i sinonim za kompenzaciju kašnjenja) i

samostalno predviđanje (engl. Self-Prediction, korišteno za simulaciju štete i kod kretanja tenkova) s većim naglaskom na premotavanje vremena.

1.4.2.1 Premotavanje vremena

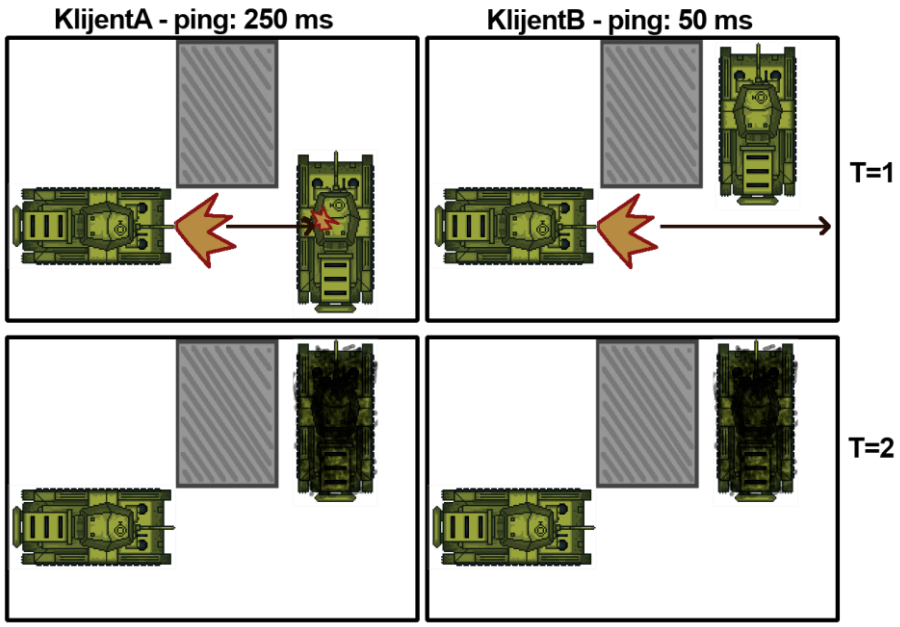
Ideja kod premotavanja vremena je da poslužitelj pohranjuje prijašnja stanja svijeta u igri na određeno vrijeme. Cilj je da se uz određeno kašnjenje klijenta, za autorizaciju koriste starija stanja svijeta sukladno njegovom kašnjenju. Potreba za premotavanjem vremena je prisutna zato što klijenti kod sebe često imaju drukčija stanja svijeta i međusobno i s poslužiteljem. Premotavanje vremena je uglavnom bitno kod igara koje se izvode u realnom vremenu (engl. Realtime) gdje se igrači mogu „kretati i/ili pucati“ što je slučaj za igru u radu. Za igre s potezima gdje likovi/objekti stoje, ili pak igre u realnom vremenu gdje nema kretanja i potrebe za gađanjem objekata koji se kreću, premotavanje vremena nije potrebno. Konkretnije, u igri u radu se pamte prijašnje pozicije i rotacije posebnih kompenziranih „sudarača“ (engl. collider) te se koriste na opisan način. Ti sudarači se koriste za detekciju pogotka tenkova tj. drugih igrača. Prikaz rada premotavanja vremena je demonstriran na slici 1.6:



Slika 1.6 Rad premotavanja vremena, sličan koncept je prikazan u radu [14]

Klijent i poslužitelj vide različite stvari u istom vremenu, bez kompenzacije kašnjenja (tj. premotavanja vremena) klijent ne bih pogodio metu jer je prema „pogledu“ tj. stanju svijeta servera meta „prešla nišan“. Uz premotavanje vremena, klijent pogađa metu i obojica (server i klijent) tako registriraju stanje.

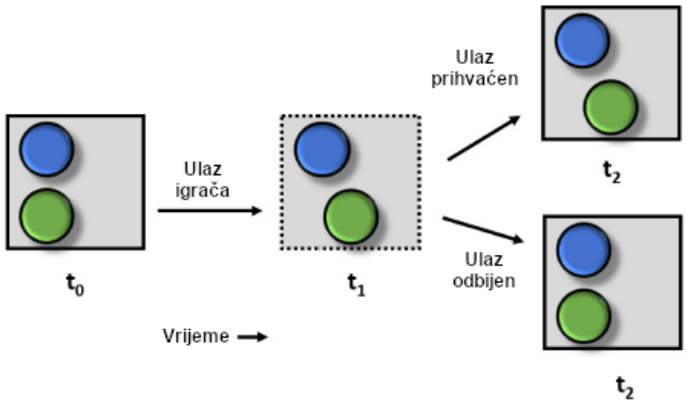
Premotavanje vremena ima jedan bitan nedostatak: ako je kašnjenje određenog klijenta iznimno veliko (npr.: 250 ms) i ako se meta giba većim brzinama (npr. 10 m/s), moguće je da meta „umre iza zida“ (gdje je zid naravno blokator bilo kakve štete). Za igrača koji ima veliko kašnjenje, premotavanje vremena mu daje „prednost“, dok igrač s malim kašnjenjem (npr. 50 ms) je „zakinut“. Taj nedostatak je prikazan na slici 1.7:



Slika 1.7 „Smrt iza zida“, efekt kod premotavanja vremena

1.4.2.2 Samostalno predviđanje

Ideja kod samostalnog predviđanja je da se ulaz za pojedinog klijenta (igrača) registrira bez „dozvole“ poslužitelja. Ako je ulaz (i stanje igre) prihvaćen od strane poslužitelja onda je metoda odradila „pogodak“/uspjeh. U slučaju da ulaz nije prihvaćen, stanje igre se vraća na prijašnje. Rad samostalnog predviđanja je prikazan slikom 1.8:



Slika 1.8 Primjer samostalnog predviđanja [14]

2. Dizajn igre

Kao što je i prije spomenuto, naziv umrežene igre razvijene u okviru ovog diplomskog rada je „Tank Fight“, a tema igra borba s tenkovima. Igra omogućuje najviše 4 igrača istovremeno (u istoj borbi).

2.1. Modeliranje tenka

Tenk u igri se sastoji od više vanjskih komponenti:

- Gusjenice
- Kotači
- Šasija
- Kupola
- Top
- Mitraljez na kupoli

Od njih su gusjenice i top također dio „unutarnjeg“ modela uz pomoć kojeg se računa simulacija štete.

Unutarnje komponente tenka (slika 2.1) su:

- Članovi posade (vozač, topnik, punjač i zapovjednik) – vitalne komponente – prikazane u zelenoj boji
- Municija – pogotkom dijelova municije postoji šansa da se cijeli tenk uništi (unutarnja detonacija municije) – prikazana ljubičastom bojom
- Zatvarač topa – prikazan narančastom bojom
- Top – prikazan crvenom bojom
- Motor – prikazan plavom bojom
- Gusjenice – prikazane plavom bojom
- Oklop – nije prikazan, no realiziran je kao skup 3D modela
- Interni prostori – nije prikazan, koristi se kod simulacije šteta za prepoznavanje kraja oklopa

Slika 2.1 Unutarnje komponente tenka

Tenk također ima oklop koji je posebno modeliran, određeni dijelovi oklopa su jači (ili „deblji“) od drugih. Prednji oklop je generalno jači od oklopa sa strane (obje), a oklop sa strane je jači od stražnjeg oklopa. Oklop je modeliran tako da je svaki njegov dio zaseban 3D model modeliran u programu za 3D modeliranje, to znači da je za modeliranje oklopa potrebno praktički ponovo modelirati tenk, no ovaj puta bez potrebe za teksturama i slično. U unutrašnjosti tenka i između razmaknutih oklopa modelirani su oblici (3D modeli) praznina (interni prostori) između njih što je bitno kod simulacije štete.

Najjači dijelovi oklopa su prednji oklop kupole (neprobojan) i oklop donjeg panela šasije (moguće ga je probiti, ali je tada nanesena šteta uglavnom slaba).

Kod simulacije štete, potrebno je probiti oklop te se uz pomoć ostatka granate i dodatnih (ponekad generiranih) gelera radi šteta nad unutarnjim komponentama tako da pogotkom komponente gelerom, komponenta gubi svoje bodove života (engl. health points).

Od posade tenka, najbitniji su vozač i topnik te (opcionalno) nakon njih punjač. Ako su vozač i/ili topnik mrtvi, a druga dva člana živi, drugi članovi će ih zamijeniti nakon određenog vremena. Isto vrijedi i za punjača, no njega samo zapovjednik može zamijeniti.

Fizičke i mehaničke komponente tenka se u slučaju oštećenja same automatski poprave nakon određenog vremena, dok to ne vrijedi za članove posade tenka, oni se samo mogu zamjenjivati.

2.2. Projektili

Tenk u igri iz svog topa može ispaljivati projektile. Nakon ispaljivanja projektila, objekt projektila se instancira i putuje svijetom po paraboličnoj putanji (konstanta brzina unaprijed (od orijentacije topa) uz silu gravitacije, bez otpora zraka). Trajektorija projektila je zapravo sastavljena od linija, gdje je svaka od tih linija posebna zraka za algoritam bacanja zraka. O algoritmu bacanja zraka se (za detalje) može pročitati u knjizi „An introduction to ray tracing“ [19]. Svaka zraka je kompenzirana tj. detekcija je li nešto pogođeno uzima u obzir prijašnja stanja svijeta (kompenzacija kašnjenja premotavanjem vremena). Zrake se bacaju s vremenom života projektila, odnosno kako projektil putuje, tako se zrake bacaju po njegovoj trajektoriji. Tijekom svog života, projektil može:

- Nestati nakon određenog vremena ako ništa nije pogođeno
- Pogoditi statične objekte (npr. teren)
- Pogoditi drugi tenk, tada se izvršava simulacija štete s obzirom na mjesto pogotka

Tokom pogodaka, projektil nestane tj. više ne putuje po svijetu niti se daljnja putanja provjerava.

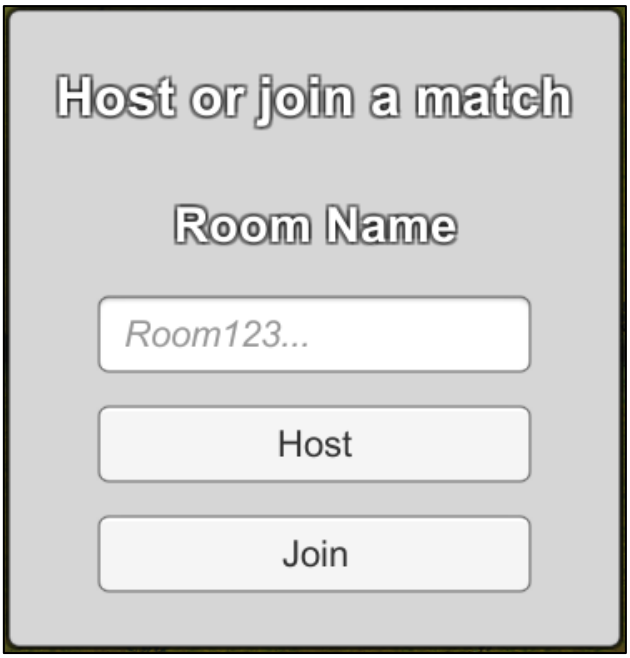
2.3. Korisničko sučelje

Otvaranjem igre prikazuje se glavni meni (engl. Main Menu, slika 2.2) u kojem igrač može odabrati sljedeće opcije:

- „Start Game“ – započinje igru
- „Tests“ – simulator štete
- „Test Polygon“ – poligon za testiranje tj. mjerenje i analizu kompenzacije kašnjenja
- „Exit“ – izlaz iz igre
- „Attributions“ – prikazuje atribucije u igri

Slika 2.2 Glavni meni u igri

Razine s gumbova „Start Game“ i „Test Poligon“ imaju identičnu funkcionalnost. Otvaranjem tih scena se prvo prikazuje prozor „Host or join a match“ (slika 2.3) uz pomoć kojega je moguće hostati vlastitu ili pridružiti se postojećoj sobi u igri klikom na prikladni gumb. Ime sobe je potrebno unijeti u prikladnom polju za unos.



Slika 2.3 Prozor za pridruživanje ili kreiranje sobe kao host

2.4. Kontrole za tenkove

Kontrole za tenk su:

- Tipka W za kretanje unaprijed ili kočenje kod kretanja unazad
- Tipka S za kočenje tijekom kretanja unaprijed ili kretanje unazad
- Tipka A za rotaciju lijevo tijekom stajanja ili skretanje tijekom kretanja
- Tipka D za rotaciju desno tijekom stajanja ili skretanje tijekom kretanja
- Pomicanje miša je rotiranje kamere igrača oko tenka pa tako i rotacija kupole prema orijentaciji kamere
- Tipka C (uz držanje) je za blokiranje rotacije kupole tijekom rotacije kamere te također omogućava prikaz pokazivača miša
- Tipka J (uz držanje) je suicid tenka tj. samouništenje
- Tipka lijevi Shift je toggle za „snajpersko ciljanje“, desna tipka miša tijekom takvog ciljanja još dodatno zumira ili od-zumira pogled/optiku
- Rotiranje kotačića miša približava ili udaljuje kameru od tenka igrača
- Tipka E omogućava „Constant Forward“ tj. tenk će se ponašati kao da je pritisnuta tipka W što je korisno kod testiranja, ova opcija se također može omogućiti na pritisak pripadnog gumba u donjem desnom dijelu prozora igre
- Tipka O za prikaz unutarnjih komponenti tenka (slika 2.6)

- Gumb Exit na donjem desnom dijelu ekrana služi za izlaz iz igre (sobe), nazad na glavni meni
- Lijevi klik miša ispaljuje projektil

2.5. Sobe za igranje

Kada igrač uđe u sobu, njegov tenk se stvori i odmah je moguće upravljati tenkom. Svaki igrač ima svoju kameru koja služi za određivanje prikaza scene, kamera se uvijek automatski pozicionira „iza“ igračevog tenka (pogled iz trećeg lica). U svakoj sobi za igranje postoje točke na kojima se tenkovi igrača instanciraju (tijekom pridruživanja igri ili ponovnog stvaranja) – točke instanciranja. Tijekom pridruživanja igri točke instanciranja se biraju sekvencijalno (ovisno o poretku pridruživanja), dok se tijekom ponovnog stvaranja ta točka bira na temelju najveće udaljenosti od svih ostalih tenkova u igri. Slika 2.4 prikazuje izgled sobe i igre tijekom igranja.

Slika 2.4 Prikaz igre tijekom igranja

2.6. HUD

Na lijevoj strani prozora u igri (slika 2.4) su prisutne ikone stanja tenka, opisi tih ikona su u nastavku (poredanih po redu čitanja po dizajnu iz prozora igre) te njihova imena i uloge su sljedeće:

- Ikona tenkovske granate - punjenje tenka municijom – kad tenk ispali granatu, potrebno je pričekati određeno vrijeme, progres punjenja tenka je takav da se ikona puni bijelom bojom
- Ikona lubanje – proces suicida (tijekom držanja tipke J) – ikona se puni crvenom bojom što označava progres prema suicidu tenka
- Ikona vozača – prikazuje je li vozač tenka živ, ako vozač tenka nije živ, tenk se ne može kretati i njegova ikona je crvena
- Ikona topnika – prikazuje je li topnik tena (engl. gunner) živ, ako nije, onda nije moguće upravljati kupolom tenka, pucati iz tenka i njegova ikona je crvena
- Ikona punjača, prikazuje je li punjač tenka (engl. loader) živ, ako nije, onda je punjenje nove granate u tenk sporije i njegova ikona je crvena
- Ikona zapovjednika – prikazuje je li zapovjednik (engl. commander) tenka živ, ako nije, onda je njegova ikona crvena
- Ikona motora – prikazuje je li motor ispravan, ako nije, tenk se ne može kretati i njegova ikona je crvena
- Ikona topa – prikazuje je li top ispravan, ako nije, iz tenka se ne može pucati i njegova ikona je crvena
- Ikona zatvarača topa – prikazuje je li zatvarač topa ispravan, ako nije, iz tenka se ne može pucati, nije moguće puniti granate u top i njegova ikona je crvena
- Ikona gusjenica – prikazuje je li su gusjenice tenka ispravne, ako je jedna od gusjenica neispravna ova ikona je crvena i oštećena gusjenica nije funkcionalna što ima utjecaj na mogućnosti kretanja tenka

Osim ikona stanja, dodatan UI/HUD u igri je prikaz brzine tenka (u km/h), prikaz broja prikazanih sličica u sekundi (FPS), opcija „Constand Forward“, stanje mrežne povezanosti, stanje pinga, gumb za izlaz iz sobe i nišan tenka (crni krug oko bijele kružnice) koji pokazuje mjesto koje će ispaljena granate tenka pogoditi. Nišan tenka postaje sve veći što je mjesto pogotka bliže tenku koji puca. Progres popravka fizičkih i mehaničkih komponenti tenka je također prikazan izbjeljivanjem njihovih ikona.

