

УНИВЕРЗИТЕТ У БЕОГРАДУ
ЕЛЕКТРОТЕХНИЧКИ ФАКУЛТЕТ



**ИМПЛЕМЕНТАЦИЈА МРЕЖНЕ ВИШЕКОРИСНИЧКЕ
ВИДЕО ИГРЕ БОРБЕ РОБОТА
у УНРЕАЛ РАДНОМ ОКВИРУ**

Дипломски рад

Ментор:
проф. др Марко Мишић

Кандидат:
Никола Илић 2017/640

Београд, септембар 2024.

САДРЖАЈ

САДРЖАЈ	2
1. УВОД	4
2. РАЗВОЈ ВИДЕО ИГАРА	5
2.1. ДИЗАЈН И ПРАВИЛА ИГРЕ	6
2.2. МОДЕЛИ	6
2.3. АНИМАЦИЈЕ И ВИЗУЕЛНИ ЕФЕКТИ	6
2.4. ФИЗИКА ВОЗИЛА	7
2.5. УМРЕЖАВАЊЕ	8
2.6. ГРАФИЧКИ КОРИСНИЧКИ ИНТЕРФЕЈС	9
3. ОПИС КОРИШЋЕНИХ ТЕХНОЛОГИЈА	10
3.1. <i>UNREAL ENGINE</i>	10
3.2. <i>C++</i> И <i>BLUEPRINT</i> СИСТЕМ	12
3.3. <i>THE UNREAL EDITOR</i>	13
3.4. <i>CHAOS VEHICLE PLUGIN</i>	13
3.5. <i>NIAGARA VFX</i>	13
3.6. <i>ENHANCED INPUT</i>	14
3.7. <i>MICROSOFT VISUAL STUDIO</i>	14
3.8. <i>BLENDER</i>	15
3.9. УПРАВЉАЊЕ ИЗВОРНИМ КОДОМ	16
4. РЕАЛИЗАЦИЈА СИСТЕМА	17
4.1. СТРУКТУРА ФАЈЛОВА	17
4.1.1. Организација пројектних фајлова	17
4.1.2. Конфигурациони фајлови	19
4.1.3. Опис фолдера и фајлова	19
4.2. АРХИТЕКТУРА ЦЕЛОКУПНОГ СИСТЕМА	20
4.2.1. Стандардизована архитектура	20
4.2.2. Конкретна архитектура пројектног система	21
4.3. ИЗРАДА 3Д МОДЕЛА РОБОТА	23

4.3.1.	Моделовање.....	23
4.3.2.	Дефинисање текстура модела.....	23
4.3.3.	Припрема модела за анимирање.....	24
4.4.	УВОЗ МОДЕЛА У <i>UNREAL ENGINE</i> ОКРУЖЕЊЕ И КОНФИГУРАЦИЈА ВОЗИЛА.....	25
4.5.	ДЕФИНИСАЊЕ ПРОЦЕСА ИГРАЊА.....	26
4.5.1.	Механика робота	26
4.5.2.	Правила игре.....	30
4.6.	ИЗРАДА МАПЕ И ТЕРЕНА ЗА ИГРУ	30
4.7.	АРХИТЕКТУРА ПОДСИСТЕМА УМРЕЖАВАЊА	31
4.7.1.	Постојећа подршка за умрежавање	31
4.7.2.	Имплементирано решење умрежавања.....	33
4.8.	КОРИСНИЧКИ ИНТЕРФЕЈС	35
5.	ОПИС РАДА СИСТЕМА	37
6.	ЗАКЉУЧАК	40
	ЛИТЕРАТУРА.....	41
	СПИСАК СЛИКА.....	42
	СПИСАК ТАБЕЛА	43
	СПИСАК БЛОКОВА КОДА.....	44

1. УВОД

Видео игре представљају јединствену комбинацију софистиране технологије и уметности. Креатор има неограничене могућности да створи виртуелни свет по својој жељи. Да би остварили нашу визију видео игре, пролазимо кроз процес испуњен изазовима, открићима, иновацијама и експериментисањем. За играче, играње видео игре представља интерактивни бег од реалности, где на тренутак постајемо супер хероји, роботи будућности или средњовековни витезови. Кроз наше одлуке и интеракције обликујемо наше јединствено путовање кроз виртуелни свет.

Овај пројекат представља истраживање свих аспеката израде једне вишекорисничке 3Д видео игре у алату *Unreal Engine*, са нагласком на само умрежавање играча. Тематика видео игре је међусобна борба робота на точковима, што је и популарна дисциплина у стварности. Играчи по свом умећу и стилу игре бирају једног од три типа робота. Борба се дешава у малој арени са другим играчима у реалном времену и захтева стратегију, умеће и брзо доношење одлука. Циљ је потпуно функционална игра где корисник бира робота и уз једноставан интерфејс се конектује у сесију са другим играчима и отпочиње борбу. *Unreal Engine* је изабран као радни оквир због своје изванредне подршке за реализацију 3Д игре: једноставни увоз модела у игру, подршка за анимацију модела, имплементирана физика за возила и колизију, једноставна израда терена и високо апстрактни систем за умрежавање играча. Захваљујући систему за умрежавање, преко позива удаљених процедура (eng. *Remote Procedure Calls - RPCs*), мрежно реплицираних својстава и лаког управљања сесијама, резултат је међусобна синхронизација између различитих играча уз минимално време кашњења.

У другом поглављу изнет је увод у развој и данашњи положај радних оквира за израду видео игара, као и описи проблема са прегледом постојећих решења по развојним целинама. Треће поглавље садржи кратку историју радног оквира, као и опис свих технологија, програма и алата коришћених при изради пројекта. У четвртном поглављу је детаљно презентована реализација система и покривени су сви кључни проблеми као и имплементирана решења. Пето поглавље описује рад система и карактеристике игре, односно представља корисничко упутство. Последње поглавље, шесто, даје рекапитулацију пројекта, и отвара дискусију о потенцијалним унапређењима и надоградњи игре.

2. РАЗВОЈ ВИДЕО ИГАРА

Развој видео игара данас укључује неколико различитих професија: софтверски и хардверски инжењеринг, дизајн ентеријера и екстеријера, аналитику, сликарство, вајарство, компоновање и многе друге. Током дугог низа година, развијани су посебни алати за ефикаснију израду видео игара, радни оквири (енг. *game engines*). Више није потребно бавити се проблемима ниског нивоа програмирања, као што су руковање графичком картицом и исцртавање игре на екрану, оптимизовање извршног кода на вишеничним системима, руковање прекидима. Радни оквири за израду видео игара функционишу на начин да садрже основу коју треба изменити и употпунити, стварајући тако конкретну видео игру. Садрже висок ниво апстракције, и посао програмера је углавном искључиво дефинисање механике и правила игре. Високобудетне компаније имају своје приватне, прилагођене радне оквири, који представљају веома вредну, строго заштићену имовину. У последњих неколико година, дешава се постепени заокрет, где све више великих компанија почиње да користи јавно доступне радне оквири. Данас многи јавни радни оквири за израду видео игара својим могућностима надмашују корпоративна решења.

Unreal радни оквир је савремени и свестрани алат за израду видео игара који корисницима омогућује потпуни и независни развој видео игре у свим доменима. Представља снажну базу за креирање високо-оптимизованих, комплексних игара и данас је стандард у индустрији развоја видео игара. Његова имплементација садржи подршку за све уобичајене захтеве садашњих видео игара, као што су 3Д и 2Д графичко рендеровање, симулација физике, систем за израду корисничког интерфејса, аудио систем, подршка за умрежавање више играча, обрада улаза од стране играча и остало. Данас *Unreal Engine* са својим широким спектром могућности није само софтвер за програмере, већ за и све остале учеснике у процесу креирања видео игара: дизајнере, скулпторе, цртаче, архитекте, аниматоре, дизајнере звука, продуcente, аналитичаре. Сам алат, његова историја, спецификације и могућности су детаљно описани у поглављу 3.1.

2.1. Дизајн и правила игре

Како би заинтересовала и задржала играче, видео игра мора да има довољно балансирана правила и параметре. Потребан је циљ, као и јасне повратне информације колико смо му близу. Кључна механика јесте борба роботима, хаотична, брза, непредвидива, динамична. Играчи очекују фер и балансирано искуство, где сваки тип робота има подједнаке шансе за победу, без очигледних фаворита. Стил игре, умеће и креативност требају бити једини параметри који предвиђају победу.

Подесиви параметри су брзина, максимални животни поени, маневарске способности, ефикасност оружја. Временом, искуством и тестирањем, потребно је свакодневно подешавати ове параметре како би достигли балансирану игру. Такође је подједнако важно редовно ажурирати игру, убацивањем нових типова робота као и нових мапа. Циљ видео игре је забава. То је уједно коначни изазов при развијању игре, учинити је забавном за остале играче.

2.2. Модели

У игри са малом и једноставном мапом, са конкретним и очекиваним акцијама и строго дефинисаним правилима, главни утисак на играча оставља модел свог виртуелног карактера. Супротно од естетике модела, потребно је водити рачуна и о његовој комплексности. Модел робота мора бити оптимизован јер ће сваки клијент у сваком тренутку приказивати и моделе свих осталих играча у сесији. Такође различити типови карактера морају бити и довољно визуелно различити, како би играч једноставно могао да препозна типове противника у жару борбе. Модел мора поштовати и логичке карактеристике карактера коме припада, и њихове различите улоге.

2.3. Анимације и визуелни ефекти

Заједно са моделима, анимације и визуелни ефекти представљају један од најупечатљивих детаља у игри. Играчи очекују реалистичне анимације у односу на извршене акције. Управљање возилом треба се јасно уочити на самом моделу возила. Убрзавање, кочење, и бирање смера треба рефлектовати на тачкове возила, као и на само возило по закону инерције. Главна одлика која разликује различите типова робота јесте њихово оружје, због чега је јако битно исправно анимирати покрете активирања оружја. Све споменуте анимације потребно је извршити у реалном времену како би корисник имао потпуни доживљај контроле робота. У окружењу непрестане борбе битно је да анимације различитих типова оружја буду јасно

видљиве и довољно међусобно различите како би играч имао шансе да избегне нападе. *Unreal Engine* садржи добру подршку за анимирање динамичких модела, где се модел дели на ситније логичке целине које је могуће засебно манипулисати.

Поред анимација, још један начин како играчи перципирају догађаје у игри су специјални ефекти, који се активирају као последица неког догађаја од изузетног значаја. У овом пројекту визуелни ефекти су последица успешног напада једног робота на други. Неправилно руковање специјалним ефектима један су од најчешћих узрока проблема оптимизације. Смањење њиховог броја је данас често решење када желимо да остваримо мање искоришћење ресурса рачунара.

2.4. Физика возила

Играчи очекују да поред исправног визуелног покрета точкова и оружја, ти покрети јасно осликавају и физичку реалност у игри. У већини случајева, заправо је физичка логика та која управља анимацијама. Главних изазови при развијању возила су како остварити исправне односе возила према различитом терену, препрекама, брзини, углу точкова итд. Захваљујући одређеним додацима, у алату *Unreal Engine* се већина физичке логике конфигурише параметрима. Битнији параметри возила које је могуће подесити су сам облик физичког тела (које је различито од визуелног модела), конфигурација точкова, трења, проклизавања, мотора, као и произвољно тежиште и маса објекта.

Посебан проблем представља дефинисање облика физичког тела, које осим што утиче на остале физичке појаве попут инерције, има и улогу детектовања колизије. Детектовање колизије представља и срж саме дефиниције игре, јер је круцијална за исправно функционисање целокупне игре. У брзој игри борбе робота, у малој арени, колизије се дешавају често и хаотично. Игнорисањем или погрешним детектовањем колизије долазимо до ситуација проласка робота једне кроз друге као и погрешног аплицирања штете од оружја.



Слика 1.1. Визуелизација колизионе мреже свих карактера и окружења

2.5. Умрежавање

У зависности од игре, архитектура умрежавања се разликује, и на програмеру је да користећи *Unreal Engine* програмерски интерфејс (енг. *application programming interface* - *API*) прилагоди мрежни систем својим потребама. Кориснички захтеви везани за умрежавање су конкретно: могућност играча да првенствено врши претрагу постојећих сесија и конектује се на одабрану (уколико нема постојећих сесија креира нову и преузима улогу сервера), и одржавање синхронизације у реалном времену о својствима и акцијама робота. Иако *Unreal Engine* садржи архитектуру умрежавања, то не значи да је умрежавање клијентских процеса брз и једноставан начин без додатних конфигурација и алгоритама. Процеси нижег нивоа, као што су креирање и одржавање клијентских конекција, слање пакета, серијализација, рутирање, балансирање опретећења (енг. *load balancing*), већ су имплементирани на нивоу радног оквира, те клијент, односно програмер, креира логику умрежавања на нивоу веће апстракције[1]. Резултат овог пројекта јесте и мрежна архитектура која може служити као основа за друге игре сличних карактеристика.

2.6. Графички кориснички интерфејс

Информације и стање игре које играчи не могу закључити из анимација и ефеката представља се путем корисничког интерфејса. У видео играма постоје два основна типа корисничких интерфејса са другачијим захтевима, *HUD (Head-Up Display)* и менији.

HUD представља све текстуалне и 2Д информације приказане играчу током 3Д симулације игре. Главни захтеви су једноставност, лака уочљивост и разазналост података. Потребан је приказ свих битних информација без затрпавања екрана сувишним. У вишекорисничкој игри често подразумева и приказивање информација о осталим играчима у реалном времену. *HUD* треба да има све потребне податке, а уједно и да не омета и одузима фокус играчима од саме игре.

Менији представљају засебни, независнији део видео игре и прате их друга правила и проблеми. Осим што служе као извор информација, такође омогућују и конкретне акције. Главни захтев је интуитивност. Играч треба да без пуно размишљања и са лакоћом дође до жељене акције. Код конкретне видео игре, мени се приказује тик пред улазак у игру, и служи да корисник пре конекције на сервер изабере жељеног карактера и започне игру. Један од главних изазова у пројекту јесте испунити споменуте захтеве за решавање проблема одабира карактера.

3. ОПИС КОРИШЋЕНИХ ТЕХНОЛОГИЈА

При изради пројекта коришћени су савремени алати који се данас углавном користе при изради игара. Главни софтвер коришћен за израду пројекта је сам радни оквир, *Unreal Engine*, као и његови уграђени алати и проширења. Описани су и алати за израду модела, писање и руковање кодом.

3.1. *Unreal Engine*

Unreal Engine је платформа и радни оквир за развој видео игара направљена и и даље одржавана од стране компаније *Epic Games*. Први пут је објављен 1998. године као део видео игре *Unreal*. Иницијално је направљен за игре пуцања оружјем из првог лица, а временом се проширио и на све остале жанрове 3Д видео игара. Тренутно активно издање је *Unreal Engine 5*, који је објављен у Априлу 2022. године, као дуго очекивано издање које је увело мноштво иновативних решења, као и подршку за конзоле нових генерација и модерни хардвер. Главне нове технологије су *Nanite* – виртуелни геометријски систем који користи нове формате за графичко представљање да прикаже високодетаљне структуре на сценама са пуно објеката[2], *Lumen* – динамички глобални систем осветљења и рефлексije који је дизајниран за модерне компјутерске системе[3], *World Partition* систем – подела великих светова на мрежу мањих ћелија које се аутоматски учитавају и бришу из меморије по потреби [4]. Верзија коришћена у овом пројекту је *Unreal Engine 5.4*, што представља последње стабилно издање.



Слика 2.1.1. Исечак из игре *Unreal* објављене 1998. године, прве игре израђане на *Unreal* радном оквиру



Слика 3.1.2. Исечак из игре *Black Myth: Wukong* објављене 2024. године, израђене на *Unreal* радном оквиру

Велика предност алата и један од главних разлога његове популарности је што је софтвер отвореног кода. Целокупан код је доступан јавности и свако може радити на њему и предлагати промене и унапређења. Захваљујући томе окупио је велики број самосталних, хоби програмера који учествују у његовом развоју. Отвореним кодом такође се остварују неограничене могућности надоградње софтвера, према потребама различитих видео игара. Лиценса за коришћење софтвера је таква да је омогућено коришћење алата чак и у комерцијалне сврхе без икаквих трошкова, све до бруто зараде од 1,000,000\$, од када смо дужни да платимо 5% од зараде компанији *Epic Games*[5].

Unreal Engine је изабран јер представља врхунац развоја радних оквира за креирање игара као и својих савремених могућности. Сам отворени код омогућује дубље разумевање целокупне архитектуре и пружа неограничене могућности даљег развијања. Такође постоји велика заједница ентузијаста и врло брзо се могу наћи одговори на сва потенцијална питања и проблеме. У јулу 2014. године, *Unreal Engine* је проглашен за најуспешнији радни оквир за израду видео игара од стране Гинисове књиге рекорда[6].

3.2. C++ и *Blueprint* систем

Понашање игре у окружењу *Unreal Engine* се дефинише на два начина. Радни оквир написан је у програмском језику C++, што је уједно и скриптни језик у систему. То омогућује високе перформансе, флексибилност и потпуну контролу над ресурсима. Будући да у игри желимо максималне перформансе и најоптималнију искоришћеност компјутерских ресурса, као и максимално прецизна извршавања, C++ представља идеално решење. Уз све предности програмског језика ниског нивоа, блиског хардверу, C++ је уједно и објектно оријентисани језик са високим нивоом апстракције. Захваљујући тој комбинацији, омогућује брз и ефикасан, прецизан код, уз познате, стандардизоване и модерне технике писања кода. Још једна модерна опција у данашњој верзији *Unreal Engine* јесте уживо кодирање (енг. *live coding*). То је процес превођења и уметања у извршни фајл нових и измењених динамичких библиотека док је *Unreal Engine* процес покренут. Још су у ранијим верзијама постојале сличне алатке под називом *Hot Reload*, међутим уз разне грешке, слабе перформансе и прљање других зависних објеката, није био подразумевани алат радног оквира и корисници су га избежавали. Од верзије 4.22 уведен је нови, стандардизовани систем под називом *Live Coding* који у великој већини случајева ради исправно и знатно скраћује време програмирања, јер није потребно угасити и поново покренути процес. Тиме избегавамо поновно учитавање свих зависних фајлова, база података и конфигурација[7].