2.7. Simulator štete

Simulator štete (slike 2.5 i 2.6) je alat za upoznavanje sa simulacijom štete u igri. Jedine kontrole u tom prozoru su:

- Pomicanje miša radi rotiranje kamere oko tenka
- Tipka C (pritisnuta) zaustavlja rotiranje kamere i omogućava prikaz pokazivača miša
- Klikom na točku na tenku se prikazuje simulacija štete za pucanje granate iz pozicije kamere prema toj točki
- Klizač „Simulation Time“ određuje brzinu animacije od simulacije štete
- Gumb „Main Menu“ je za povratak u glavni meni igre

Slika 2.5 Simulator štete u igri

Slika 2.6 Prikaz simulacije štete u igri (uz prikaz unutarnjih komponenti tenka)

2.8. Simulacija štete

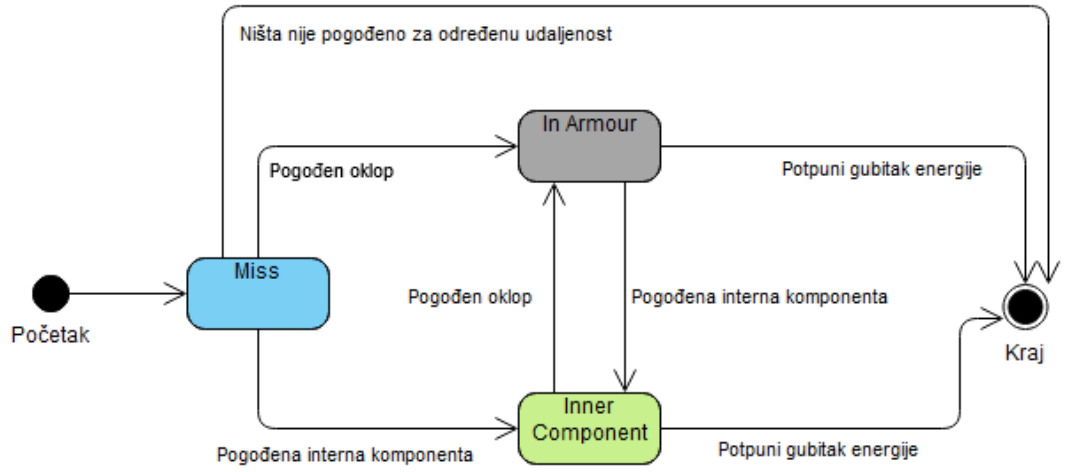
Za simulaciju štete potrebno je pogoditi tenk granatom. Pozicija i orijentacija granate tijekom pogotka je izrazito bitna (i potrebna informacija). Uz poznatu poziciju pogotka (i prikladnu orijentaciju granate), potrebno je realizirati sljedeće funkcije (pseudokod):

```
1.   DoubleRaycastAll(Vector position, Vector direction):
2.               origin1 = position
3.               origin2 = position + direction
4.               direction2 = -direction
5.               forwardHits = RaycastAll(origin1, direction)
6.               backwardHits = RaycastAll(origin2, direction2)
7.               allHits = ConcatArrays(forwardHits, backwardHits)
8.               allHits.SortBy(Distances from origin1)
9.               return allHits
```

Funkcija DoubleRaycastAll vraća sve pogođene točke gađanjem dviju zraka. Točke su sortirane po udaljenosti od izvora gađanja od najbliže do najdalje. Svrha ove funkcije je da kasnije pomogne odrediti kolika količina oklopa je „za probiti“ i je li neka komponenta pogođena nakon proboja oklopa. Funkcija se bazira na RaycastAll funkciji koja vraća sve pogođene točke koje su na putu bacanja zrake, problem RaycastAll funkcije je to što ona

(često) ne pogađa sve toče na svom putu, pogotovo ako je površina 3D modela tj. njena normala istog smjera (skalarni produkt normale i smjera zrake je veći od nula), onda se pogodak na tu površinu ne registrira. S tom činjenicom se na funkciju DoubleRaycastAll može gledati kao na proširenje standardne RaycastAll funkcije. Objekti koji se gađaju funkcijom su samo unutarnje komponente tenka.

Sljedeća potrebna funkcija je ShrapnelRaycast, ona u sebi koristi spomenutu DoubleRaycastAll funkciju kojom pregledava putanju gelera (engl. Shrapnel). Sama granata tenka je modelirana kao geler u simulaciji. Putanja jednog gelera je „simulirana“ tj. „aproksimirana“ ravnom linijom. Moguće je i da se geler reflektira nad oklopom, no to se događa samo u slučajevima kada je kut pogotka oklopa jako velik. Svaki geler ima svoju energiju uz pomoć koje može (ili ne može, ako nema dovoljno energije) probiti oklop i/ili oštetiti unutarnju komponentu tenka. Automat stanja gelera se može prikazati sljedećim grafom (slika 2.7):



Slika 2.7 Automat stanja za pojedini geler

Pseudokod je sljedeći:

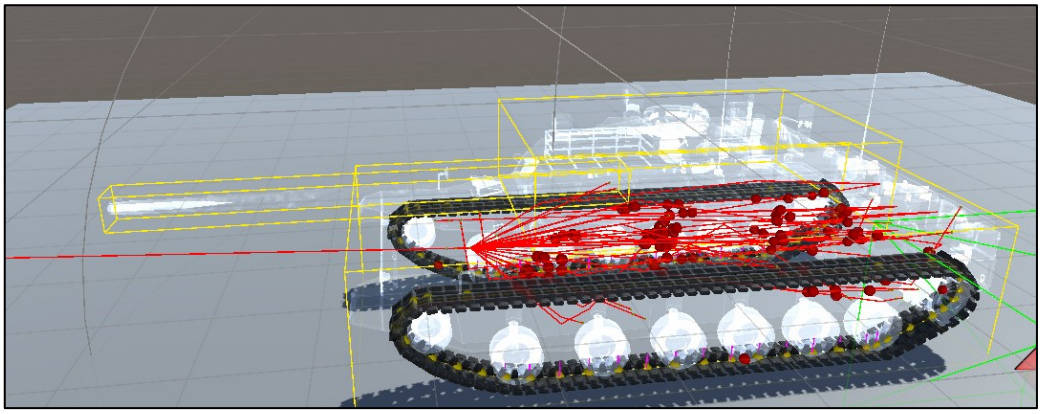
```
1.   ShrapnelRaycast(Vector position, Vector direction, float energy):
2.       hits = DoubleRaycastAll(position, direction)
3.       if hits.length < 1:
4.           return
5.       state = "MISS"
6.       ArmourBegin = ArmourNormal = ArmourEnd = Vector(NaN) // Vector full on
     NaN values
7.       for i=0, i < hits.length, i++:  //iterate through every point we hit
8.           hitObject = hits[i].object
9.           switch hitObject.type:
10.              case "INNER_AREA":
11.                  switch state:
```

```
12.                     case "MISS":
13.                         state = "IN_INNER"
14.                     case "IN_ARMOUR":
15.                         EndArmourPiercing(hits[i].point)
16.                         if energy <= 0
17.                             return  //absorbed shrapnel
18.                         state = "IN_INNER"
19.                         subShrapnelCount = energy /
    ENERGY_PER_SHRAPNEL_COST
20.                         CreateNewShrapnels(hits[i].point, direction,
    subShrapnelCount, energy / DISPERSION_COST, state)
21.                     case "IN_INNER":
22.                         //nothing, just continue moving
23.             case "ARMOUR":
24.                 switch state:
25.                     case "MISS":
26.                         HandleArmourHit(hits[i].point, direction,
    hits[i].normal)
27.                     case "IN_ARMOUR":
28.                         ArmourEnd = hits[i].point
29.                     case "IN_INNER":
30.                         HHandleArmourHit(hits[i].point, direction,
    hits[i].normal)
31.             case "INNER_COMPONENT":
32.                 switch state:
33.                     case "MISS":
34.                         DamageComponent(hitObject)
35.                     case "IN_ARMOUR":
36.                         EndArmourPiercing(hits[i].point)
37.                         if energy <= 0
38.                             return  //absorbed shrapnel
39.                         state = "IN_INNER"
40.                         DamageComponent(hitObject)
41.                     case "IN_INNER":
42.                         DamageComponent(hitObject)
43.             if energy <= 0
44.                 return  //absorbed shrapnel
45.
46.     EndArmourPiercing(Vector point):
47.         if ArmourEnd has NaN
48.             ArmourEnd = point
49.         piercedThickness = Vector.Distance(ArmourBegin, ArmourEnd)
50.         angle = Vector.angle(direction, ArmourNormal)
51.         // reduce energy after piercing
52.         energy = CalculateEnergyLossWhenPiercing(energy, piercedThickness,
    angle)
53.         ArmourBegin = ArmourNormal = ArmourEnd = Vector(NaN)
54.
55.     HandleArmourHit(Vector point, Vector direction, Vector normal):
56.         angle = Vector.angle(direction, normal)
57.         if angle is very high:  //probability map function, greater angle
    -> greater bounce off probability
58.             reflection = Vector.reflect(direction, normal)
59.             // reflect current shrapnel by creating a new one, current one
    stops
60.             ShrapnelRaycast(point, reflection, energy / 2)
61.         else:
62.             state = "IN_ARMOUR"
63.             ArmourBegin = point
64.                         ArmourNormal = normal
65.
```

```
66.                 CreateNewShrapnels(Vector point, Vector direction, int
                count, float dispersion, string state):
67.                     if count <= 0:
68.                         return
69.                     //very low energy shrapnels are ignored
70.                     if energy < SUB_SHRAPNEL_ENERGY_COST:
71.                         return
72.                     for i = 0, i < count, i++:
73.                         newDirection = GetRandomDirectionAround(direction,
                dispersion)
74.                         ShrapnelRaycast(point, newDirection, energy)
```

Pseudokod koristi i određene pomoćne lokalne metode i konstante čija je uloga ograničiti beskonačno kreiranje novih gelera te progresivno smanjivanje energije u sustavu.

Drugim riječima, simulacija štete je samo rekurzivno nizanje bacanih zraka koje se ili apsorbiraju u oklopu ili probiju sve i onda nestanu ili stanu „negdje po putu“ ili se odbiju o oklop. Vizualizacija tih zraka je prikazana slikom 2.8:



Slika 2.8 Vizualizacija zraka kod simulacije štete

3. Proces analize kompenzacije kašnjenja

Tehnika koja se u radu analizira je premotavanje vremena. Cilj analize je ispitati granice uspješnog rada implementacije spomenute tehnike na objektivan način. Za odrediti je li sustav uspješan, potrebno je osmisliti hipotezu koji treba potvrditi određenim podacima koji se mogu prikupiti izvođenjem metodologije. Metrika uspješnosti je definirana tako da ako je tenk tijekom testiranja pogođen, smatra se da sustav radi uspješno. Donekle slična (objektivna) analiza ovoj (od ovoga rada) je odrađena u radu „*Objective and subjective evaluation of the influence of small amounts of delay and jitter on a recent first person shooter game*“ [20].

3.1. Hipoteza

Za objekte (ili u slučaju igre, tenkove) koji se kreću konstantnom brzinom (iznosa 44 km/h ili slično, tj. ispod 60 km/h radi jednostavnosti testiranja) i u jednom trenutku bivaju pogođeni, udaljenost između pozicije pogođenog tenka (na klijentu koji puca) i pozicije kompenziranog „sudarača“ tog objekta (isto promatrane iz klijenta koji puca) je proporcionalna iznosu mrežnog kašnjenja što je prikazano formulom (1).

$$distance(P_o, P_k) \sim lat, lat < maxlat \tag{1}$$

Gdje je:

- P_o - pozicija objekta
- P_k - pozicija kompenziranog sudarača
- $distance(A, B)$ - euklidska udaljenost između točaka A i B
- lat – mrežno kašnjenje (engl. Latency)
- $maxlat$ – maksimalno podržano mrežno kašnjenje za kompenzaciju

Drugim riječima, što je mrežno kašnjenje veće, spomenuta udaljenost također postaje sve veća dok god vrijeme ne prođe maksimalno podržano, tada kompenzacija „stagnira“ tj. kvaliteta pogodaka otpada. U nekim slučajevima se može dogoditi i da sustav uopće ni ne registrira pogodak što ovisi o brzini i veličini mete. Udaljenost između objekta i njegovog kompenziranog sudarača postaje konstanta što je prikazano formulom (2).

$$distance(P_o, P_k) \approx const. \vee \emptyset, lat \geq maxlat \tag{2}$$

Gdje $const.$ reprezentira brojčanu konstantu, a $\emptyset$ označava da nije bilo pogotka.

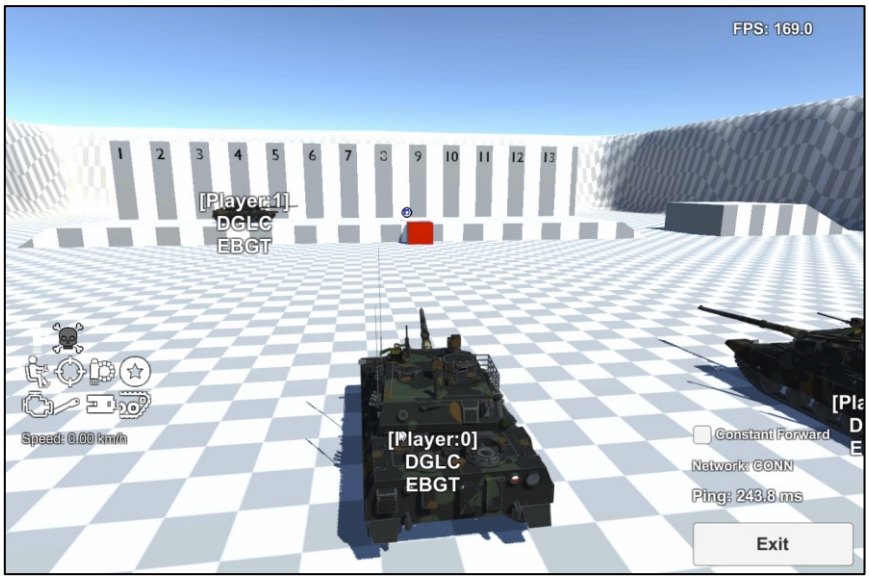
Uz stagnaciju sustava, očekuje se da i odrađena šteta na tenku (meti) postaje slabija, sustav kompenzacije kašnjenja i dalje može „ dobro/uspješno raditi“, no točka pogotka neće biti dovoljno točna iz čega simulacija štete počinje „zakazivati“. To se može prikazati formulom (3).

$$damage(a,s) \leq damage(b,s), a \in [maxlat, +\infty >, b \in [0, maxlat > \quad (3)$$

Gdje je $damage(t,s)$ očekivana šteta (realan broj) za kašnjenje t i situaciju gađanja s koja reprezentira skup okolnosti kod pogotka (npr. kut gađanja, pozicija (točka) pogotka i sl.).

3.2. Sustav za mjerenje

Sustav se sastoji od 3 instance igre, Host (oznaka H – igrač koji je poslužitelj), prvi klijent (K1 – klijent1) i drugi klijent (K2 – klijent 2). Uloga Hosta je da kreira sobu za igranje na koju se spajaju klijenti. Instance igre mogu biti pokrenute na istom računalu ili na različitim računalima. U svakom slučaju je potrebno imati mogućnost određivanja mrežnog kašnjenja i gubitka paketa. Početno mrežno kašnjenje tj. najmanje moguće mrežno kašnjenje treba biti do 30 ms, a gubitak paketa također treba biti minimalan moguć tj. u intervalu od 0% do 2%. Drugim riječima, potrebni su „normalni“ mrežni uvjeti [21]. Mjerenje može provesti jedan čovjek (tester - idealno kada se sve odvija na jednom računalu) ili više ljudi (kada se stvari provode na više računala - testeri). Mjerenja su vremenski osjetljiv postupak i potrebno je imati dobru i brzu agilnost rada. Za mjerenje postoji posebna „soba“ u igri. Soba za testiranje sadrži pomoćni objekt za ciljanje i određivanje vremena kada treba ispaliti granatu zvan „plava kocka“.U sobi za testiranje, igrači su posebno pozicionirani kada se pridruže igri i kada se ponovo stvaraju nakon uništenja tenka tako da je testiranje jednostavnije. Primjer pozicioniranja tenkova za testiranje je na slici 3.1:



Slika 3.1 Primjer pozicioniranja tenkova za testiranje, Player:0 i Player:1 su klijenti

3.3. Metodologija

Sustav za prikupljanje podataka (testiranje) je takav da se između dva klijenta K1 i K2, obavljaju dvije radnje. K1 u određenom trenutku puca u tenk od K2 koji se kreće konstantnom brzinom. Nakon pucnja, određuje se je li tenk bio pogođen i u slučaju da je, potrebno je očitati izmjerenu udaljenost i štetu na tenku od K2. Ako tenk nije bio pogođen, udaljenost se zapisuje kao „nepostojeći znak“ npr. „/“, a šteta je 0.