Други начин за дефинисање логике је *Blueprint* (срп. нацрт, шема, план) систем. Представља високофункционални визуелни скриптни систем који омогућује и програмерима и не-програмерима да дефинишу логику извршавања брзо, ефикасно и флексибилно, креирајући графове извршавања. Чак и поред *Live Coding* решења, прототипирање, тестирање и експериментисање у *Blueprint* систему је и даље много брже него у C++. Мане овог система су перформансе: *Blueprint* логика се преводи на C++ међукод, те се додаје још један ниво апстракције, што утиче на перформансе, поготово код високо комплексних система са пуно гранања. То доводи и до дужег времена превођења кода. Такође има лошу скалабилност: како се пројекат развија, *Blueprint* логика постаје комплекснија и конфузнија, граф постаје тежак за разумевање, и проналажење грешака и дебаговање постаје све комплексније до тренутка када је практично немогуће.

За квалитетан, ефикасан, одржив и скалабилан пројекат видео игре, потребно је користити и C++ и *Blueprint* систем, заједно их комбиновати и искористити најбоље из обе

опције. Једноставно правило којим се може водити јесте да комплексни логички код који утиче директно на игру треба писати директно у C++, док остале једноставније ствари које су подложније променама треба имплементирати као *Blueprint* граф, углавном активирање и манипулација визуелних компоненти.

3.3. *The Unreal Editor*

Технички, *Unreal Engine* представља програмску основу – базу, коју надограђујући претварамо у нашу видео игру. Уређивач (енг. *editor*) представља алатку за само надограђивање и манипулисање те базе. Садржи мноштво конкретнијих алатки које нам помажу у томе. Покреће се заједно са игром и током развоја игре служи као главни софтвер. Крајњем кориснику, играчу, доставља се игра без уређивача, као извршни фајл, у зависности од платформе. У овом документу, *Unreal Engine* као надоградиви софтвер над којим је уграђена наша логика која представља саму игру, и *Unreal Engine* као алатка, су изједначени и мисли се на обе ствари, јер заједно чине нераздвојиви *Unreal Engine* екосистем.

3.4. *Chaos Vehicle Plugin*

Chaos (срп. хаос) је назив за подразумевани систем физике и разарања окружења развијен од стране компаније *Epic Games*. Могуће је једноставно у своју игру убацити и проширење овог система - посебни модул за подршку за возила. Он омогућује кориснику развијања реалистичних возила, уз конфигурацију свих стандардних својстава за возила као у реалности: амортизера, диференцијала, мењача, мотора, квачила. Могуће је подесити по својој жељи и аеродинамику, криву убрзања, број обртаја. Детаљније, нуди и засебну конфигурацију точкова: ширину и пречник, масу, трење, окретност, као и понашање у случају активирања кочнице и ручне кочнице. Веома је детаљно и реално осмишљен да постоји и опција додавања система против блокирања точкова (АБС). Због својих широких могућности *Chaos Vehicle Plugin* тренутно нема алтернативу при развоју возила у *Unreal Engine* радном оквиру и представља очигледни избор.

3.5. *Niagara VFX*

Нијагара систем је основни и стандардизовани алат за симулацију визуелних ефеката унутар *Unreal Engine* платформе. Садржи уграђену подршку за најкоришћеније ефекте попут ватре, дима, експлозија, варница, разних ефеката воде. Уз систем за визуелно скриптовање сличан *Blueprint* систему, омогућује потпуну контролу над најмањим елементима ефекта,

честицама. У конкретном пројекту није било потребе за пуно различитих ефеката. Главни ефекат представља варничење при успешном нападу, и то је омогућено уз мало труда и максималне могућности.

3.6. *Enhanced Input*

Један од новитета алата *Unreal Engine* од верзије 5 јесте побољшано руковање улаза играча. За разлику од старог система који је зависио од фиксно уграђеног мапирања улаза и акција, нови систем представља флексибилније и више одрживо решење. Такође поједностављује могућност играча да кроз игру мења на који конкретан унос се активира која акција.

3.7. *Microsoft Visual Studio*

Подразумевани алат за писање кода за *Unreal Engine* је *Microsoft Visual Studio*. Могуће је користити и друга популарна развојна окружења попут *JetBrains Rider*, али је потребно направити неколико додатних корака. *Visual Studio* можемо покренути директно из уређивача, док је *Unreal Engine* активан, а можемо и самостално покренути програм и као резултат превођења добити нашу игру у *Unreal Engine* окружењу. У пројектном фолдеру генерисаном од стране радног оквира, добијамо и скрипте за генерисање *Visual Studio* пројекта, па се коришћење овог програма само намеће. Инсталација свих потребних модула за исправно функционисање у развојном окружењу *Visual Studio* је праволинијска и своди се на покретање посебног програма *Visual Studio Installer*. Под посебном категоријом развоја игара, једноставним штиклирањем можемо увести и инсталирати све неопходне компоненте. Док код осталих развојних окружења би овај посао морали радити ручно, често уз много проблема док покушавамо из конзолног излаза да сазнамо зашто се пројекат не преводи. *Visual Studio* садржи и јако богат избор различитих проширења (енг. *extensions*) која знатно олакшавају свакодневно програмирање. Уз инсталирање *Unreal Engine* компонената добијамо и потребна проширења за истицање синтаксе, аутоматско допуњавање кода и предлоге кода за *Unreal Engine* специфичне кључне речи.

Једно од кориснијих проширења током израде пројекта било је *VSChromium*, који садржи колекцију алатки за измену кода и навигацију кроз код. Конкретно, коришћена је његова претрага кода, која је неколико пута бржа и прецизнија од подразумеване. Добро скалира са величином пројекта, што је у овом случају неколико десетина хиљада фајлове (углавном *Unreal*

Engine отворени код) и може приказати резултате претраге за мање од 0.1 секунде за 100,000+ фајлова[8].

Дебаговање *Unreal Engine* програма у алату *Visual Studio* нуди разноврсне опције и информације, омогућавајући програмеру да брзо идентификује и уклони проблеме у коду. Развојно окружење садржи сталне и условне тачке прекида (енг. *breakpoints*), могућност прекида програма у било ком тренутку, приступ вредностима променљивих и меморији, као и стеку позива функција. Такође, конзолни излаз је повезан са *Unreal Engine* излазом и у реалном времену нам исписује информације, упозорења и грешке. Све наведене могућности представљају све што је потребно за развој и одржавање квалитетног кода без грешака. За израду пројекта коришћена је последња верзија, *Microsoft Visual Studio Enterprise 2022*.

3.8. Blender

Unreal Engine поседује и своје уграђене алате за креирање модела. Ови алати су данас доста ограничени и не пружају довољну слободу и могућности за потребе већине игара. Углавном служе за прототипирање и израду једноставних статичких модела. За комплексније, динамичке моделе који подржавају анимације, користе се засебни програми. У великим професионалним студијама користе се углавном плаћени софтвери попут алата *Maya*, *3D Max*, *ZBrush* који омогућавају високе перформансе и широки спектар опција. *Blender* је програм отвореног кода који је ривал многим другим плаћеним решењима у свету 3Д моделинга. За потребе израде модела робота коришћен је програм *Blender 4.1*. У њему су урађени сви аспекти визуелне репрезентације робота: модел, текстура и припремање модела за потребе анимација. Сам модел је урађен у стилу малог броја полигона, као скуп тачака, ивица и површина. Текстура је дефинисана у алату у оквиру *Blender* алатке за директно бојење модела, која после извози текстуру као 2Д слику. Припремање модела за потребе анимације представља одвајање логичних целина модела у посебне скупове зване кости (енг. *bones*) и конфигурисање њихове међусобне зависности. Ове целине представљају скуп полигона који се могу заједно, независно или зависно анимирати. Углавном се мења њихова позиција у свету, ротација, величина.

3.9. Управљање изворним кодом

Ради лакшег развоја и праћења промена и напретка, коришћен је гит систем за управљање изворним кодом. У конкретном пројекту занемарена је главна одлика гит система – колаборација више људи при изради софтверских решења, већ је гит коришћен за логичко раздвајање различитих временских периода у развоју софтвера. Удаљени репозиторијум на платформи *GitHub* направљен је за осигурање пројекта од могућих хаварија на локалном рачунару. Као кориснички интерфејс за управљање гит системом изабран је програм *GitKraken*, који садржи добру визуелизацију свих стања на локалном и удаљеном директоријуму, као и преглед свих могућих гит операција.