Za precizno gađanje za K1 tj. umanjivanje skupa okolnosti kod gađanja tenka za mjerenje štete se koristi pomoćni objekt „plava kocka“ (za ciljanje) na koju se prije pucanja i puštanja tenka od K2 nacilja nišanom. Iako je skup okolnosti time umanjen, on je i dalje ovisi o vremenu reakcije testera da odradi pucanj što opet može imati utjecaj na rezultate mjerenja gubitka posade, no točno mjerenje gubitka posade je općenito težak zadatak (u realnom vremenu) te su spomenute mjere dovoljno optimalne za njegovo mjerenje. Trenutak pucanja K1 na K2 je kada tenk od K2 „pojede“ tj. prijeđe pomoćni objekt („plavu kocku“).

Testiranje se obavlja više puta za razne iznose mrežnog kašnjenja (skup pozitivnih realnih brojeva), gubitka paketa (skup realnih brojeva u intervalu $[0, 1]$) i strane gađanja (skup diskretnih vrijednosti npr. „sa strane“ i „sprijeda“, također zvano „kut gađanja“). Ukupan broj testiranja (mjerenja) je prikazan formulom (4).

$$M = R * L * G * S \tag{4}$$

Gdje je:

- M – ukupan broj mjerenja
- R – broj ponavljanja mjerenja za istu trojku parametara $(l, g, s), l \in L, g \in G, s \in S$
- L – broj vrijednosti u skupu mrežnih kašnjenja
- G – broj vrijednosti u skupu gubitka paketa
- S – broj vrijednosti u skupu strana gađanja

3.4. Proces mjerenja

Klijenti i poslužitelj mogu biti pokrenuti na istom ili na različitim računalima (svako na jednom). U svakom slučaju potrebno je imati opciju simuliranja mrežnog kašnjenja i gubitka paketa.

Konkretni koraci kod testiranja su sljedeći:

1. Postaviti mrežno kašnjenje na potrebne vrijednosti
2. Prilagoditi tenk od K1 na prikladnu poziciju ovisno o kutu gađanja te postaviti nišan na pomoćni objekt za ciljanje
3. Prilagoditi kupolu od K2 tako da je orijentirana u smjeru šasije tenka K2
4. Pokrenuti tenk K2 na konstantno kretanje unaprijed
5. Na klijentu K1 pričekati da K2 „pojede“ pomoćni objekt i odmah ispaliti rundu
6. Očitati vrijednosti udaljenosti i gubitka posade (čak iako nije bilo pogotka)
7. Ako je potrebno, uništiti tenk od K2 za „reset“ testa te pričekati da se ponovo stvori
8. Ako nisu testirane sve trojke parametara potreban broj puta, odi na korak 1.
9. Kraj testiranja

Nakon svih mjerenja, potrebno je analizirati podatke i na temelju analize zaključiti da li podaci potvrđuju hipotezu ili ne. Iz ponovljenih mjerenja je potrebno izračunati srednje vrijednosti i standardne devijacije (za privid u odstupanja). Dobivene srednje vrijednosti i standardne devijacije se onda mogu prikazati u grafovima tipa udaljenost/kašnjenje (za određeni gubitak paketa i stranu gađanja), udaljenost/gubitak (za određeno kašnjenje i stranu gađanja), gubitak posade/kašnjenje (za određeni PL i stranu gađanja) i gubitak posade/PL (za određeno kašnjenje i stranu gađanja). Slučajevi kada je tenk potpuno promašen se također reprezentiraju grafom u kojemu je prikazan broj promašaja/kašnjenje (za određeni PL i stranu) i slično. Ako tijekom testiranja nije bilo ni jednog promašaja, grafovi za promašaje tada nisu potrebni. Dobiveni grafovi se mogu i kombinirati u jedan za lakši pregled.

4. Izrada igre

Za izradu igre korišten je program Unity Engine (verzija 2022.3.1f1), a za umreženi dio igre je korištena biblioteka Photon Fusion (1.1.8 F 725). Za 3D modeliranje određenih objekata korišten je program Blender (verzija 3.0 i 4.0). Za pisanje koda korišten je program Visual Studio 2022 (verzija 17.8). Razvoj je odrađen na računalu sljedećih specifikacija:

- CPU: Intel Core i-9 12900k
- GPU: Nvidia GeForce RTX 3090 24 GB
- RAM: 64 GB DDR5 (5200 MHz)
- SSD: 5 TB

Performanse igre na navedenom računalu su 144 FPS (v-sync ograničenje).

3D Model tenka u igri je zvan Osório (drugi prototip). "EE-T1 Osório" (https://skfb.ly/6UnvZ) by Pedro B. Goulart is licensed under Creative Commons Attribution (http://creativecommons.org/licenses/by/4.0/).

3D Model tenkovske granate DM 33 (na njemačkom: Deutsche Model 33) APFSDS (engl. Armor Piercing Fin-Stabilized Discarding Sabot) je preuzet od: "DM 33 APFSDS" (https://skfb.ly/oI7FW) by HEAVYCLOUD is licensed under Creative Commons Attribution (http://creativecommons.org/licenses/by/4.0/).

3D modeli posade (za prikaz unutarnjih komponenti) su preuzeti od: "Male Character PS1-Style" (https://skfb.ly/o7JPX) by vinrax is licensed under Creative Commons Attribution (http://creativecommons.org/licenses/by/4.0/).

Unutarnje komponente imaju konveksne sudarače (zahtjev od Unityja), za konverziju nekonveksnih 3D oblika u konveksne dijelove korišten je program koji to radi aproksimativno: https://github.com/kmammou/v-hacd

Ikone za HUD su preuzete iz: Skull icons created by Freepik – Flaticon

Iako je igra „realističnija“, modeliranje tenkova i tenkovskih granata je „nasumično“ tj. komponente se ne ponašaju isto kao u realnom svijetu radi: jednostavnosti modeliranja i činjenice da informacije o tenku (Osório) i granatama (DM 33) nisu javno dostupne, već su (vrlo vjerojatno) klasificirane/povjerljive (pogotovo za granatu).

4.1. Photon Fusion

Fusion je biblioteka za umrežavanje digitalnih igara, namijenjen je za uporabu s pogonom Unity. Fusion je mrežna biblioteka „više razine“ što znači da se developer aplikacije koja koristi Fusion ne mora brinuti o „nižim“ stvarima npr. kompresija podataka i slanje istih preko mrežnog socketa. [22]

Fusion ima više načina rada (mrežnih topologija): Shared, Client-Host i Dedicated Server. Igra u radu koristi Client-Host način rada gdje jedan od igrača uzima ulogu poslužitelja. Dedicated Server je slična opcija Client-Host načinu, no umjesto da je jedan igrač poslužitelj, poslužitelj je zasebno računalo „u oblaku“. Prednost Dedicated Server topologije je u tome da je zaštita od varanja jako dobra, dok u Client-Host topologiji igrač koji je poslužitelj može varati. Nedostatak Dedicated Server topologije je visoka cijena (poslužitelj mora izvoditi instancu igre za svaku borbu/meč), dok je Client-Host mnogo jeftiniji. Povezivanje klijenata u Client-Host topologiju je odrađeno Photnonovim poslužiteljima.

Fusion ima dvije vrste autoriteta: ulazni (engl. Input Authority) i stanja (engl. State Authority). Igrač poslužitelj može varati zato što on ima autoritet stanja nad svim mrežnim objektima. Svaki od klijenata tijekom pridruživanja igri dobiva svoj mrežni objekt (tenk) i nad njim on jedini ima autoritet za ulaz.

Fusion za umrežavanje i sinkronizaciju koristi vlastite mrežne otkucaje (mrežnu simulaciju) u metodi FixedUpdateNetwork koje se izvode u fiksnom vremenskom intervalu uz moguća ponavljanja tih mrežnih otkucaja na klijentima (ponovna simulacija). Umjesto standardne MonoBehaviour klase koju koriste obične Unity skripte, Fusion uvodi NetworkBehaviour tip koji je naslijeđen od MonoBehaviour klase. Svaki NetworkBehavior ima pristup NetworkRunner objektu uz pomoć kojega može baratati s mrežnim djelom igre. Varijable u kodu (Unity skriptama) koje je potrebno sinkronizirati putem mreže unutar NetworkBehaviour skripte se može odraditi dodavanjem [Networked] atributa iznad potrebnih varijabli, time se osigurava mrežna sinkronizacija tih varijabli. Za sinkroniziranje postojećih Unity komponenti npr. tipa Rigidbody ili Transform postoje prikladne Fusion komponente NetworkRigidbody i NetworkTransform. Primjer korištenja Fusiona u kodu (jako skraćeni dio skripte za upravljanje tenkom gdje se postavlja rotacija kotača (za kretanje tenka) s obzirom na ulaz):

```
1. public class PlayerTankController : NetworkBehaviour
2. {
3.     ...
```

```
4.
5.      [Networked]
6.      public float CurrentRotationSpeed { get; set; }
7.      [Networked]
8.      public float CurrentTraverseSpeed { get; set; }
9.
10.     ...
11.
12.     public override void FixedUpdateNetwork()
13.     {
14.         ...
15.         if(GetInput(out NetworkInputData data))
16.         {
17.             ...
18.             //traverse processing (example from inlined function)
19.             if(data.ForwardPressed)
20.                 SetRotation(CurrentTraverseSpeed, WheelSide.Right)
21.             ...
22.         }
23.     }
24. }
```

Za kontrolu mrežnih događaja, dovoljno je napraviti Unity skriptu te joj dodati (i implementirati) sučelje INetworkRunnerCallbacks gdje se nalaze metode za stvaranje soba, izlaz iz umrežene igre, procesiranje dolaska ili odlaska igrača, procesiranje ulaza, prekidanje veze s poslužiteljem i slično.

Za spajanje u sobu ili kreiranje nove, potrebno je, između ostalog, pozvati funkciju u kojoj se specificira ime sobe, način pridruživanja (host ili klijent) te ID Unity scene.

4.1.1. Projektili

Implementacija projektila (u Fusionu [23]) je bitan dio tehničke realizacije igre, bez kinetičkih projektila kojima treba određeno vrijeme da stignu do mete i koji se gibaju po paraboli (gravitacija), igra gubi na određenoj dozi realizma. Projektili u igri su realizirani kao posebne umrežene strukture podataka koje implementiraju strukturu INetworkStruct, to sučelje označava da se struktura može dijeliti putem mreže tj. neki od njenih dijelova (varijabli) označenih s [Networked]. Ostale varijable u takvoj strukturi bi trebale biti konstante. Korištena struktura podataka u igri je ovakva:

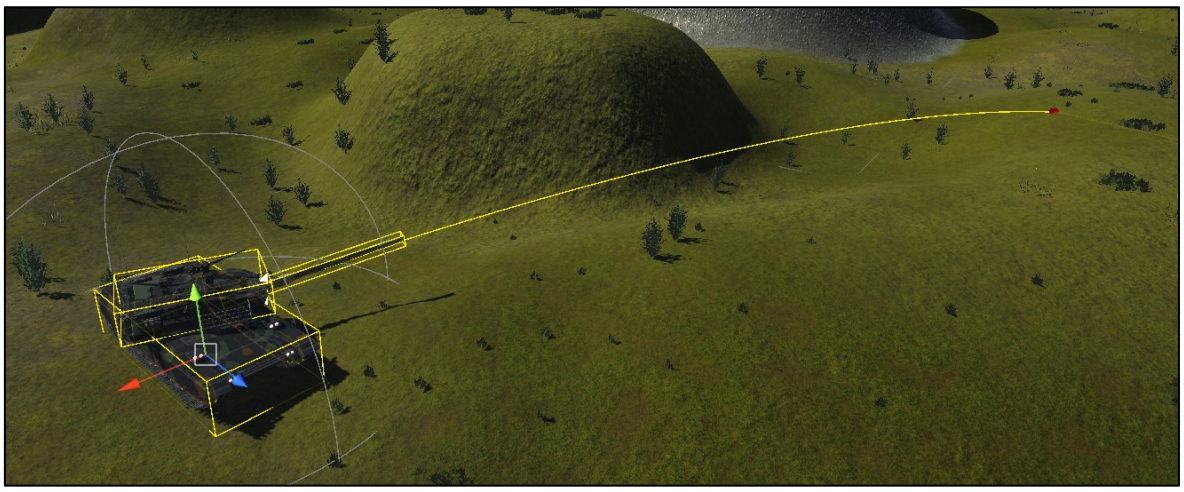
```
1.  private struct ProjectileData : INetworkStruct
2.  {
3.      public bool IsActive => FireTick > 0;
4.
5.      public int FireTick;
6.      public int FinishTick;
7.      public int Index;
8.
9.      public Vector3 FirePosition;
```

```
10.     public Vector3 FireVelocity;
11.
12.     [Networked]
13.     public Vector3 HitPosition { get; set; }
14.     [Networked]
15.     public Vector3 HitDirection { get; set; }
16.
17.     public Vector3 PointingDirection;
18. }
```

Struktura ProjectileData se kasnije koristi u umreženoj listi. Informacije o projektilu tipa putanja su posebno izračunate:

```
1.  private Vector3 GetMovePosition(ref ProjectileData data, float
    currentTick)
2.  {
3.      float time = (currentTick - data.FireTick) * Runner.DeltaTime;
4.
5.      if (time <= 0f)
6.          return data.FirePosition;
7.
8.      return data.FirePosition + data.FireVelocity * time + time * time *
    Physics.gravity;
9.  }
```

Pozicija projektila (tenkovske granate) se računa ovisno o trenutnom otkucaju. Uz pomoć te metode se grafički prikaz projektila ažurira, a on je zaseban objekt u igri. Putanja projektila je „parabola“ tj. parabola sačinjena od linija, te linije su i zrake koje se koriste za provjeru je li projektil nešto pogodio (uz korištenje kompenzacije kašnjenja). Primjer trajektorija projektila je prikazan na slici 4.1:

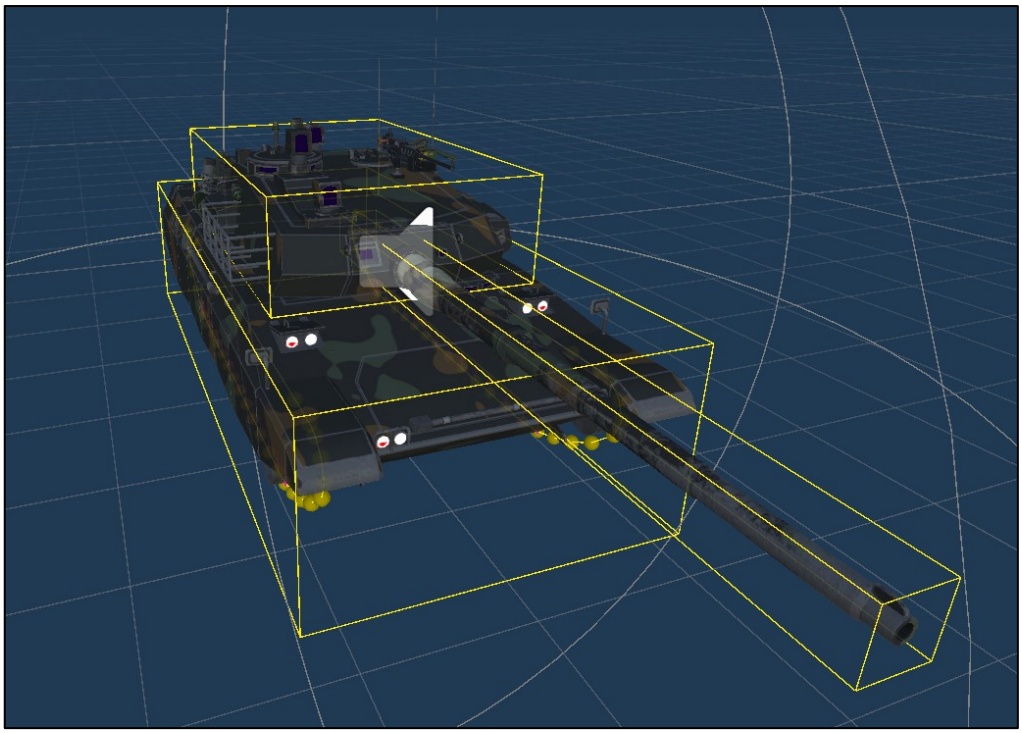


Slika 4.1 Trajektorija projektila (s umanjenom brzinom radi ljepšeg primjera)

Pogotkom nekog objekta stvara se nova struktura (po istom obrascu kao i ProjectileData), no ovaj puta za sinkronizaciju informacija o pogodcima drugih tenkova (ProjectileHitInfo). Kada se informacije o pogodcima sinkroniziraju, radi se simulacija štete.

4.1.2. LagCompensation klasa

U igri u radu su korištene dvije tehnike kompenzacije kašnjenja: premotavanje vremena i samostalno predviđanje. Samostalno predviđanje je realizirano tako da klijenti (uz poslužitelja) također izvode simulaciju štete (radi grafičkog ispisa što je oštećeno, nema efekt na mehaniku igre klijenta), no autoritet nad stanjem svih objekata ima samo poslužitelj. Samostalno predviđanje se koristi i kod kretanja tenkova da ono izgleda „glađe“ ako je predikcija bila dobra [24]. Druga tehnika koja je primarno u fokusu rada je premotavanje vremena. Fusion ima ugrađenu podršku za premotavanje vremena u obliku posebnih sudarača koji su kompenzirani kašnjenjem: hitboxovi. Za korištenje hitboxova potrebno je dodati komponentu HitboxRoot na objekt koji koristi hitboxove (uključujući i njegovu djecu) te nakon toga dodati hitboxove na njihova mjesta (tj. prikladne objekte). Tenkovi u igri imaju 3 hitboxa koji ih u potpunosti „zaokružuju“. Jedan hitbox je za šasiju, drugi za kupolu, a treći za top što je prikazano na slici 4.2:



Slika 4.2 hitboxovi tenka prikazani su žutim linijskim kvadrima

Za (kompenziranu) provjeru je li hitbox pogođen koristi se posebna metoda i klasa LagCompensation:

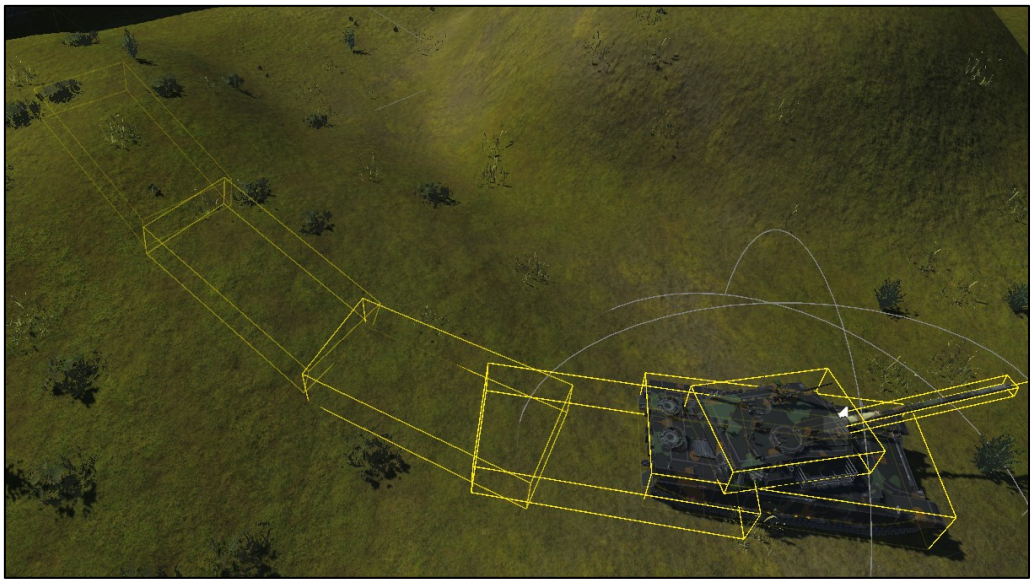
```
Runner.LagCompensation.Raycast(…)
```

Metoda koristi algoritam bacanja zraka za provjeru pogodaka hitboxova (koji su tada kompenzirani radi kašnjenja) i običnih Unityjevih sudarača (nisu kompenzirani).

Način na koji kompenzacija kašnjenja radi u Fusionu je tako da Fusion sprema prijašnje pozicije (i rotacije) hitboxova na određen broj mrežnih otkucaja [25]. U projektu je broj „kompenziranih“ otkucaja postavljen na 16 što uz činjenicu da je frekvencija mrežnih otkucaja u igri 60 Hz znači da je kompenzirano vrijeme kašnjenja (teorijski limit) izraženo formulom (5):

$$16 * \frac{1}{60} = 266.67\,ms \tag{5}$$

Grafički prikaz prijašnjih pozicija hitboxova za tenk koji se kreće izgleda (približno) ovako:



Slika 4.3 Povijest hitboxova šasije tenka

Na slici 4.3 prikazane su pozicije i rotacije šasije tenka koji se kreće niz brdo i tijekom toga lagano skreće, stariji hitboxovi postaju prozirniji. U realnom slučaju su stariji hitboxovi zapravo dosta bliži jedan drugome. Na slici situacija baš i nije realna, ali služi kao dobar uvid/primjer rada sustava.

4.1.3. Kompenzacija dijelova tenka

Osim kompenzacije u postupku detekcije je li određen tenk pogođen, potrebno je kompenzirati pozicije i rotacije (unutarnjih) dijelova tenka kako bih simulacija štete također bila dobro kompenzirana.

Sažeta hijerarhija unutarnjih, pa tako i vanjskih dijelova tenka je sljedeća:

- Šasija (engl. Chassis)
    - Kupola (engl. Turret)
        - Top (engl. Barrel)

Za svaki od tih dijelova postoji prikladan hitbox, što je dobar početak, zato što bi bez te činjenice bilo nemoguće dobro kompenzirati njihove pozicije i rotacije. Kompenzacija tih pozicija i rotacija je takva da se (kompenzirane) pozicije i rotacije hitboxa koriste kao pozicije i rotacije unutarnjih (tj. njihovih pripadnih) objekata tenka, time se osigurava da su dijelovi tenka tijekom simulacije štete kompenzirani. Fusion ima posebnu funkciju kojom se dobivaju kompenzirane pozicije i rotacije hitboxa:

```
Runner.LagCompensation.PositionRotation(root.Hitboxes[i], player, out
HitboxPosition, out HitboxRotation);
```

Uz kompenzaciju dijelova tenka, potrebno je i kompenzirati i točku pogotka. Iako Fusion već daje kompenziranu točku pogotka, no zbog implementacijskog detalja, tu točku je potrebno transformirati u prostor unutarnjih objekata. Unutarnji objekti tenka su implementirani kao obični sudarači (engl. colliders) koji se kao objekti odvoje od pripadnog tenka prilikom instanciranja. Zbog toga ti objekti ni ne „prate“ tenk kako se on pomiče, već samo stoje na istom mjestu (i rotiraju se po potrebi – kada je tenk pogođen), a to je zato što je jednostavnije ne zaustavljati (ili posebno rotirati) cijeli tenk prilikom simulacije štete što bih izgledalo „čudno“. Kod funkcije za prebacivanje točke pogotka između sustava je sljedeći:

```
1.  Vector3 TransfromFromObjectCoords(Vector3 point, Transform a,
    Transform b)
2.  {
3.      Vector3 local = a.InverseTransformPoint(point);
4.      return b.TransformPoint(local);
5.  }
```

Ideja je pretvoriti točku u lokalne koordinate hitboxa, pa onda te lokalne koordinate (od tog pripadnog hitboxa) prevesti u globalne koordinate pripadnog unutarnjeg objekta.

Ovim postupkom točka i dalje ostaje kompenzirana, te se samo premješta na prikladno mjesto za simulaciju štete.

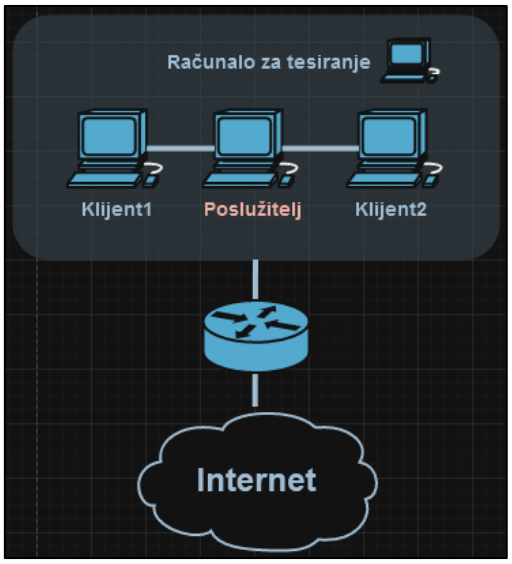
5. Proces mjerenja i analize rezultata

Testira se uspješnost kompenzacija kašnjenja s obzirom na sljedeće mrežne parametre: mrežno kašnjenje (RTT - ping) i gubitak paketa. Mjerenje se također mjeri i za gađanje iz više kuteva (sprijeda i s desne strane). Uspješnost je definirana tako da ako je tenk pogođen, smatra se da sustav dobro funkcionira.

Platforma (Operacijski sustav) na kojoj je testiranje obavljeno je MS Windows 10, no procedura testiranja je identična i za testiranje na novijim verzijama MS Windows sustava. Za testiranje je potrebno imati internetski pristup (konekcija s Photon serverima) i projekt treba imati konfiguriran Photon AppId.

Prije testiranja, potrebno je podesiti određene parametre u igri te prevesti igru (File->Build Settings->Build). Parametri koji moraju biti omogućeni su „Debug Mode“ za objekt „Prefabs/Tanks/OsorioPrototype.prefab“ tj. parametar Debug je potrebno staviti na true u komponenti PlayerTankController (slika 5.1). Mjerenja/testiranje se obavlja na jednom računalu tako da se igra pokreće u više instanci (više programa), topologija takve simulirane mreže je prikazana na slici 5.2.

Slika 5.1 Debug opcija u PlayerTankController komponenti



Slika 5.2 Topologija mreže tijekom testiranja

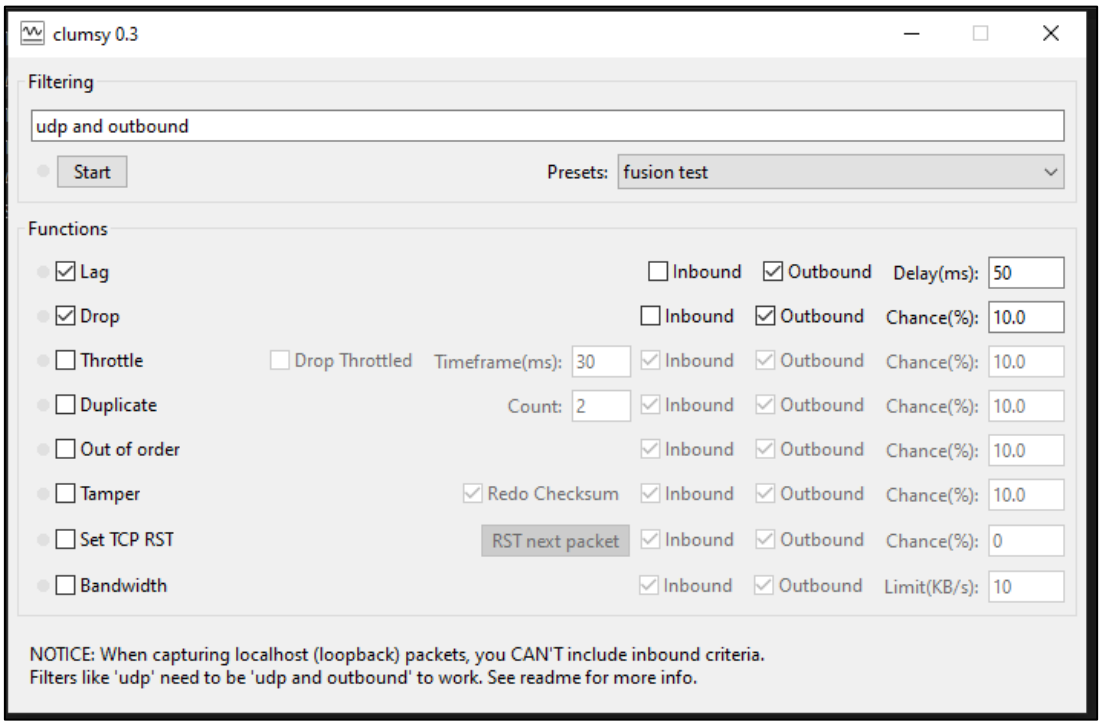
5.1. Simuliranje mrežnog kašnjenja

Mrežno kašnjenje je simulirano programom Clumsy [26]. Program Clumsy neće dati uvijek 100% isto kašnjenje, nego će uvijek biti varijacija kod kašnjenja ili gubitka paketa što je u redu jer se mjerenja ionako ponavljaju.

Simuliranje mrežnog kašnjenja je odrađeno programom Clumsy, korištena konfiguracija (filter) je iz izvora [27]:

```
fusion test: udp and outbound
```

U datoteci config.txt spomenutog programa, a nalazi se u istom direktoriju kao i izvršna datoteka programa. Program je jako jednostavan za korištenje, no zahtjeva administrativne privilegije za uspješan rad (simulaciju mrežnog kašnjenja i gubitka paketa – slika 5.3):



Slika 5.3 Primjer konfiguracije u programu Clumsy

Za testiranje je potrebno postaviti vrijednost izbornika „Presets“ na „fusion test“, te po potrebi omogućiti opcije „Lag“ (mrežno kašnjenje) i „Drop“ (gubitak paketa). Za navedene opcije treba isključiti postavku „Inbound“ i uključiti postavku „Outbound“. Same vrijednosti mrežnog kašnjenja i gubitka paketa se postavljaju u pripadnim poljima „Delay(ms)“ i „Chance(%)“. Potrebno je imati na umu da je mrežno kašnjenje u igri dvostruko veće od zadanog u programu Clumsy, a to je zato što Clumsy postavlja OWD (One Way Delay) vrijeme dok igra prikazuje RTT, u to su moguće i oscilacije samog mrežnog kašnjenja. Isto

vrijedi i za gubitak paketa, stoga se njegova vrijednost postavlja na rješenje x iz sljedeće jednadžbe (6):

$$1 - (x - 1)^2 = p \tag{6}$$

Gdje je p željeni postotak stvarnog gubitka paketa u „simuliranoj“ mreži. Očito je da dana jednadžba ima dva rješenja, no uzima se rješenje iz intervala $[0, 1]$ jer se traži vjerojatnost dok je drugo rješenje uvijek veće ili jednako 1 za $p \in [0,1]$.

5.2. Postupak

Za mrežno kašnjenje se koriste sljedeće vrijednosti (u milisekundama): $[20\ ms, 80\ ms, 150\ ms, 250\ ms, 300\ ms]$.

Za gubitak paketa koriste se sljedeće vrijednosti (u postocima): $[0\%, 10\%, 20\%, 40\%]$. Unosi u Clumsy za gubitke paketa u $[0\%, 10\%, 20\%, 40\%]$ su zapravo $[0\%, 5.132\%, 10.557\%, 22.54\%]$.

Vrijednosti koje se mjere su:

1. Udaljenost pozicije kompenziranog hitboxa i trenutne pozicije hitboxa (realan broj ili znak „/“ kada nema pogotka)
2. Broj izgubljenih ljudi u posadi pogođenog tenka (cijeli broj iz skupa $\{0, 1, 2, 4\}$ ili znak „/“ kada nema pogotka)*

* Broj 3 nije u broju izgubljenih ljudi jer sa smrću 3 člana tenk postane uništen, pa tako i jedini preostali član posade „pogine“.

Za svaku trojku (mrežno kašnjenje, gubitak paketa, kut gađanja), mjerenje se ponavlja 10 puta.

Ukupan broj mjerenja je $5\ x\ 4\ x\ 2\ x\ 10\ =\ 400$ mjerenja.

Rezultati mjerenja se upisuju u pripadne tablice u radnom listu MS Excel programa (ili slično). Iščitavanje broja poginule posade se radi brojanjem nestalih slova u drugom redu na debug tekstu od pripadnog tenka, tenk s čitavom posadom ima 4 slova: DGLC (engl. Driver Gunner Loader Commander), svako pripada jednom (živom) članu posade, ako je član poginuo onda se njegovo slovo izbriše iz teksta.