4. РЕАЛИЗАЦИЈА СИСТЕМА

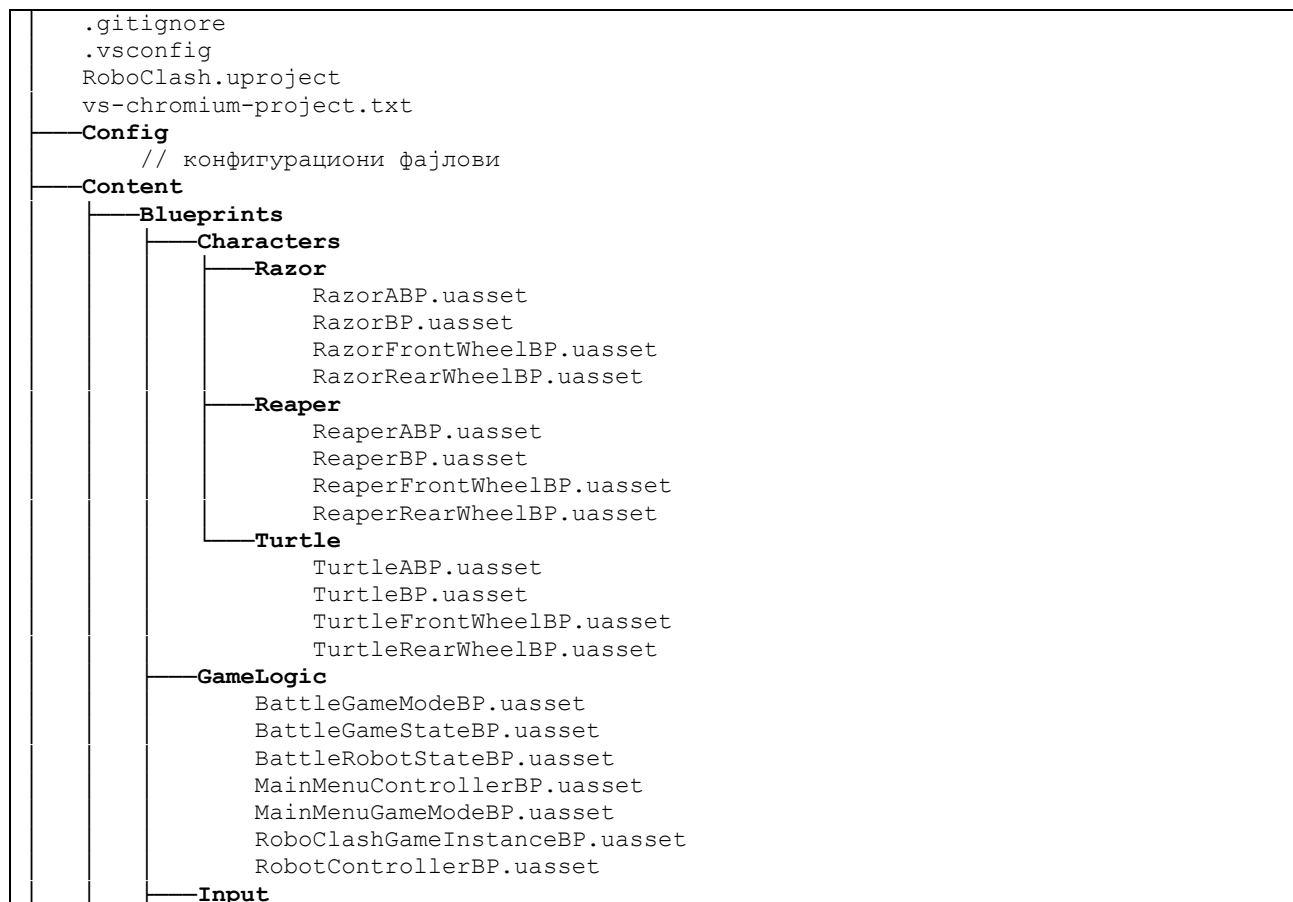
Реализација система прати архитектуру *Unreal Engine* окружења. Основне класе које представљају стандардизоване логичке целине у систему су проширене и њихове уграђене међусобне везе су прилагођене нашим потребама.

4.1. Структура фајлова

Коришћена је подразумевана структура пројектих фајлова. У кореном директоријуму фајлови су подељени на изворни код и придружене фајлове чији се садржај користи у току игре и динамички учитава у процес. Даље су фајлови разврстани по изабраним логичким целинама.

4.1.1. Организација пројектних фајлова

У наредном прилогу дато је стабло свих корисних фајлова. Привремени фолдери и фајлови су изостављени јер не утичу на дефиницију игре, већ представљају аутоматски генерисане фајлове од стране радног оквира, процеса превођења и саме видео игре.





Блок кода 4.1.1.1. Стабло структуре фолдера и фајлова

4.1.2. Конфигурациони фајлови

Фајл који дефинише који остали ће се чувати од стране гит система је **.gitignore**. Садржи регуларне изразе помоћу којих се дефинишу сви изузимајући фајлови. Конкретно, то су помоћни фајлови који се генеришу при превођењу програма, извршни и остали бинарни фајлови, пројектни фајлови које генеришу радно окружење и радни оквир и остали кеширани фајлови. Генерисан је помоћу алата *gitignore.io*[9].

У конфигурационом фајлу **.vsconfig** налази се списак компонената које радно окружење *Microsoft Visual Studio* треба да активира при отварању софтверског пројекта. Слично томе, **RoboClash.uproject** говори алату *Unreal Engine* који пројекат, верзију и модуле да учита у меморију при покретању. Фајл **vs-chromium-project.txt** конфигурише екстензију *VSChromium*, којој одређује путем регуларних израза које фајлове да индексира и укључи у своју претрагу.

У конфигурационом фолдеру налази се неколико аутоматски генерисаних *.ini* фајлова. Служе за чување опција измењених у *Unreal Engine* окружењу. **DefaultEngine.ini** чува подешавања видео и аудио система, мапа, комуникације са оперативним системом и хардвером. **DefaultInput.ini** садржи дефиниције регистрација улаза и њихову осетљивост.

4.1.3. Опис фолдера и фајлова

Сами изворни фајлови подељени су у два подразумевана фолдера, *Content* и *Source*. Унутар *Unreal Engine* радног оквира појављује се и трећи фолдер у коме се налази изворни код самог радног оквира.

Content садржи све придружене фајлове који се користе у игри и које је потребно учитати у програм током извршавања. Углавном су у питању бинарни фајлови са посебном *.uasset* екстензијом. Овај фолдер даље је подељен на типове фајлова који се у њима налазе. *Blueprint* фајлови (графови извршавања) који су и главне логичке целине, налазе се у посебном, истоименом фолдеру. Унутар, све *Blueprint* класе се разврставају по конкретнијим типовима, и то класе за: карактере (типове робота - возила), сам механизам игре, обраду корисничког улаза, кориснички интерфејс, визуелне ефекте. Унутар *Content* фолдера налази се и посебан фолдер који садржи све мапе у игри. Пројекат тренутно садржи две мапе, главну, у којој се дешава сама борба робота, и мапу која се налази на почетном екрану и служи за одабир карактера и улазак у борбу. Сви материјали коришћени у игри се такође налазе у посебном фолдеру. Материјали представљају дефиницију завршног изгледа неке површине, садржи текстуре са додатним математичким операцијама о томе како њен изглед зависи од светлости. Осим засебног

материјала за сваког типа робота, постоји и материјал који је коришћен за бојење мапе. Логичка целина 3Д модела има посебан фолдер под називом *Meshes* (срп. мрежа). Добили су такав назив су јер сваки 3Д модел представља скуп међусобно повезаних тачака, ивица и површина – мрежу. При увозу сваког динамичког 3Д модела у *Unreal Engine* радни оквир, добијамо три различите мреже, три различита фајла. Прва представља визуелну репрезентацију – скелетна мрежа. Други тип фајла назива се скелет, и садржи информације о логичким целинама које се могу засебно анимирати – костима. Трећи фајл је физичко тело модела, и одређује како ће се модел понашати при симулацији физике. Последњи фолдер унутар *Content* фолдера садржи текстуре, 2Д слике које као основа комплекснијих структура материјала дефинишу изглед површина у игри.

Source фолдер има нешто једноставнију структуру. Садржи изворни програмски код написан у C++ програмском језику. Заглавља и изворни C++ фајлови су подељени у засебне фолдере. Унутар, подељени су у две целине, код за механизам и правила игре (енг. *gameplay*) и код за кориснички интерфејс (енг. *user interface* - *UI*).

4.2. Архитектура целокупног система

Изграђена архитектура система проширује стандардизовану архитектуру за *Unreal Engine* пројекте. Ова архитектура садржи већ дефинисане класе за различите потребе које је потребно наследити и прилагодити својим потребама.

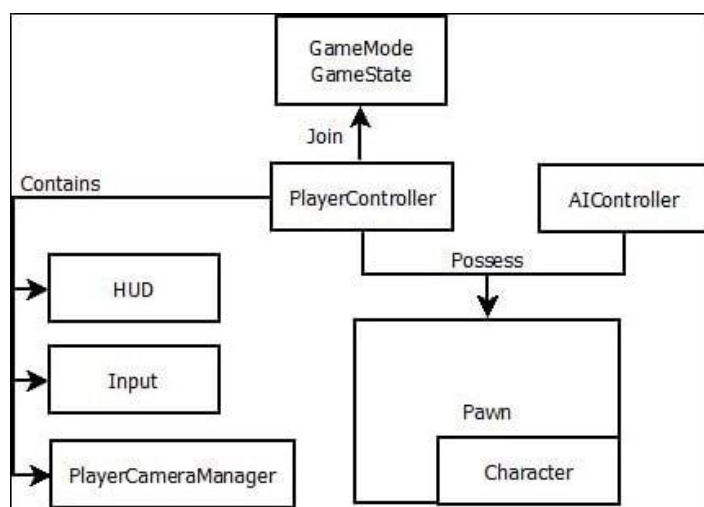
4.2.1. Стандардизована архитектура

Главна базна класа која одређује правила и механизме игре јесте *GameMode*, односно *GameModeBase*. У њој се налазе дефиниције за поставку и ток игре. Везана је за тренутно учитану мапу и има глобални приступ те се може сматрати имплементацијом пројектног узорка уникат (енг. *singleton*). Ради лакше манипулације подацима у току игре и лакше синхронизације тих података са клијентима у међукорисничкој игри, ова класа садржи посебну класу *GameState*. Директно је контролише и дефинише. Класа *GameState* садржи информације о тренутним дешавањима и стању у игри и прослеђује ове податке свим играчима тако да садрже исправне податке у сваком тренутку.