Za testiranje se koristi scena „Test Polygon“ (opcija u glavnom meniju igre). Potrebno je pokrenuti 3 instance igre (na istom računalu), dvije instance su prevedena igra

(„DipRadProjectFusion.exe“ datoteka tj. izvršna datoteka), a treća je „play mode“ u Unity Editor programu. Jedan od igrača je Host (H), a druga dva su klijenti (oznake K1 i K2). Za početak testiranja potrebno je pokrenuti prvu instancu prevedene igre (preko izvršne datoteke), u glavnom meniju je potrebno odabrati „Test Polygon“ te nakon toga odabrati „Host“ (način rada poslužitelja) opciju, uz to se može i zadati ime sobe u igri, no tada je potrebno to isto ime zadati tijekom pokretanja drugih dviju instanci. Ovaj igrač koji je Host ne radi ništa tijekom konkretnog testiranja tj. ne kreće se nigdje, no može rotirati kupolu i pucati na klijenta K2 (samo kada je to potrebno). Nakon toga je potrebno pokrenuti igru u Unity Editoru, učitati scenu „Test Polygon“ te odabrati opciju „Join“ (način rada klijenta), igrač u Unity Editoru će se na dalje zvati „Klijent 1“ ili K1. U Unity Editoru je potrebno uz „Game“ prozor potrebno imati i otvoren „Console“ prozor na kojemu se iščitava udaljenost hitboxa. Treću instancu igre je potrebno pokrenuti u preko izvršne datoteke igre, učitati „Test Polygon“ scenu te odabrati opciju „Join“, ovaj klijent će se na dalje zvati „Klijent 2“ ili K2.
Algoritam mjerenja

U nastavku je dan (proširen) algoritam za mjerenja:

1. Postaviti potrebne mrežne parametre u programu Clumsy

2. Ovisno o kutu gađanja, tenk od K1 treba:

    - Ostati na mjestu u slučaju gađanja s desne strane

    - Pomaknuti se na uzvišenu platformu do testnog poligona u slučaju gađanja sprijeda te rotirati kupolu prema tenku K2

2. Rotirati kupolu od tenka K2 tako da je top orijentiran u pravcu smjera (rotacije šasije) tenka od K2

3. Rotirati kupolu od tenka K1 prema poziciji poligona s oznakom 8 tj. 8.5 (između 8 i 9), iznad crvene kocke i prilagoditi pomoćni nišan topa na plavu kocku uz pomoć snajperskog nišana

4. Pripremiti prozor igre K2 i prozor Unity Editora K1 za brzi prijelaz između njih (brzi alt-tab na Windows OS-u)

5. Prebaciti se na prozor od K2

5. Omogućiti "Constant Forward" opciju (tipka E na tipkovnici ili pritisak na gumb do pripadnog teksta) na instanci igre od K2

6. Brzo se prebaciti na prozor od Unity Editora (K1)

7. Pričekati da tenk od K2 dostigne točku 8.5 tj. "uđe u nišan" ili „pojede plavu kocku“ te odmah ispaliti rundu

8. Iz Konzole u Unity Editoru iščitati posljednju poruku tipa "Hitbox Dist: 0.435" (to je udaljenost hitboxova) te iz prozora "Game" iščitati koliko je članova posade poginulo nakon pogotka (vidljivo nakon 0.5 sekundi za svako od testiranih mrežnih kašnjenja) iz debug teksta koji je prati tenk od K2

9. Zapisati opažene vrijednosti, u slučaju da nije bilo pogotka zapisuje se znak "/"

10. Počistiti konzolu Unity Editora (gumb "Clear")

11. U igri od K2, isključiti opciju "Constant Forward"

12. Uništiti tenk od K2 (ima više načina: suicid (držanjem tipke J) ili ponovno gađanje s tenkovima od K1 ili H)

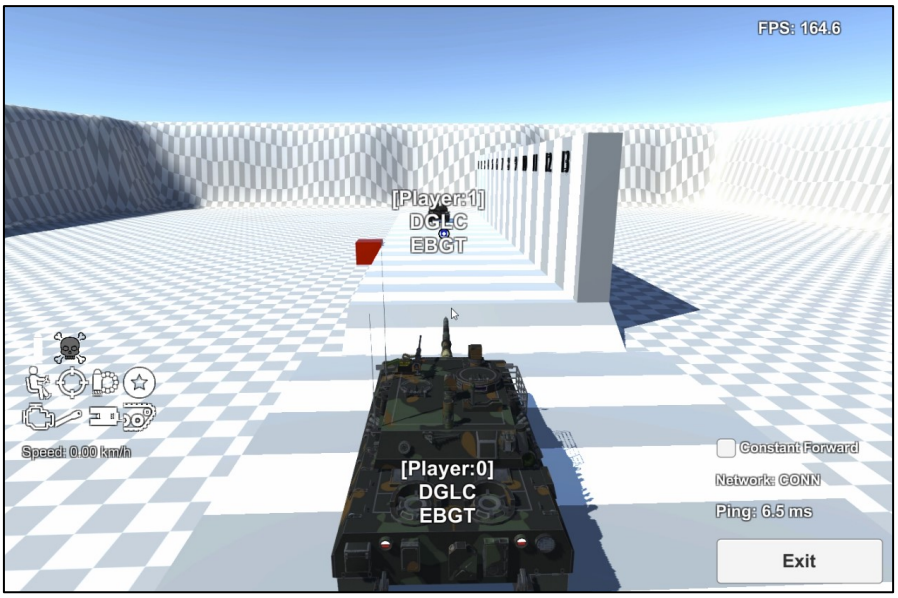
13. Pričekati da se tenk od K2 ponovo stvori (respawn)

14. Ako nisu testirane sve trojke mjerenja po potreban broj puta, odi na korak 1.

15. Kraj testiranja

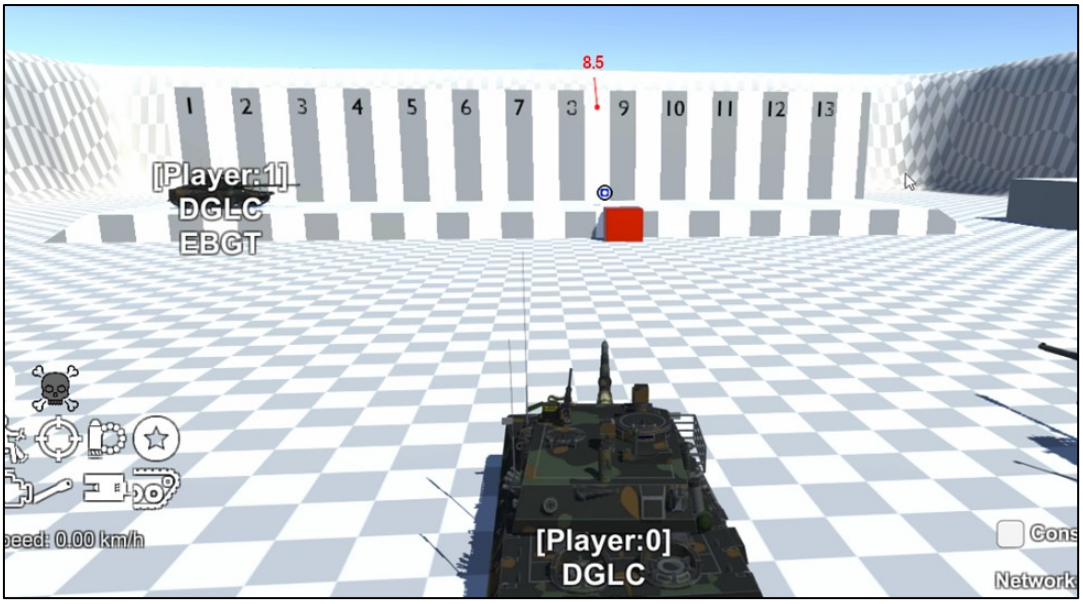
Naputci za algoritam:

- Kod gađanja sprijeda, pozicija K1 bi trebala biti ovakva (približno – slika 5.4):



Slika 5.4 Gađanje sprijeda

- Prilagođeni nišan na plavu kocku ne treba biti 100% centriran na njoj, no što bliže je bolje. Oznaka 8.5 je na donjoj slici 5.5:



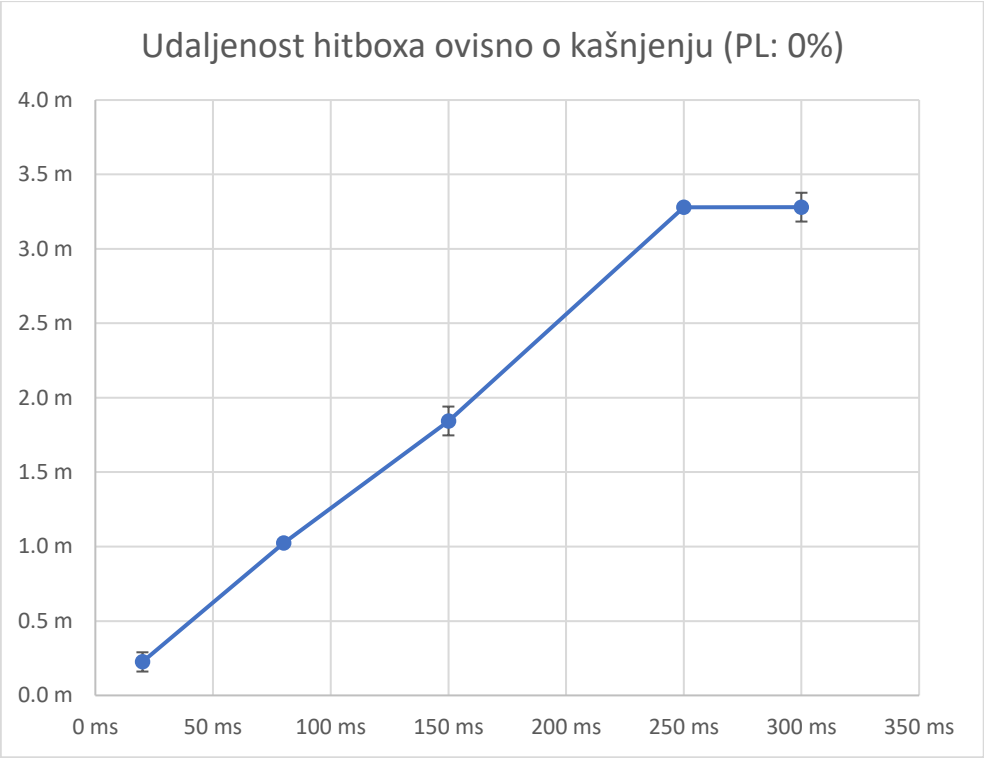
Slika 5.5 Gađanje sa strane

- Plava kocka se ponaša kao duh prema tenkovima (no ne i nišanu) tj. ne sudara se s njima i obratno
- Mjerenja se zapisuju (idealno u program tipa MS Excel) te se za svakih 10 ponavljanja (za istu stranu, mrežno kašnjenje i gubitak) računaju srednja vrijednost i standardna devijacija i onda se na temelju njih crtaju grafovi
- Mjerna jedinica za udaljenost hitboxa su „metri“ tj. jedinična udaljenost u Unity Engineu

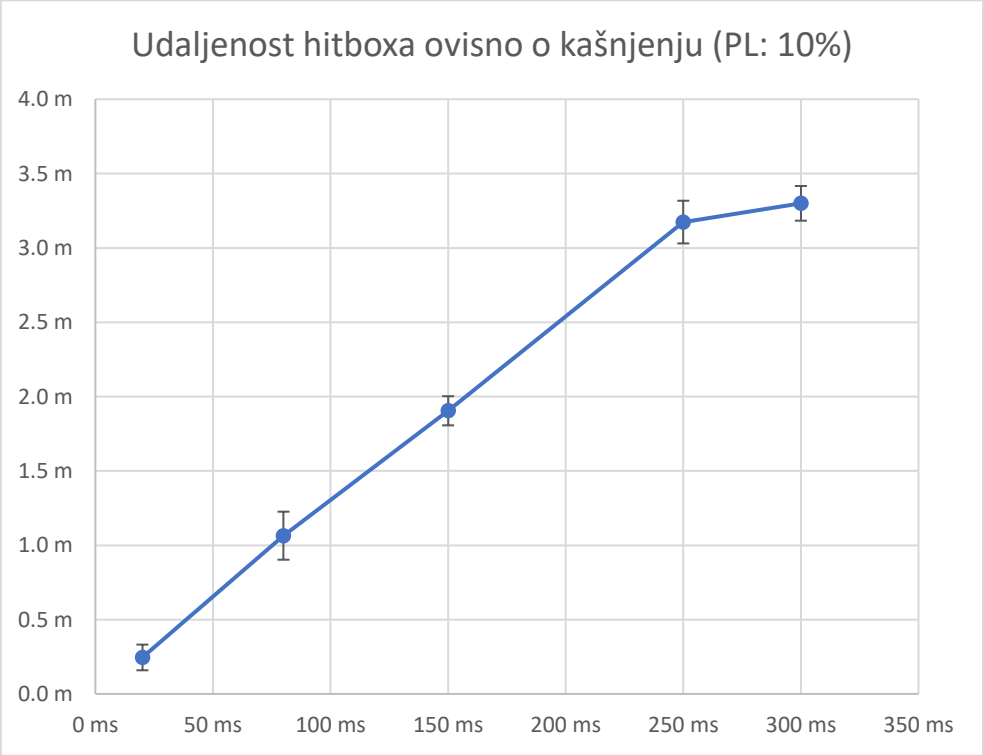
5.3. Analiza rezultata

5.3.1. Gađanje sa strane

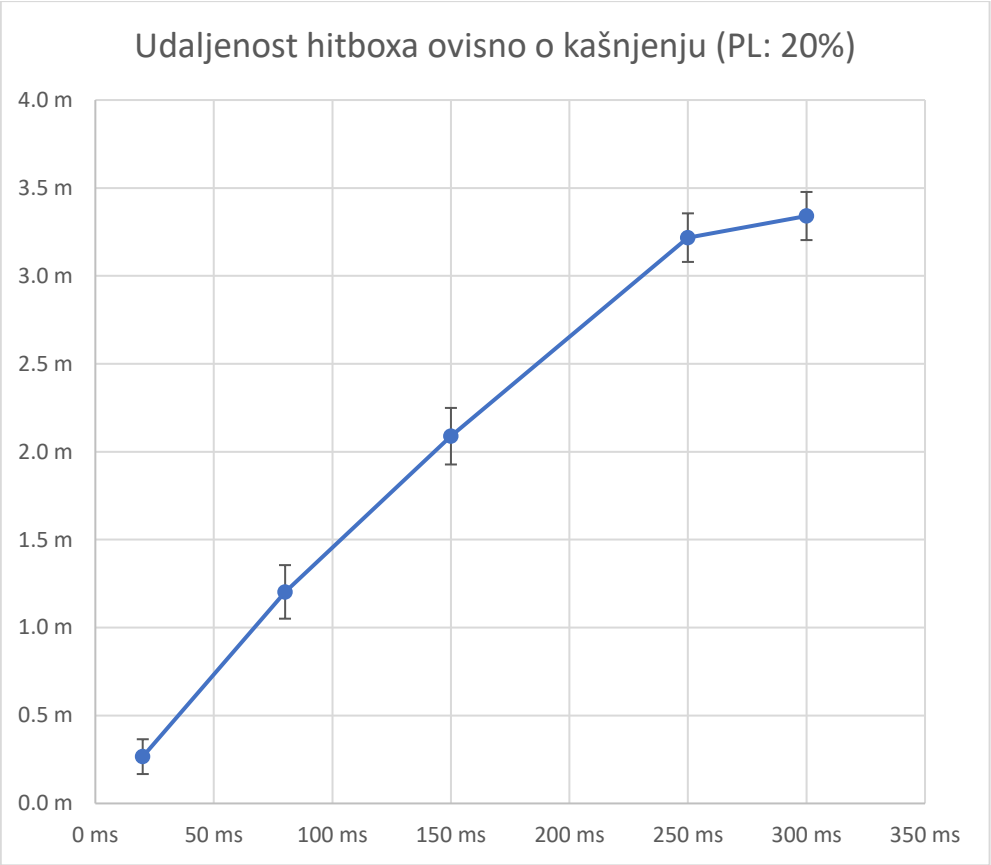
Za testirane vrijednosti, grafovi udaljenosti hitboxa su (gađanje s desne strane) prikazani slikama 5.6, 5.7, 5.8 i 5.9:



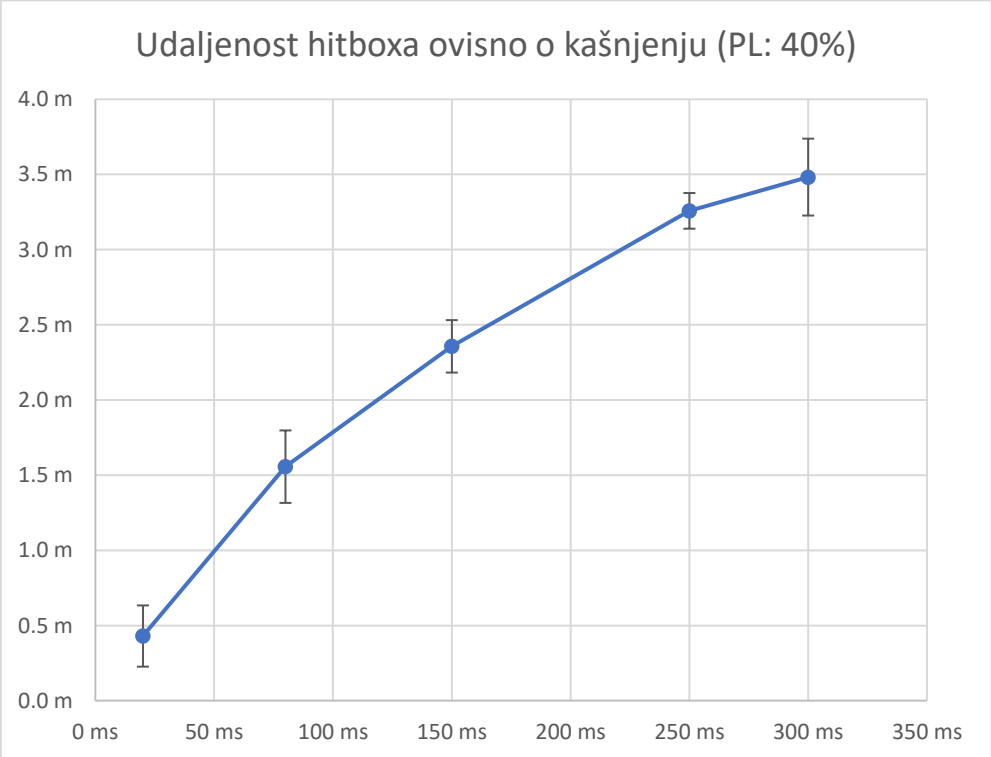
Slika 5.6 Graf za udaljenosti hitboxa, gađanje sa strane, PL: 0%