Класе специфичне за сваког појединачног играча у међукорисничкој игри су *PlayerController* и *PlayerState*. Имају сличан однос као *GameMode* и *GameState*, али не на нивоу целокупне игре, већ на нивоу играча. Контролер играча је представља „мозак“ контролисаног

ентитета, односно садржи претварање улаза корисника у корисне акције над ентитетом. Осим играчког контролера, у систему постоји и засебна класа задужена за аутоматско контролисање ентитета од стране рачунара, односно вештачке интелигенције. Такође сваки засебни контролер садржи и објекат класе *PlayerState* у који смешта све тренутне информације о играчу. Ове информације се у вишекорисничкој игри прослеђују и свим осталим играчима, и углавном садрже информације о играчима које би остали играчи требали да знају: име играча, освојени бодови, статистика.

Класа *Pawn* (срп. пион, пешак у шаху) представља ентитет који може бити контролисан, односно запоседнут (енг. *possessed*) од стране контролера. Углавном има комплексну логику јер садржи дефиниције понашања ентитета и интеракције са остатком света. Класа *GameInstance* представља објекат глобалног опсега унутар програмског кода, који постоји локално над покренутом игром. Животни век му је од покретања до гашења процеса игре, и постоји кроз различите мапе и модове. Служи за чување података који требају бити доступни у било ком тренутку, у било којој мапи. Елементи графичког корисничког интерфејса наслеђени су од класе *UserWidget*.



Слика 3.2.1.1. Поједностављена стандардизована архитектура *Unreal Engine* пројекта [10]

4.2.2. Конкретна архитектура пројектног система

Процес програмирања у окружењу укључује комбинацију *Blueprint* и *C++* класа, односно *Blueprint* класе треба да буду изведене од *C++* родитељске класе. Дата је табела логичких целина, *Blueprint* класа, назив родитељске *C++* класе и кратак опис њихових садржаја. Колона Тип представља родитељску *Unreal Engine* класу. *Blueprint* класе се по конвенцији завршавају BP (скраћено од *Blueprint*) суфиксом. У коду, класе имају префикс у

зависности од родитељске класе. Префикс *A* означава да је класа изведена у неком тренутку од класе *Actor*, што представља објекат који може постојати у игри, у мапи. Префикс *U* означава наслеђивање од класе *UObject*, што је најосновнија класа у *Unreal Engine* систему.

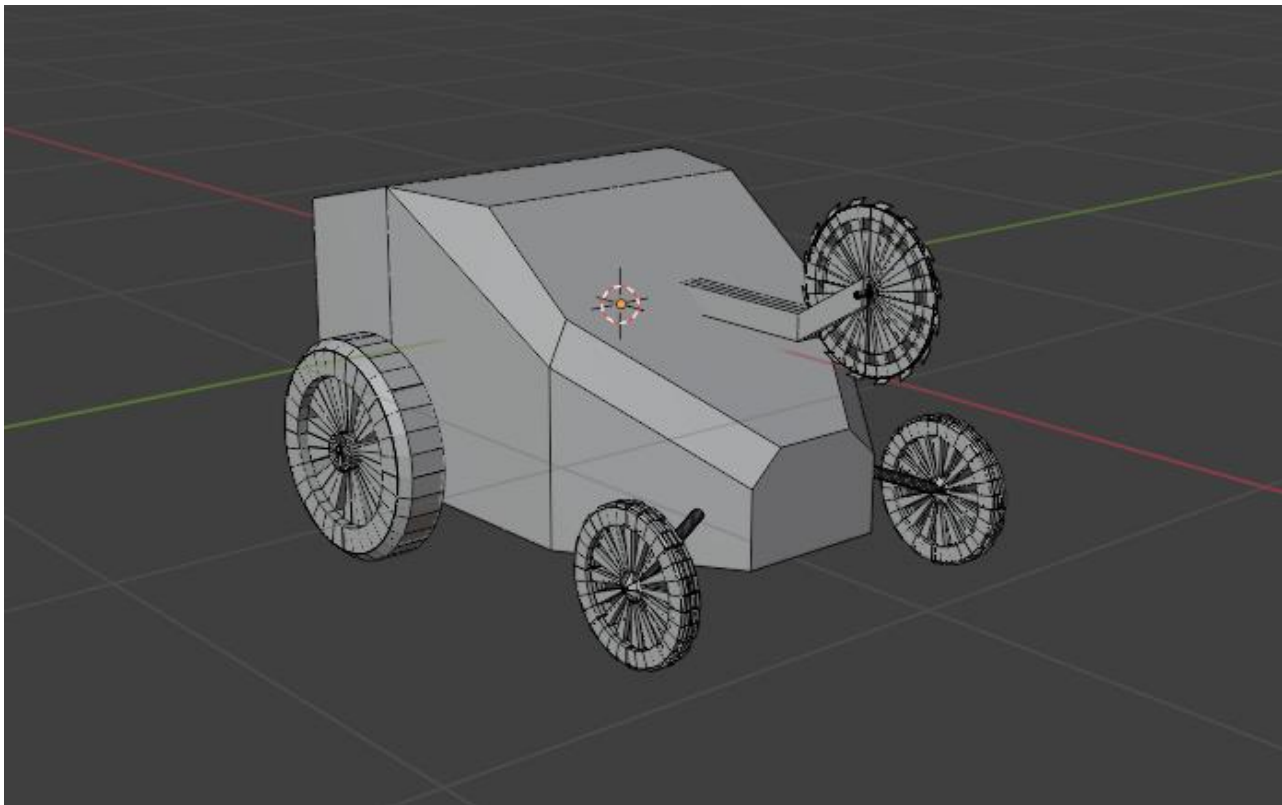
Табела 4.2.2.1. Табела логичких целина пројекта и њихов опис

Blueprint класа (C++ класа)	Тип	Садржај класе
BattleGameModeBP (ABattleGameMode)	GameMode	Главна класа која контролише главну мапу и дефинише борбу возила. Увезује остале класе. Рукује приступом нових играча сесији.
BattleGameStateBP (AGameState)	GameState	Увезује остале класе. У пројекту, подаци се синхронизују путем PlayerState класе.
BattleRobotStateBP (ABattleRobotState)	PlayerState	Садржи основне податке о играчима током игре који треба да се синхронизују. Као и методе за њихову синхронизацију између сервера и клијената.
RobotControllerBP (ARobotController)	PlayerController	Омогућује иницијалну комуникацију између клијента и сервера. Креира HUD кориснички интерфејс и рукује његовим ажурирањем. Дефинише мапирање корисничког улаза и конкретних акција над заспоседнутим пионом.
RoboClashGameInstance (URoboClashGameInstance)	GameInstance	Служи за чување података између мапа, конкретно при одабиру карактера у главном менију.
MainMenuGameModeBP (AGameModeBase)	GameMode	Главна класа која контролише мапу почетног менија. Иницијализује мапу и кориснички интерфејс.
MainMenuControllerBP (APlayerController)	PlayerController	Рукује логиком одабира карактера са почетне мапе путем миша.
RazorBP (ARazorPawn)	Pawn	Механика и анимације типа робота.
ReaperBP (AReaperPawn)	Pawn	Механика и анимације типа робота.
TurtleBP (AReaperPawn)	Pawn	Механика и анимације типа робота.
MainMenuBP (UUserWidget)	UserWidget	Визуелни изглед и логика главног менија. Чување одабира типа робота. Претрага и креирање нове сесије.
HealthBarBP (UUserWidget)	UserWidget	Визуелни изглед и ажурирање преосталих животних поена робота.
OverlayBP (UBattleOverlay)	UserWidget	Визуелни изглед <i>HUD</i> интерфејса који исписује поене свих играча у сесији.
LeaderboardEntryBP (ULeaderboardEntry)	UserWidget	Визуелни изглед појединачног улаза у табели поена. Логика за испис имена, увећавања поена и посебног обележавања улаза за локалног играча.

4.3. Израда 3Д модела робота

4.3.1. Моделовање

Сви 3Д модели робота имплементирани су техником малог број полигона (енг. *low poly*). Уместо комплексних алатки за вајање, модели се обликују комбиновањем једноставних елемената: коцки, ваљака, лопти, сфера. Даље, директним манипулацијама тачака, ивица и површина модела добијамо коначан производ.



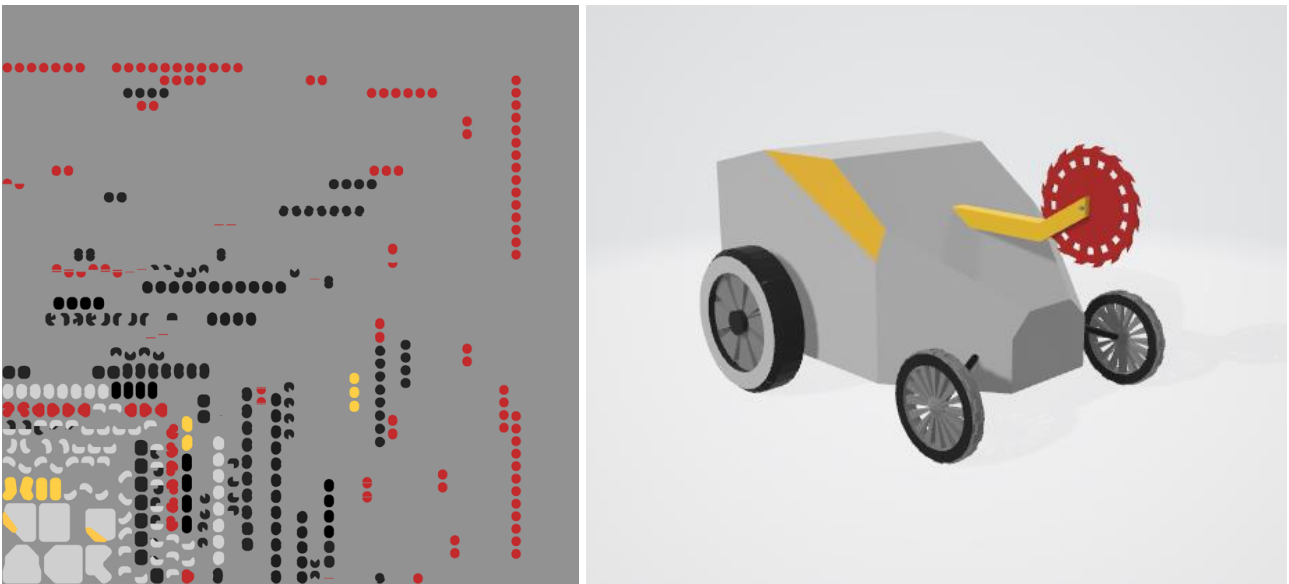
Слика 4.3.1.1. Изглед једног од модела у алату *Blender* након процеса моделовања

4.3.2. Дефинисање текстура модела

Пре него што кренемо са дефинисањем текстуре бојењем модела, потребно је да 3Д модел отпакујемо (енг. *unwrapping*) у 2Д приказ. Тај процес назива се и УВ мапирање. Термин „УВ“ означава називе координата у 2Д пројекцији, У за хоризонталну, и В за вертикалну осу. Пошто је излазни фајл текстуре дводимензионална слика, потребно је одредити који делови те слике одговарају којој површини 3Д модела. УВ мапирањем остварујемо управо то, мапирање различитих делова модела на различита места у текстури. Алат *Blender* садржи неколико различитих алата за отпакивање модела, са циљем да се оствари што прецизније мапирање, са мање празног простора на 2Д репрезентацији и самим тим мањим димензијама излазне слике

текстуре. Проблем који се овде јављао јесте велики број површина на точковима и округлим елементима модела. Да би се веродостојно симулирале заобљене ивице, потребно је користити већи број полигона, како се не би приметио угао између површина. Међутим, већи број површина загушује УВ мапу, и текстура постаје неверодостојна, где се боје премазују између површина. Овај проблем је решив на два начина. Смањити број површина, или повећати резолуцију текстуре. Текстура су резолуције 2048x2048, те би њихово повећање на 4К резолуцију било превише при учитавању у меморију, и имало би утицаја на перформансе игре. Такође би остављало доста празног простора за овако једноставне моделе. Решење је пронађено у смањењу броја полигона на точковима, тако да буду и даље веродостојно представљени као заобљена тела.

Само бојење и дефинисање текстура је након тога једноставно. Алат *Blender* има могућност директног бојења површина модела, који се после аутоматски пресликава на 2Д слику, чиме добијамо текстуру.



Слика 4.3.2.1. и слика 4.3.2.2. Текстура модела и финални изглед модела са текстуром

4.3.3. Припрема модела за анимирање

Да би анимирали неки 3Д модел, прво треба да дефинишемо које целине у моделу се могу засебно померати. У конкретном пројекту са роботима возилима, то су точкови, који се могу ротирати око своје осе, и оружје, чији покрети активирања зависе од типа возила. У алату *Blender* можемо дефинисати ове целине, које се називају кости (енг. *bones*) а скуп свих костију и њихове међусобне релације се назива арматура. Потребно је дефинисати све површине које

припадају једној кости и дати јој назив. Конкретно, четири кости за сваки точак и једна за оружје. Потребно је и исте исправно позиционирати, тачније почетну тачку костију, како би ротација око своје осе била веродостојна.

За исправно функционирање треба дефинисати и корену (енг. *root*) кост, која је родитељ свим осталим. Налази се на врху хијерархије и представља референтну тачку за све остале кости.

4.4. Увоз модела у *Unreal Engine* окружење и конфигурација возила

Увоз 3Д модела направљеног у *Blender* алату у *Unreal Engine* радни оквир је праволинијски процес. Потребно је сачувати модел у неком од стандардних формата, конкретно је коришћен формат *FBX* и кликом на *Import* дугме увести га у *Unreal Engine* пројекат. Као резултат за сваки модел добијамо неколико фајлова описаних у одељку 4.1.3.

Најважнији и једини фајл где је потребно урадити додатни посао је физичко тело, које одређује понашање објекта при симулацији физичких процеса. Пре свега представља колизиону мрежу, односно физичке границе чврстог тела. *Unreal Engine* садржи основне примитиве за дефинисање физичког тела, као што су квадар, сфера, капсула. Њиховим комбиновањем дефинишемо границе физичког тела. Постоји и опција аутоматски генерисаног тела у односу на визуелни модел, и у неколико случајева то решење је био довољно, али у већини доста непрецизно. Овај процес био је доста комплексан и осетљив јер је сваким, чак и ситним променама, физика тела била знатно измењена – инерција, убрзање, проклизавање.

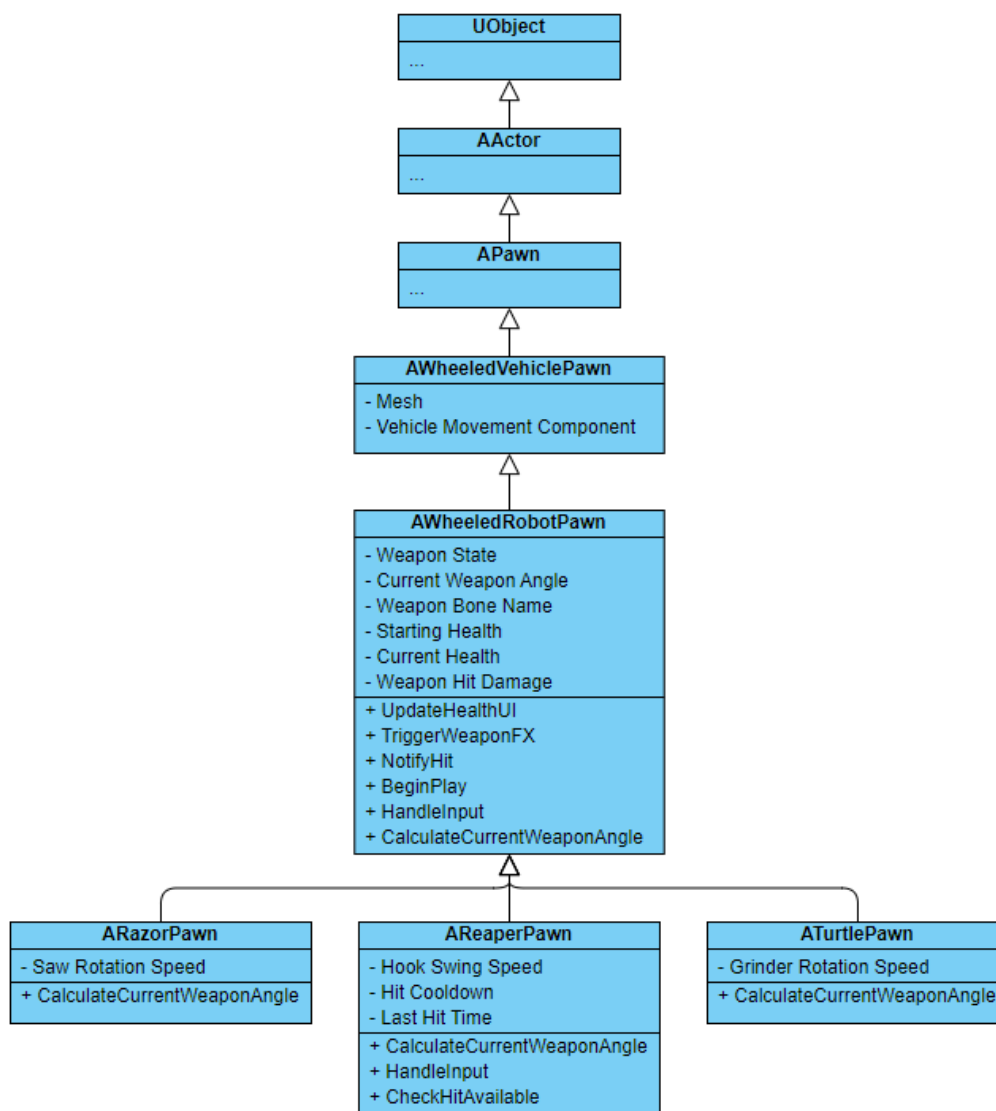
Да би 3Д моделе робота учинили игривим, потребно је креирати *Blueprint* објекат типа *AWheeledVehiclePawn*. Ова класа налази се у увезеном додатку *Chaos Vehicle Plugin* описаном у одељку 3.4. У овом пројекту ова класа је знатно проширена да подржи механику робота и детаљно је описана у наредном поглављу. У класи бирамо визуелни модел за возило и класу за анимације. Као основну компоненту садржи *Vehicle Movement Component* (срп. компонента за кретање возила) и њеном конфигурацијом оспособљавамо модел за вожњу. Такође потребно је креирати и класе за конфигурацију предњих и задњих точкова за сваки од типа робота, што је део истог додатног пакета *Chaos Vehicle Plugin*. Чест проблем, који се јављао и у овом пројекту, при креирању возила из овог пакета, јесте превелики занос возила при скретању услед силе инерције. Решење је пронађено тако што се у класи возила спусти вертикална позиција центра масе. Центар масе више није на средини, већ на дну возила, и то спречава заносење и љуљање возила при скретању.