Slika 5.7 Graf za udaljenosti hitboxa, gađanje sa strane, PL: 10%



Slika 5.8 Graf za udaljenosti hitboxa, gađanje sa strane, PL: 20%

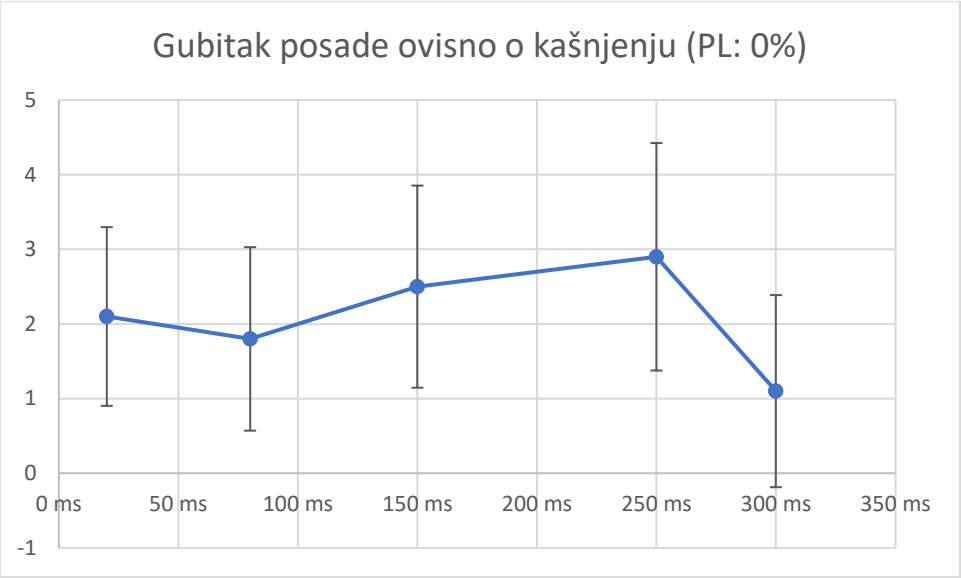


Slika 5.9 Graf za udaljenosti hitboxa, gađanje sa strane, PL: 40%

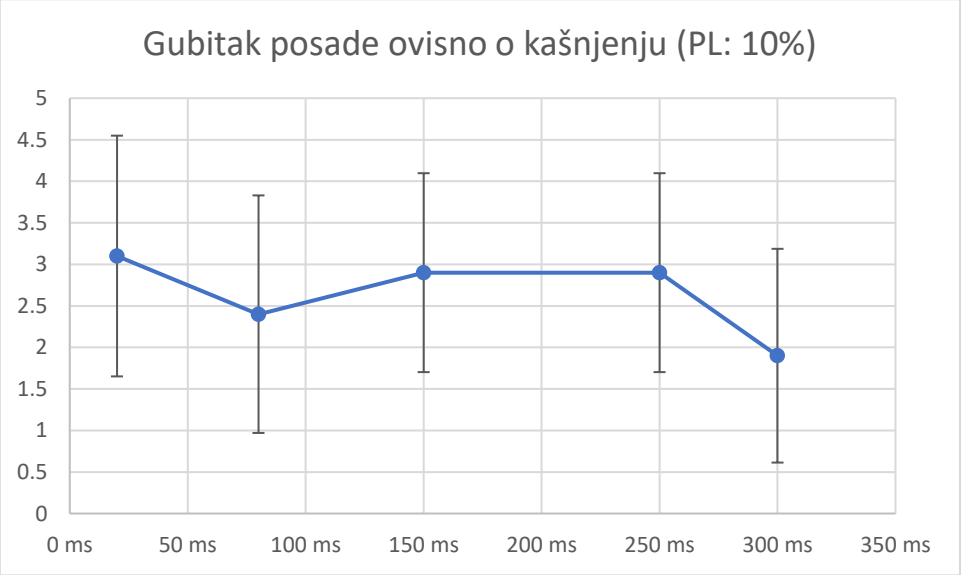
Iz danih grafova se vidi da je veza između udaljenosti i kašnjenja linearna sve do 250 ms, a tada udaljenost više ni raste značajno. To je zato što je u projektu konfigurirano da se hitboxovi tj. njihove pozicije i rotacije kompenziraju za 266.67 ms, a dobiveni podaci se i po tome dobro ravnaju (uz naravno manja odstupanja jer mjerenja nisu 100% savršena). Veza između kašnjenja i udaljenosti je linearna zato što se meta (tenk mete) giba konstantom brzinom prilikom pogotka tijekom testiranja.

Iz danih grafova se također vidi da udaljenost lagano raste s povećanjem gubitka paketa i odstupanja (standardne devijacije) također rastu. Iz toga se može zaključiti da gubitak paketa također ima utjecaj na kompenzaciju kašnjenja i da veće vrijednosti daju veći utjecaj.

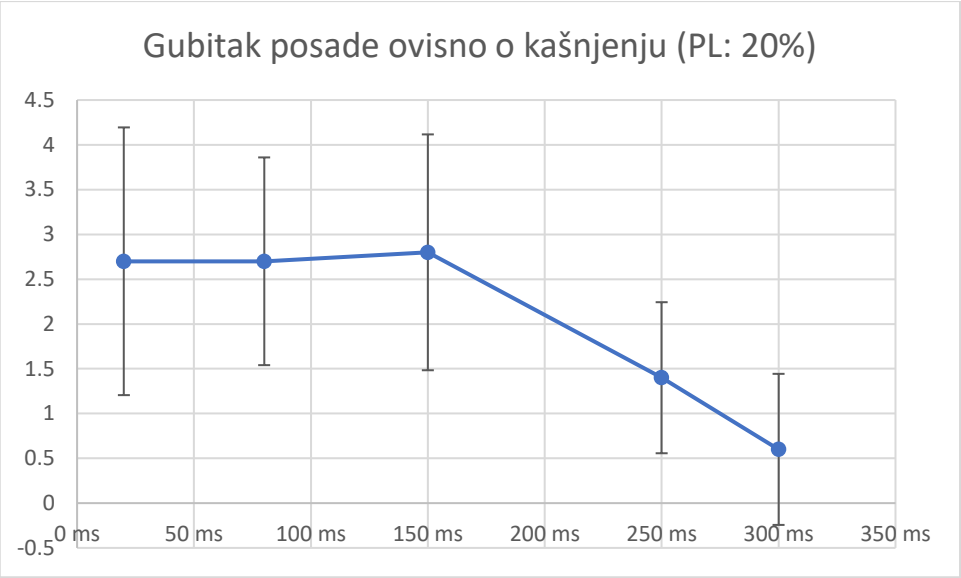
Potrebno je spomenuti da u ni jednom od testova nije bilo promašaja, što isprva zvuči kontra-intuitivno za kašnjenje od 300 ms. Stvar je u tome da iako je kašnjenje veće od podržanog, tenk je i dalje moguće pogoditi radi njegove veličine (duljina je ~6 metara, širina je ~2.6 metara, a visina je ~2 metara) i činjenice da je test osmišljen tako da je jako teško ne pogoditi tenk (pucanje odmah nakon što tenk „proguta“ plavu kocku). Iz podataka se vidi i da je moguća manifestacija „smrti iz zida“ za kašnjenja od ~200 ms, no samo u slučaju kada je polovica tenka još vidljiva napadaču jer je kompenzirana udaljenost ~3 metra, a duljina tenka je ~6 metara. No činjenica da kompenzacija i dalje „radi“ na kašnjenju većem od 267 ms ne znači da nema određenih gubitaka tj. drugi parametar mjerenja gubitak posade počinje „patiti“. To je prikazano sljedećim grafovima (slike 5.10, 5.11, 5.12, 5.13) gubitka posade ovisno o kašnjenju:



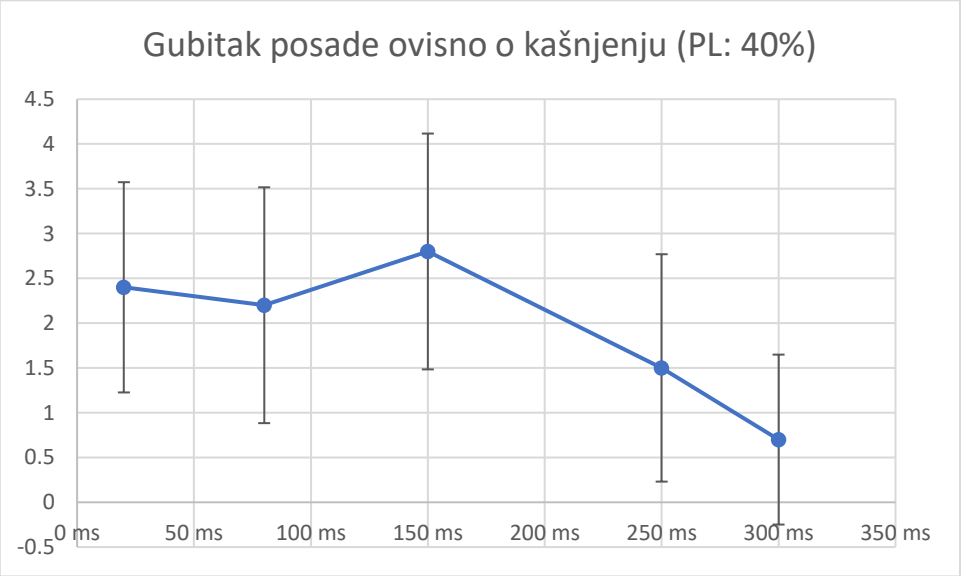
Slika 5.10 Graf za gubitak posade, gađanje sa strane, PL: 0%



Slika 5.11 Graf za gubitak posade, gađanje sa strane, PL: 10%



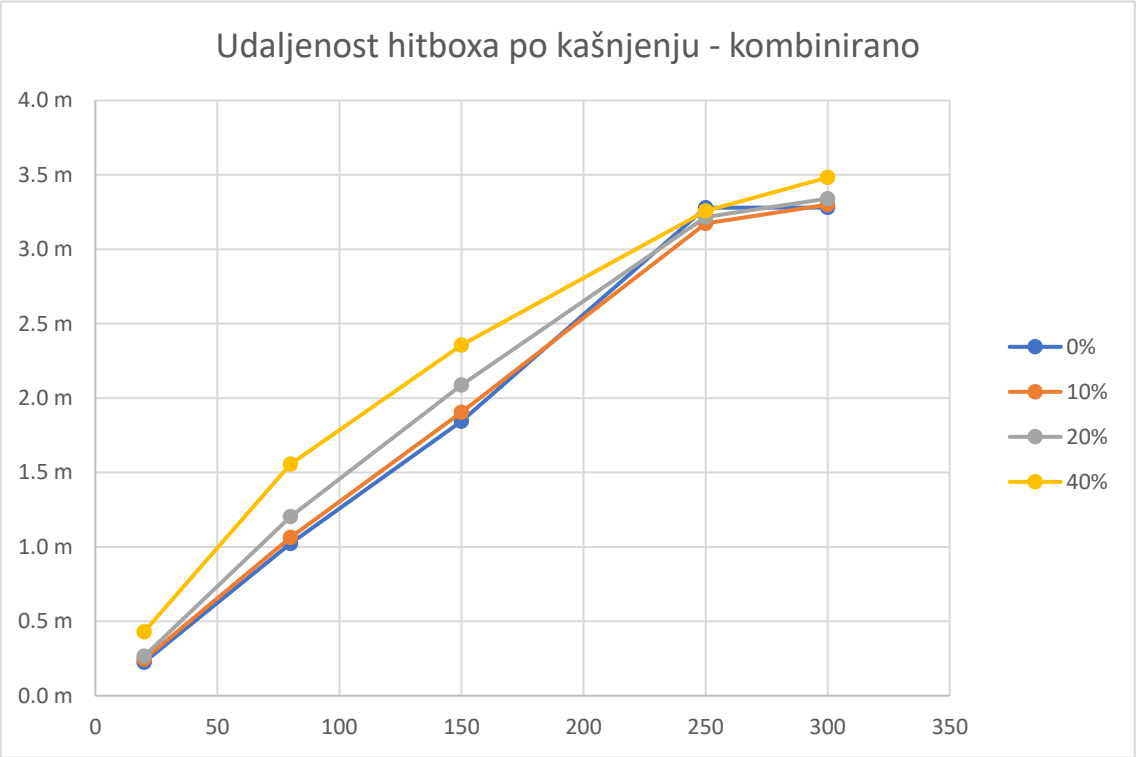
Slika 5.12 Graf za gubitak posade, gađanje sa strane, PL: 20%



Slika 5.13 Graf za gubitak posade, gađanje sa strane, PL: 40%

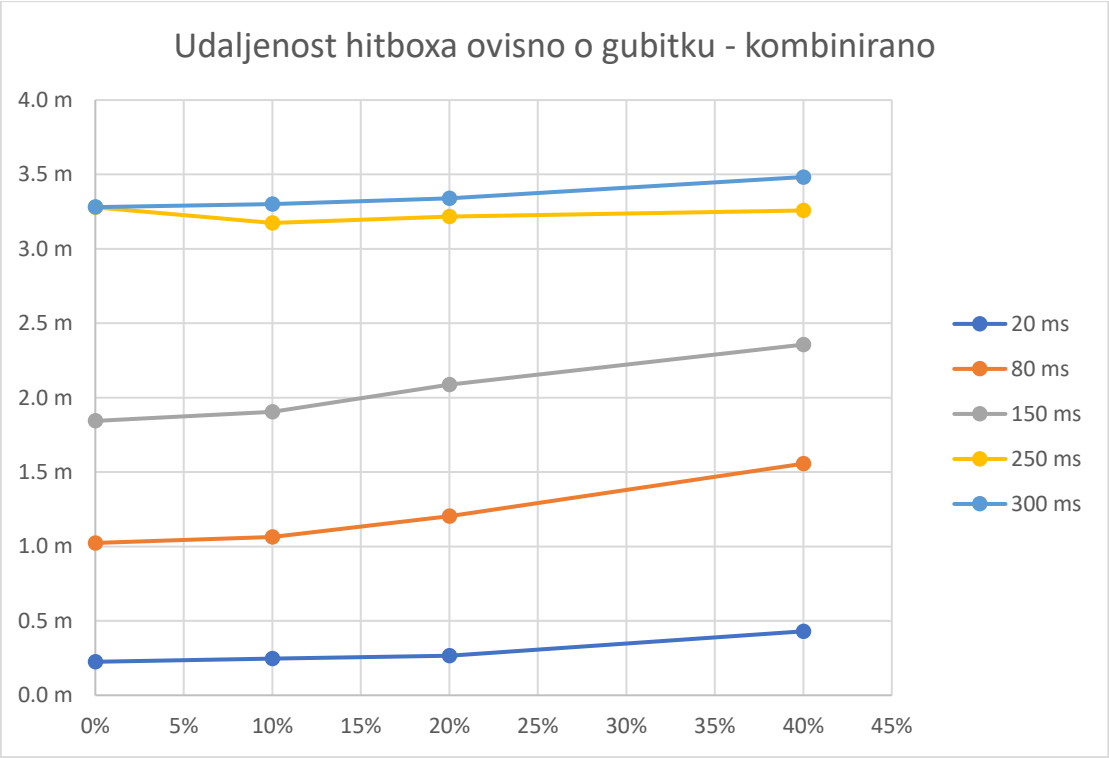
Iz prikazanih grafova se vidi da je gubitak posade znatno manji za kašnjenje od 300 ms nego u svim drugim slučajevima. U tom slučaju, iako je hitbox pogođen, točka gdje je on pogođen nije dobro kompenzirana, nego je „promašena“. Gubitak posade nije jednostavno izmjeriti, što proizlazi iz visokih odstupanja (standardne devijacije) svake točke u grafovima. Razlog tome je zato što on jako ovisi točki pogotka koja ovisno o vremenu reakcije (kada treba ispucati top – iskustvo testera) može biti jako dobra (3-4 članova posade je ubijeno) ili loša (0-1 članova posade ubijeno). Uz to, sama simulacija štete je (pomalo) ovisna o pseudo-nasumičnom generatoru brojeva (RNG) koji također može imati presudan utjecaj na napravljenu štetu.

Iz donjeg grafa (slika 5.14) se može vidjeti da povećavanje gubitka također povećava udaljenost hitboxa:



Slika 5.14 kombinirane udaljenosti po kašnjenju, gađanje sa strane

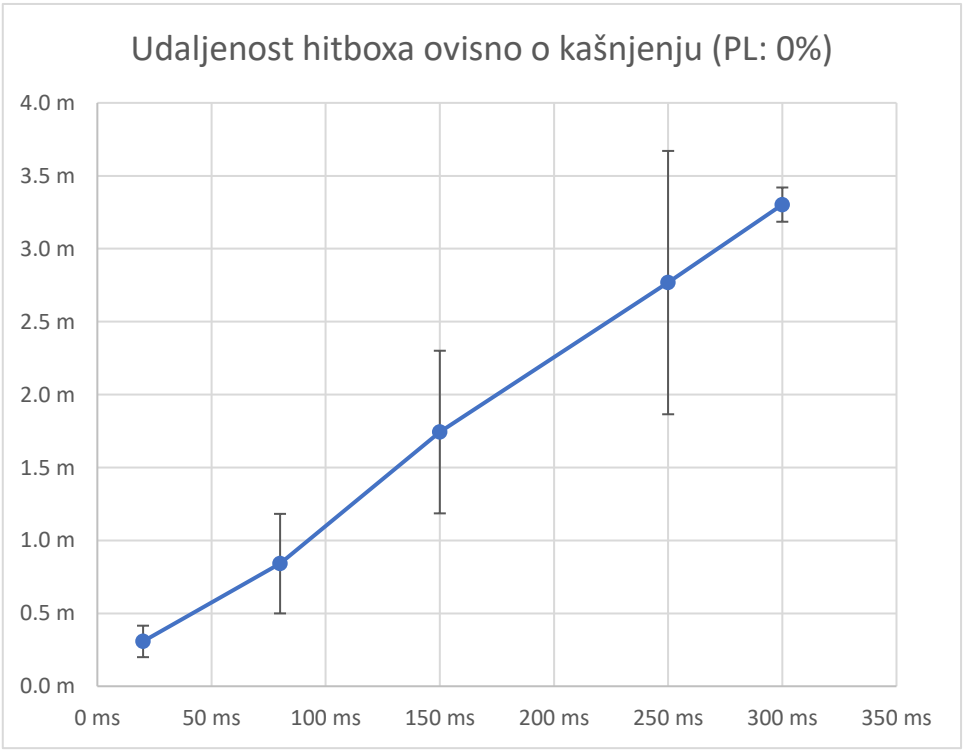
Sljedeći donji graf (slika 5.15) prikazuje odnos udaljenosti i gubitka za sva mjerena kašnjenja. Iz njega je lako uočiti da veći gubitak uzrokuje veću udaljenost.



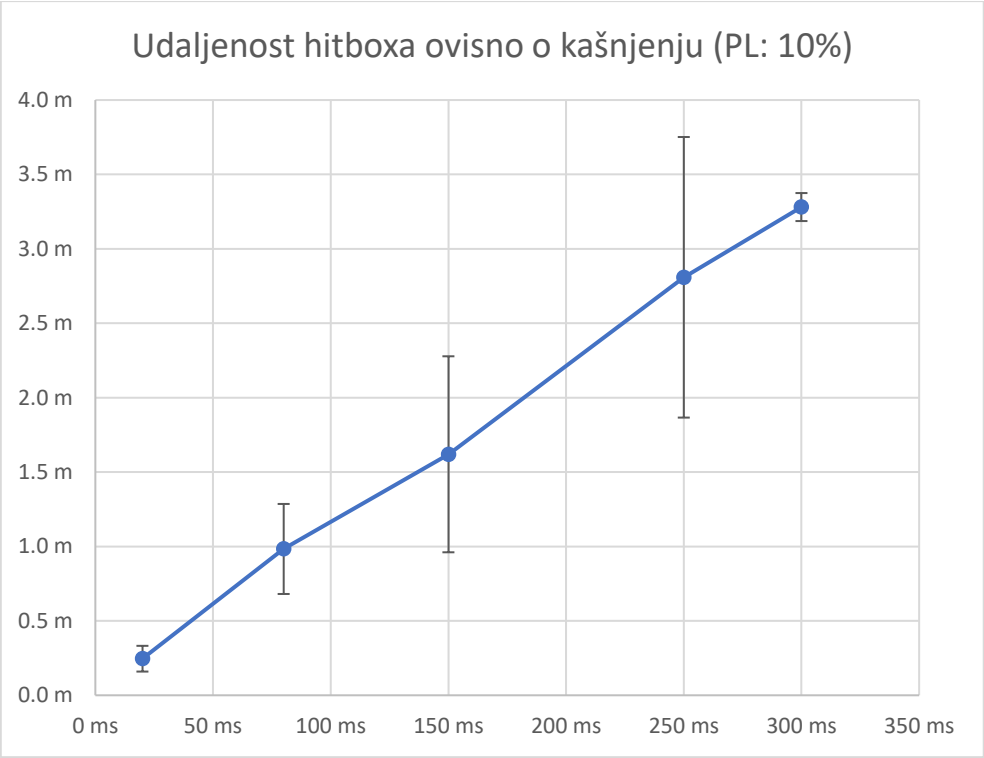
Slika 5.15 kombinirane udaljenosti ovisno o gubitku, gađanje sa strane

5.3.2. Gađanje sprijeda

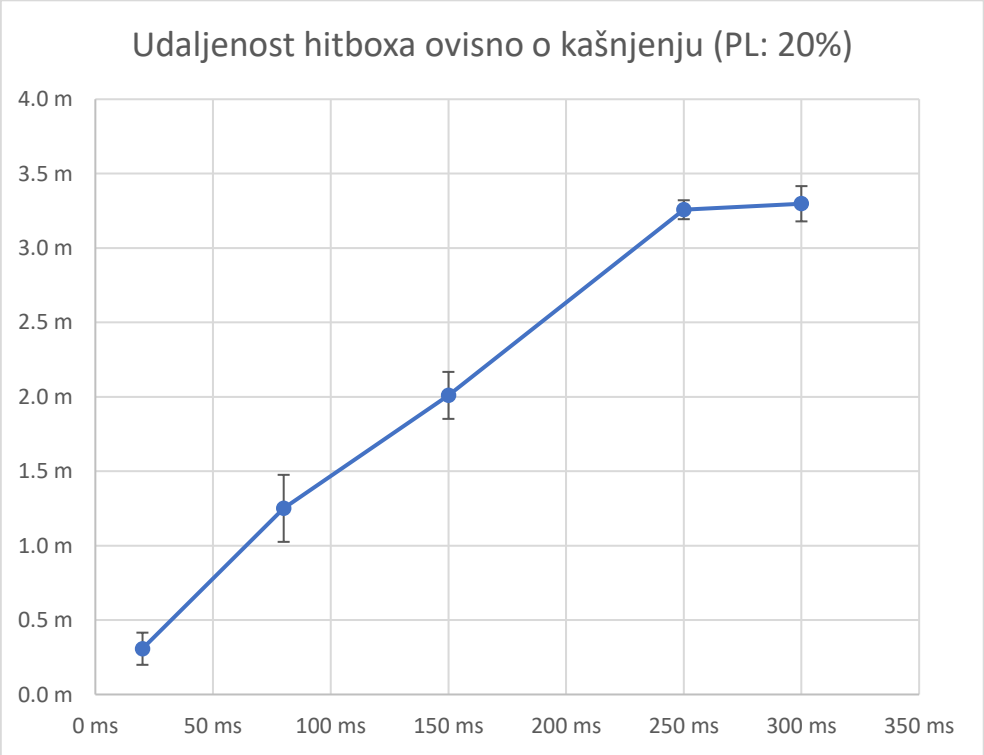
Za gađanje sprijeda, grafovi udaljenosti hitboxa su sljedeći:



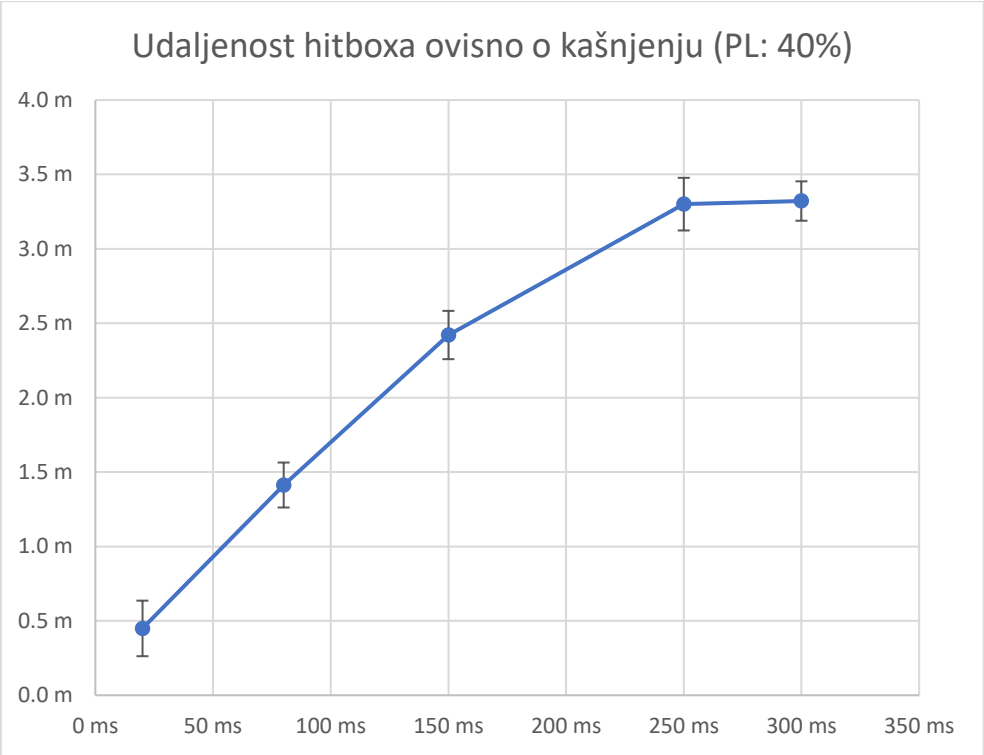
Slika 5.16 Graf za udaljenosti hitboxa, gađanje sprijeda, PL: 0%



Slika 5.17 Graf za udaljenosti hitboxa, gađanje sprijeda, PL: 10%



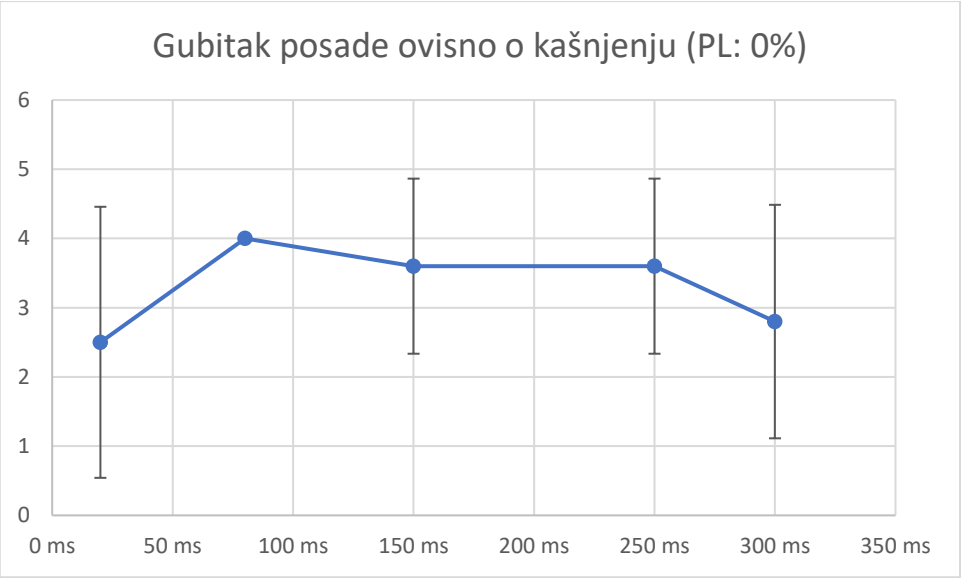
Slika 5.18 Graf za udaljenosti hitboxa, gađanje sprijeda, PL: 20%



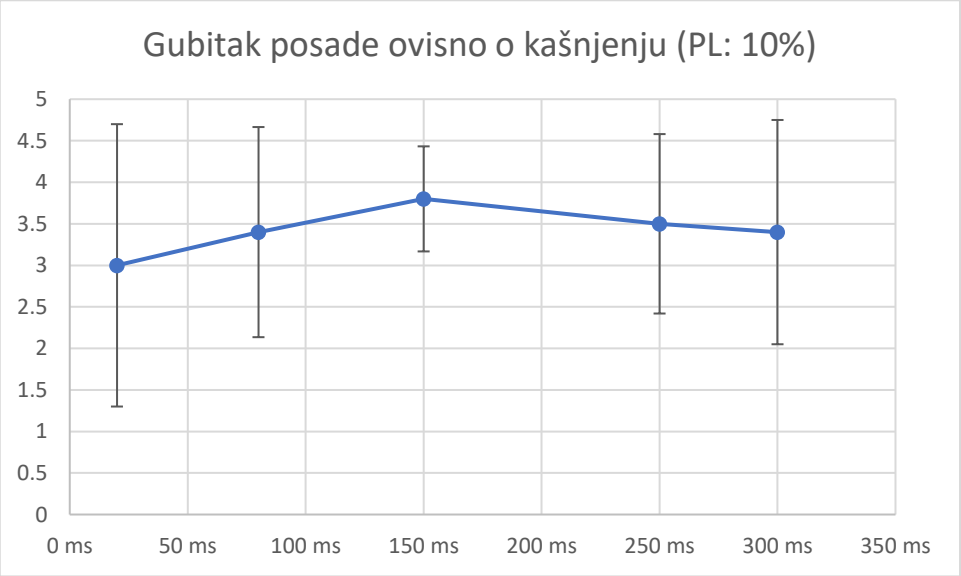
Slika 5.19 Graf za udaljenosti hitboxa, gađanje sprijeda, PL: 40%

Prikazani grafovi (slike 5.16, 5.17, 5.18 i 5.19) generalno pokazuju istu stvar kao i kod gađanja sa strane, no na prva dva grafa (za gubitke paketa od 0% i 10%, slike 5.16 i 5.17) se vide izrazito velika odstupanja za kašnjenje od 250 ms (izazvano outlier-ima u mjerenjima (male vrijednosti udaljenosti). Razlog događanja tih outliera nije lako utvrditi, no moguće je da su izazvani nestabilnošću simuliranja mrežnog kašnjenja tijekom testiranja), u druga dva grafa odstupanja su znatno manja i situacija je „vraćena u normalu“ tj. potvrđuje hipotezu da sustav radi dobro do otprilike 267 ms. Ovdje također nije bilo slučaja da tenk nije bio pogođen.