Потребно и креирати и увезати у логику возила и класу за анимације. Од конфигурације, примењена је уграђена анимација за ротирање точкова у односу на скретање, и уведена нова анимација за кост оружја, и то ротирање по логици која је засебно дефинисана у коду за сваки од типова робота.

4.5. Дефинисање процеса играња

4.5.1. Механика робота

Сваки тип возила (*Razer*, *Reaper* и *Turtle*) имају своју *Blueprint* и *C++* класу. Сви они проширују базну класу *AWheeledRobotPawn* која садржи заједничке особине и понашања за сваког робота и наслеђује постојећу *AWheeledVehiclePawn* класу.



Слика 4.5.1.1. Класни дијаграм архитектуре возила

У базној класи *AWheeledRobotPawn* постоји неколико конфигурационих варијабла које су заједничке за све типове робота, и даље се користе при дефиницији логике робота. При креирању *Blueprint* класе за тип возила, њих је потребно попунити унутар класе. То су: назив кости оружја (користи се при детектовању напада оружјем), почетни животни поени и снага оружја (колико животних поена скида противнику при успешном нападу). Ове вредности дефинишу јачину возила због чега су лако досупни за измену. Сваки тип робота унутар класе треба да унесе и брзину анимације возила. Ове варијабле имају другачији назив за сваки тип робота али им је семантика иста.

Заједничке варијабле за све типове робота су и тренутно стање оружја (активно или неактивно), тренутни животни поени и тренутни угао оружја (помоћна варијабла која се користи при анимацији).

За анимирање оружја служи метода *CalculateCurrentWeaponAngle* која као аргумент прима делта време, односно протекло време између два позива. Позива се унутар *Blueprint* аниматорске класе сваки фрејм, те је аргумент време између два фрејма. У зависности од брзине анимирања оружја, рачуна њихову ротацију, чија се вредност прослеђује аниматорског класи која симулира њихову активност ротирајући кост оружја. За типове робота *Razer* и *Turtle* рачунање овог угла је праволинијски, тренутном углу се дода или одузме (у зависности од жељеног смера ротација) производ делта времена и брзине. И ова вредност се ограничи опсегом од 0 до 360. Проблем се јавља за класу робота *Reaper* чија анимација оружја подразумева замах унапред до 180 степени, а након тога повратак у почетни положај истом брзином.

```

float AReaperPawn::CalculateCurrentWeaponAngle(float DeltaTime)
{
    static bool IsForward = true;
    if (mWeaponState == RobotWeaponState::Inactive)
    {
        return 0.0f;
    }
    else //active
    {
        IsForward = IsForward && mCurrentWeaponAngle > -180.f;
        if (IsForward)
        {
            mCurrentWeaponAngle = mCurrentWeaponAngle - DeltaTime *
mHookSwingSpeed;
        }
        else
        {
            mCurrentWeaponAngle = mCurrentWeaponAngle + DeltaTime *
mHookSwingSpeed;
            if (mCurrentWeaponAngle >= 0) //reset state after animation
is complete
            {
                mWeaponState = RobotWeaponState::Inactive;
                IsForward = true;
            }
        }
        return mCurrentWeaponAngle;
    }
}

```

Блок кода 4.5.1.1. Функција за рачунање тренутног угла оружја робота типа *Reaper*

Ова функција позива се сваки фрејм, те је на почетку декларисана променљива *IsForward* која одређује тренутни смер. Кључном речи *static* остварујемо да се променљива иницијализује само једном и њен животни век важи кроз цео процес. Уколико је статус оружја активан, проверавамо смер и да ли смо стигли до краја анимације. У том случају потребно је увећати тренутни угао у смеру унапред. У супротном, налазимо се у тренутку повлачења оружја ка назад. Тада је потребно променити угао оружја у супротном смеру и проверити да ли смо дошли до краја у смеру ка назад. Уколико јесмо, то је крај анимације и постављамо статус оружја у неактивно стање. Такође постављамо *IsForward* на почетну вредност, активну.

Једна од најважнијих метода у читавом систему јесте метода регистравања успешног напада. У класи *AWheeledRobotPawn* преопредењена је метода *NotifyHit* из класе *AActor*. Ова метода позива се када наш ентитет удари у блокирајући објекат, односно и када дође до судара два робота. Као аргументе, између осталих, прослеђује објекат класе *AActor* са којим смо се сударили, компоненте које су регистровале колизију, локацију где се десио судар, и објекат структуре *FHitResult* који садржи скуп корисних информација о ударцу. Ова метода позива се сваки пут када смо у контакту са другим објектом, сваког фрејма. За типове робота са

ротирајућим оружјем ово не представља проблем, јер се штета аплицира сваког тренутка када контакт постоји. Проблем опет настаје са класом робота *Reaper*, где се дешава да се штета аплицира више пута током једног удarca, докле год је његово оружје у контакту са другим роботом. Потенцијално решење би било конфигурисање робота да чини мању штету, те би се акумулацијом ове мање штете добила права штета. Међутим ово решење није изабрано јер није константно, и у очима играча би штета деловала насумично.

```
void AWheeledRobotPawn::NotifyHit(UPrimitiveComponent* MyComp, AActor* Other,
UPrimitiveComponent* OtherComp, bool bSelfMoved, FVector HitLocation, FVector
HitNormal, FVector NormalImpulse, const FHitResult& Hit)
{
    //Damaged robot calls this
    AWheeledRobotPawn* AttackerRobot = Cast<AWheeledRobotPawn>(Other);
    if (AttackerRobot == nullptr
        || AttackerRobot->GetWeaponState() == RobotWeaponState::Inactive
        || AttackerRobot->GetWeaponBoneName() != Hit.BoneName)
    {
        return;
    }

    if (AREaperPawn* ReaperRobot = Cast<AREaperPawn>(AttackerRobot))
    {
        if (!ReaperRobot->CheckHitAvailable())
        {
            return;
        }
    }

    if (HasAuthority())
    {
        mCurrentHealth -= AttackerRobot->GetWeaponHitDamage();
        OnRep_CurrentHealth();

        ABattleRobotState* AttackerRobotState =
        Cast<ABattleRobotState>(AttackerRobot->GetPlayerState());
        AttackerRobotState->AddScore(AttackerRobot->GetWeaponHitDamage());
    }

    AttackerRobot->TriggerWeaponFX(HitLocation);
}
```

Блок кода 4.5.1.2. Функција која се позива при међусобном контакту два робота

Метода треба да региструје ударац оружјем над роботом који је жртва удarca. Прво проверавамо да ли је оружје нападача активирано и да ли је судар настао од стране оружја противника. Уколико јесте, потребно је урадити додатну проверу за нападачку класу робота *Reaper*. Ова класа садржи податак о времену последњег успешног напада, као и колико времена треба да протекне да се региструје нови.

```

bool AReaperPawn::CheckHitAvailable()
{
    const float CurrentTime = GetWorld()->GetTimeSeconds();
    const bool rtn = CurrentTime - mLastHitTime > mHitCooldown;
    if (rtn)
    {
        mLastHitTime = CurrentTime;
    }
    return rtn;
}

```

Блок кода 4.5.1.3. Функција која проверава спремност оружја робота типа *Reaper*

Уколико је разлика тренутног времена већа од времена које треба да протекне, значи да је оружје спремно за чињење штете, и ажурирамо време последњег напада. Овом методом омогућили смо да се штета ове класе робота не акумулира, већ да се региструје само први напад у кратком временском периоду.

4.5.2. Правила игре

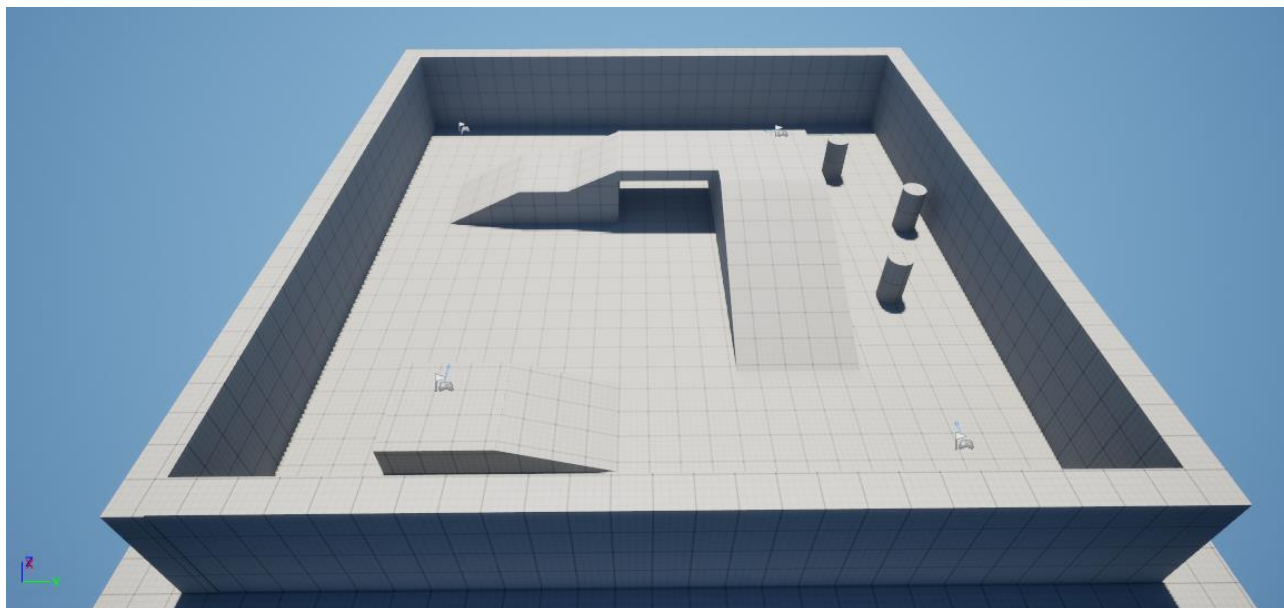
Главна правила игре дефинишу се у класи *GameMode*, односно у конкретном случају у класи *BattleGameMode*, која је наслеђује. Инстанца класе везана је за учитану мапу и у мрежној вишекорисничкој игри постоји само на серверу и служи да управља клијентима, односно играчима. Након дохватања иницијалних података од играча који се тек прикључио сесији, потребно је створити робота који је играч изабрао на оптимално место у свету и спојити га са играчем, тако да може да га контролише.