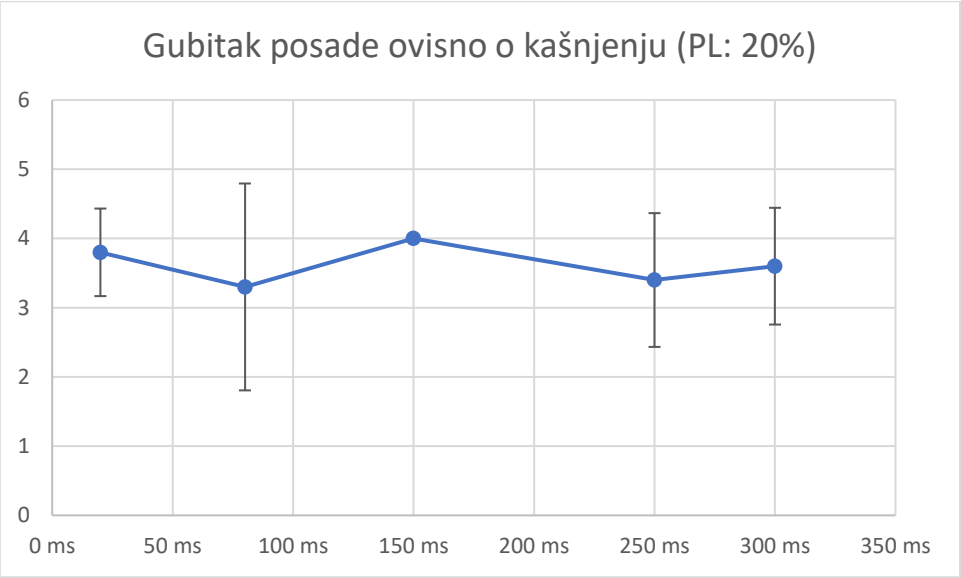
Gubitci posade ovisno o kašnjenju su sljedeći (slike 5.20, 5.21, 5.22 i 5.23):



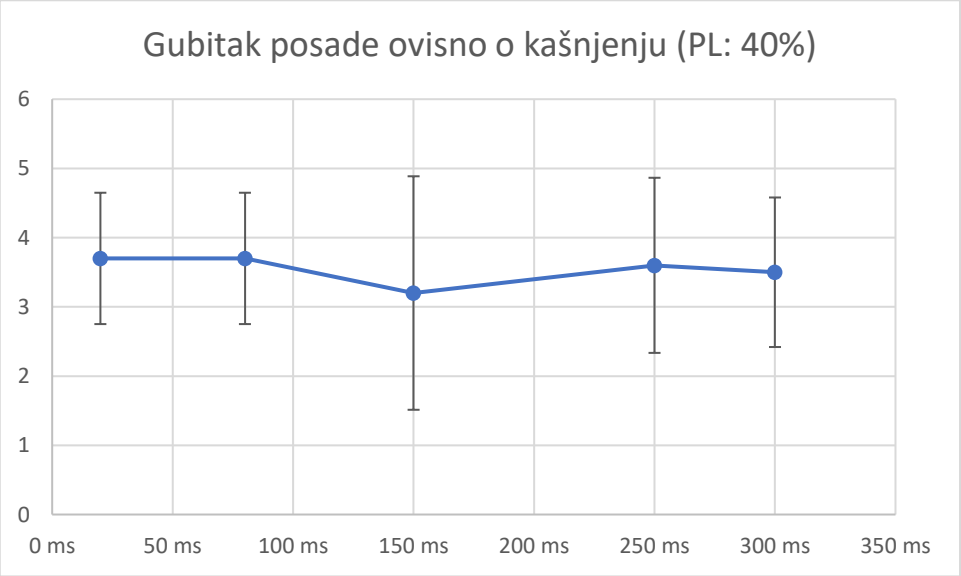
Slika 5.20 Graf za gubitak posade, gađanje sprijeda, PL: 0%



Slika 5.21 Graf za gubitak posade, gađanje sprijeda, PL: 10%



Slika 5.22 Graf za gubitak posade, gađanje sprijeda, PL: 20%

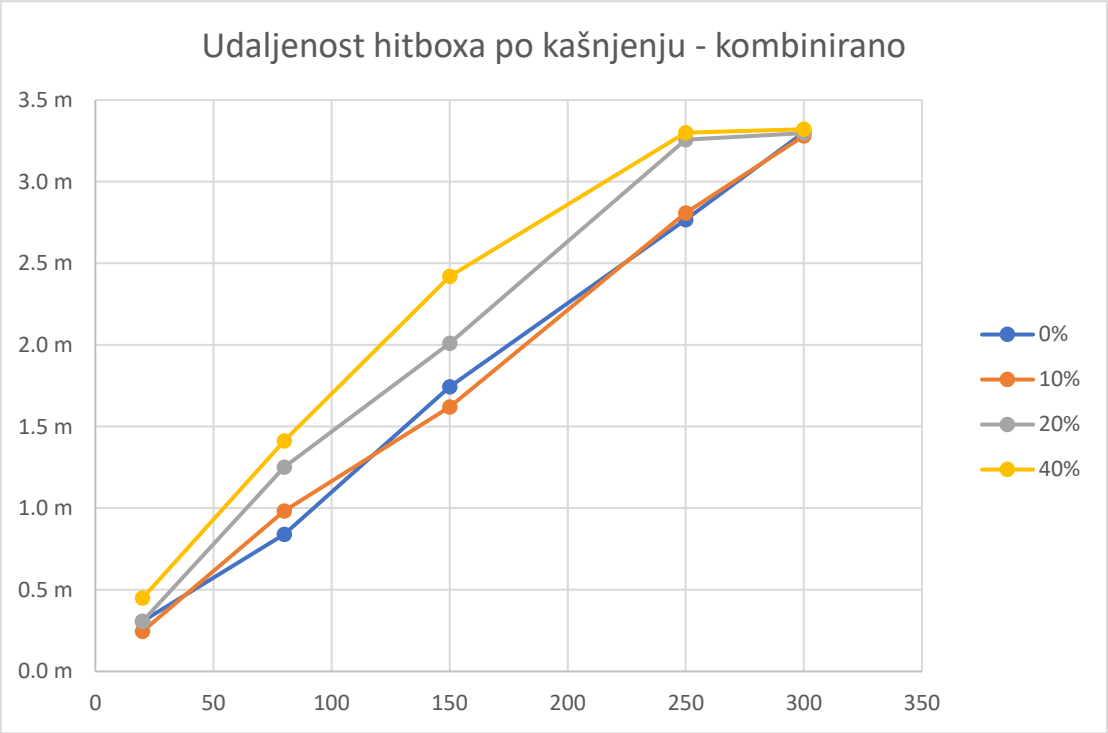


Slika 5.23 Graf za gubitak posade, gađanje sprijeda, PL: 40%

Na gornjim slikama (5.0, 5.21, 5.22 i 5.23) se vidi da je gubitak posade nekada „veći od 4“ tj. odstupanje pokazuje iznad 4. Razlog tomu je to što je za odstupanje korištena standardna devijacija koja je prikazana jednako (jednaka vrijednost) za gornju i donju granicu. Najveći mogući gubitak posade je zapravo 4. Gubitci posade su znatno veći i stabilniji kod gađanje sprijeda naspram gađanje s desna. To je i očekivano zato što je u testiranju plava kocka (na koju se pozicionira nišan) postavljena na idealno mjesto za uništavanje tenka – tanki nagnuti oklop ispred vozača u tenku, pošto je oklop tanak, granata će nakon proboja zadržati više energije pa time i odmah nakon proboja generirati više gelera što jednostavno znači više štete, često fatalne za svakog člana posade u tenku. Gađanje s desne (ili lijeve) strane je

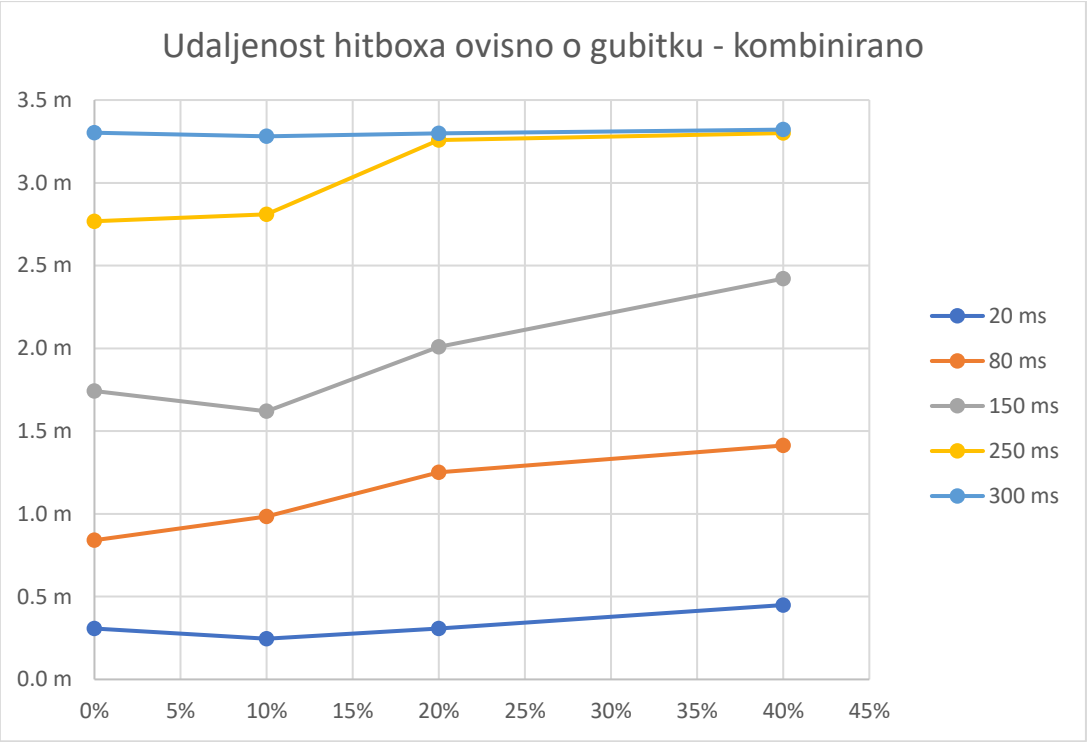
daleko manje optimalno (ako je cilj uništiti tenk). Iz danih grafova se također može opaziti da za veliko kašnjenje od 300 ms, gubitci posade nisu znatno manji, to je opet radi optimalnijeg smjera za gađanje i činjenice da je lakše pogoditi metu koja se približava napadaču, nego metu koja se od njega udaljava.

Donji graf (slika 5.24) pokazuje da udaljenost raste s kašnjenjem – kao i prije:



Slika 5.24 kombinirane udaljenosti hitboxa po kašnjenju, gađanje sprijeda

Potrebno je primijetiti da su udaljenosti za gubitke od 0% i 10% „isprepletene“, no to je zbog većih odstupanja njihovih vrijednosti. Sljedeći graf (slika 5.25) pokazuje ovisnost udaljenosti o gubitku – isti zaključak kao i kod gađanja s desne strane.



Slika 5.25 kombinirane udaljenosti hitboxa po gubitku, gađanje sprijeda

5.4. Zaključak analize

Sustav kompenzacije kašnjenja u igri se generalno dobro prilagođava za razne iznose kašnjenja, odnosno radi uspješno. Gađanje tenka sa strane daje slabije rezultate u smislu štete za veliko mrežno kašnjenje, dok je gađanje sprijeda otpornije na mrežno kašnjenje. Maksimalno mrežno kašnjenje do kojega sustav dobro funkcionira je do 250 ms što je dokazano podacima i teorijom (danim izračunom). Za kašnjenja izvan 250 ms je i dalje moguće pogoditi tenk, no mjesto pogotka nije dobro kompenzirano i s time šteta može značajno zakazati.

Zaključak

Rezultat ovog rada je (pokazna) umrežena igra „Tank Fight“ s implementacijom dviju tehnika kompenzacije kašnjenja, samostalno predviđanje i premotavanje vremena. U radu je analizirana tehnika premotavanja vremena tj. njena uspješnost s obzirom na razne iznose mrežnog kašnjenja i gubitka paketa. Rezultat analize je da tehnika uspješno radi za razne iznose mrežnog kašnjenja, no kvaliteta kompenzacije pada za visoke iznose kašnjenja i gubitka paketa.

Literatura

[1] Radian Simulations LCC, „Gunner, HEAT, PC!“
Poveznica: https://gunnerheatpc.com/ pristupljeno: 7.veljače 2024.

[2] Gaijin Entertainment, „War Thunder“
Poveznica: https://warthunder.com/en/ pristupljeno: 7. veljače 2024.

[3] Matijašević, Maja: prezentacija *Koncepti umrežavanja* u sklopu predmeta „Umrežene igre“, Sveučilište u Zagrebu, 2022./2023.

[4] Jim Kurose , Keith Ross. *Computer Networking: A Top Down Approach* , 8. izdanje, Pearson, 2021

[5] Andrew Tanenbaum , Nick Feamster, David Wetherall *Computer Networks* , 6. izdanje , Pearson, 2021

[6] Berson, A. (1996). *Client/server architecture*. McGraw-Hill, Inc..

[7] R. Schollmeier, "A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications," *Proceedings First International Conference on Peer-to-Peer Computing*, Linköping, Sweden, 2001, pp. 101-102, doi: 10.1109/P2P.2001.990434.

[8] darvishdarab *Client-Server Architecture* poveznica: https://darvishdarab.github.io/cs421_f20/docs/readings/client_server/ pristupljeno 8.veljače 2024.

[9] Kumar, Amlendra, *Socket programming in C using TCP/IP*, (Lipanj, 2017.). Poveznica: https://aticleworld.com/socket-programming-in-c-using-tcpip/ pristupljeno 10. siječnja 2024.

[10] Limi Kalita, *Socket Programming* , 5. izdanje. 2014. ISSN:0975-9646

[11] Liu, Shengmei & Claypool, Mark. (2023). The Impact of Latency on Target Selection in First-Person Shooter Games. 51-61. 10.1145/3587819.3590977.

[12] R. Jota J. Deber, C. Forlines, and D. Wigdor. 2015. How much faster is fast enough? User perception of latency and latency improvements in direct and indirect touch. In Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI’15).

[13] I. Scott MacKenzie and Colin Ware. 1993. Lag as a determinant of human performance in interactive systems. In Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI’93). 6.

[14] Liu, Shengmei, XiaokunXu, and Mark Claypool. "A Survey and Taxonomy of Latency Compensation Techniques for Network Computer Games." *ACMComputing Surveys (CSUR)* (2022). Web poveznica: https://web.cs.wpi.edu/~claypool/papers/lag-taxonomy/LatencyCompensation.html pristupljeno: 7.veljače 2024.

[15] Zander, Sebastian & Armitage, Grenville. (2004). *Empirically measuring the QoS sensitivity of interactive online game players*.

[16] M. Ahmed, S. Reno, M. R. Rahman and S. H. Rifat, "Analysis of Netcode, Latency, and Packet-loss in Online Multiplayer Games," *2022 International Conference on Augmented Intelligence and Sustainable Systems (ICAISS)*, Trichy, India, 2022, pp. 1198-1202, doi: 10.1109/ICAISS55157.2022.10010926.

[17] Sužnjević, Mirko: prezentacija *Umrežena simulacija* u sklopu predmeta „Umrežene igre“ , Sveučilište u Zagrebu, 2022./2023.

[18] K. -T. Chen, P. Huang and C. -L. Lei, "Effect of Network Quality on Player Departure Behavior in Online Games," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 5, pp. 593-606, May 2009, doi: 10.1109/TPDS.2008.148.

[19] Andrew S. Glassner, *An introduction to ray tracing*, 1989. ISBN 0-12-286160-4

[20] Quax, Peter & Monsieurs, Patrick & Lamotte, Wim & De Vleeschauwer, Danny & Degrande, Natalie. (2004). *Objective and subjective evaluation of the influence of small amounts of delay and jitter on a recent first person shooter game*. 152-156. 10.1145/1016540.1016557.

[21] Pingman Tools LCC, *What's "normal" for latency and packet loss?* (Siječanj, 2019.)

[22] Photon Fusion online manual: https://doc.photonengine.com/fusion/current/fusion-intro pristupljeno: 18. siječnja 2024

[23] Photon Fusion, *Projectiles*: https://doc.photonengine.com/fusion/v1/technical-samples/fusion-projectiles-essentials pristupljeno: 7.veljače 2024.

[24] Photon Fusion, *Prediction*: https://doc.photonengine.com/fusion/v1/tutorials/host-mode-basics/3-prediction pristupljeno 7.veljače 2024.

[25] Photon Fusion, *Lag Compensation*: https://doc.photonengine.com/fusion/current/manual/advanced/lag-compensation pristupljeno 7.veljače 2024.

[26] Clumsy (program), verzija 0.3: https://github.com/jagt/clumsy/releases/tag/0.3, pristupljeno: 2. siječnja 2024

[27] Photon Fusion, Simulating Network Conditions, povezinca: https://doc.photonengine.com/fusion/current/manual/testing-and-tooling/simulating-network-conditions pristupljeno .7.veljače 2024.

Sažetak

Cilj ovog diplomskog rada je bio realizirati umreženu pokaznu igru s tehnikama kompenzacije kašnjenja. Uz realizaciju tih tehnika, odrađena je i analize uspješnosti rada određene tehnike s obzirom na razne mrežne parametre u kojoj je pokazano da tehnika radi uspješno.

Ključne riječi:

digitalna igra; umrežena igra; razvoj igara; mrežno kašnjenje; kompenzacija kašnjenja; Unity; Photon Fusion;

Summary

The goal of this thesis was to realize a networked demo game with lag compensation techniques. In addition to the implementation of these techniques, an analysis of the performance of a certain technique regarding various network parameters was also carried out, in which it was shown that the technique works successfully.

Keywords:

digital game; networked game; game development; latency; lag compensation; Unity; Photon Fusion;

Skraćenice

RTT	*Round Trip Time*	vrijeme ukupnog putovanja
PL	*Packet Loss*	gubitak paketa
FFA	*Free for all*	slobodno za sve, tip igre
LAN	*Local Area Network*	lokalna mreža
WAN	*Wide Area Network*	mreža na širem području
TCP	*Transmission Control Protocol*	komunikacijski protokol
UDP	*User Datagram Protocol*	komunikacijski protokol
IP	*Internet Protocol*	komunikacijski protokol
NAT	*Network Address Translation*	mapiranje IP adresa
URP	*Universal Render Pipeline*	cjevovod za prikazivanje
UI	*User Interface*	korisničko sučelje
HUD	*Heads-Up Display*	posebni UI tijekom igranja
FPS	*Frames Per Second*	broj sličica u sekundi
CPU	*Central Processing Unit*	procesor računala
GPU	*Graphics Processing Unit*	grafička kartica
RAM	*Random Access Memory*	radna memorija računala
SSD	*Solid State Drive*	sekundarna memorija računala
ID	*Identification*	identifikator
DGLC	*Driver Gunner Loader Commander*	članovi posade (debug tekst)
APFSDS	*Armour Piercing Fin-Stabilized Discarding Sabot*	vrsta tenkovske granate