У овом сегменту развоја игре највећи изазов био је креирање алгоритма за одабир оптималног места одакле играч започиње игру. Метода *FindOptimalSpawnPoint* као аргумент прима *PlayerController* инстанцу од играча за који се ствара објекат робота, а враћа објекат типа *AActor* који представља једну од инстанци типа *PlayerStart* које смо при обради мапе створили у свету. Алрогирам је једноставан и ствара низ парова почетних позиција играча (инстанци типа *PlayerStart*) и целобројног броја који представља тежину одговарајуће позиције. Обиласком свих почетних позиција и позиција постојећих играча у свету, рачунамо раздаљину између њих и додајемо је на тежину позиције. На крају, позиција која има највећу тежину је одабрана, јер је она укупно најудаљенија од свих играча и представља најпоштеније место за почетну позицију.

4.6. Израда мапе и терена за игру

За израду мапа коришћени су модови *Landscape* и *Modelling* унутар *Unreal Engine* окружења. Главна мапа, где се дешава борба робота, направљена је по узору на једноставне

мапе које долазе на примерима пројеката за *Unreal Engine* радни оквир. За израду статичких модела рампи, препрека и зидова коришћен је алат за креирање модела унутар самог радног оквира, јер су у питању једноставни геометријски облици. Како би се симулирало небо и атмосфера, садржи дирекционо светло, експоненцијалну маглу, атмосферу неба и светлост неба. Постављена су четири ентитета типа *PlayerStart* који означавају потенцијална места за почетни положај играча при уласку у игру.



Слика 4.6.1. Изглед арене где се дешава борба робота

Друга мапа је мапа главног менија, и садржи три стуба на којој су приказани модели типова робота које корисник може да изабере. Изнад сваког од њих стоји дирекционо светло различитих боја које се активира при одабиру робота.

4.7. Архитектура подсистема умрежавања

4.7.1. Постојећа подршка за умрежавање

Систем умрежавања унутар *Unreal Engine* платформе прати сервер – клијент архитектуру. То подразумева да у мрежи постоји један рачунар који представља сервер који управља свим умреженим клијентима. Критичне операције обављају се на серверу и прослеђују клијентима. У супротном, малициозним изменама корисничког програма могли би остварити неовлашћену предност, варати. Улога клијента је углавном, између осталог, да проследи улаз од играча ка серверу, као и да добијене податке од сервера интерпретира, односно прикаже на екрану.

Постоје два могућа типа сервера – наменски сервер (енг. *dedicated server*) и ослушкујући сервер (енг. *listen server*). Наменски сервер се покреће одвојено од процеса игре, стално ради у позадини и не зависи од клијената. Његова предност је што клијенти стално, неометано могу да приступају његовим операцијама и прикључују се сесијама. Наменски сервери често немају визуелну репрезентацију, већ се покрећу као конзолни програм. Потребно га је редовно одржавати и напајати, па је зато често опција искључиво високобуџетним компанијама.

Друга опција је да уместо посебног процеса који представља сервер, један од клијената преузме и улогу сервера. То решење назива се ослушкујући сервер. Играч који иницијално креира сесију, преузима уједно и улогу сервера на тој сесији. Пошто се сада сервер извршава на рачунару играча, није потребно изнајмити посебни хардвер за ту намену, што смањује трошкове. Такође, клијент који је и сервер не доживљава никакво кашњење. Лоша страна су слабије перформансе сервера, и лоша скалабилност. Представља идеално решење за игре са мањим бројем играча направљене од стране малих, самосталних студија.

Репликација је систем прослеђивања вредности варијабла са сервера на клијенте. Потребно је означити класу и њене атрибуте реплицираним, и они ће аутоматски, у оквиру мрежног подсистема, бити синхронизовани са свим осталим клијентима. Реплицирана својства се не прослеђују стриктно у тренутку када су измењена, већ се време њихове синхронизације одређује комплексним алгоритмима, са циљем оптимизације мрежних ресурса. Можемо делимично утицати на рад овог алгоритма тако што ћемо унети приоритете у реплицирана својства. Време извршења репликације зависи и од релевантности ентитета. Нису сви ентитети у свету релевантни играчу да их је потребно ажурирати на тренутно стање. Подразумевана логика за рачунање релевантности је удаљеност ентитета од играча – ближи објекти су више релевантни него удаљени и треба их чешће ажурирати, док је удаљене потребно ажурирати тек када остваре одређену границу блискости. Ово подразумевано понашање могуће је редефинисати. Такође, *Unreal Engine* радни оквир нам омогућује да извршимо позив жељене функције када год се вредност репликативног атрибута промени услед репликације. Потребно је додати кључну реч *ReplicatedUsing* и навести име методе која ће се позивати.

У супротном смеру, од клијента ка серверу, комуникацију није могуће остварити преко система репликације. За ове потребе користе се позиви удаљених процедура (енг. *remote procedure calls - RPCs*). Подразумевају извршавање кода на удаљеном рачунару по нашем захтеву. Постоји неколико типова удаљених процедура у зависности где се процедура

извршава. Клијентске удаљене процедуре се извршавају на удаљеном клијенту, а позива их сервер. Углавном се избегавају у корист репликације, која омогућује оптималнији рад мреже. Серверске удаљене процедуре извршава сервер, а захтевају их клијенти. Представљају једини начин како клијент може иницирати комуникацију са сервером. Овом типу удаљених процедура можемо додати и опцију *WithValidation*, где је потребно специфирати методу која ће вршити валидацију података ради спречавања варања. Уколико је валидација неуспешна, удаљена процедура се неће извршити и клијент ће бити избачен из сесије. Последња врста је мултикаст процедура, која се извршава и на серверу и на свим увезаним клијентима. Све удаљене процедуре можемо означити као поуздане (енг. *reliable*) или непоуздане (енг. *unreliable*). Непоуздане процедуре се неће извршити уколико је мрежа загушена, односно нема гаранције да ће се извршити, могу бити прескочене. Поуздане процедуре гарантују извршење и то у редоследу у коме су позване. Користе више ресурса, могу загушити мрежу и треба их користити за кључне, критичне догађаје.

Све стандардне класе подржавају умрежавање и репликацију. *GameModeBase* класа која управља целокупном игром постоји само на серверу. *GameStateBase* класа која служи за информисање играча о стању у игри се реплицира на све клијенте. *PlayerController* класа постоји на серверу и само на клијенту који је власник контролера. *PlayerState* класа која служи за прослеђивање информација о конкретном играчу се реплицира на све клијенте. Објекти класе *Pawn* се реплицирају на све релевантне клијенте, по установљеном алгоритму релевантности.

4.7.2. Имплементирано решење умрежавања

За тип сервера изабран је ослушкујући сервер, што значи да уколико не постоји слободна сесија, играч креира нову и добија улогу сервера. Први корак ка умрежавању играча је креирање сесије. Након притиска на дугме за покретање борбе, прво позовемо асинхрони позив за претрагу сесија. Када добијемо одговор, уколико је успешан, проверавамо низ пронађених сесија. Уколико низ није празан, приступамо првој сесији у низу. Ако јесте, потребно је креирати нову сесију за путем асинхроног позива креирања сесије. Када добијемо повратну поруку о успешном креирању сесије, прелазимо на нову мапу, мапу за борбу, и у опцијама назначимо да преузимамо улогу ослушкујућег сервера.

При уласку у сесију, јавља се проблем како проследити тип робота који је корисник изабрао на сервер који је задужан за креирање робота. Овај проблем се јавља на клијентима

који приступају постојећој сесији. Пошто улазимо у нову мапу, самим тим прелазимо на контролер и нови мод диктиран од стране сервера, и тиме губимо досадашње податке дефинисане у почетном менију. Имплементирано решење користи двосмерну комуникацију између сервера и клијента путем позива удаљених процедура.

```
void ABattleGameMode::OnPostLogin(AController* NewPlayer)
{
    // ... ostala logika

    ARobotController* NewRobotController =
    Cast<ARobotController>(NewPlayer);
    NewRobotController->RequestClientData();
}
```

Блок кода 4.7.2.1. Функција која се позива при уласку новог играча у сесију

Главна серверска класа *ABattleGameMode* редефинише метод који се позива при сваком конектовању новог играча. Одатле на новом контролеру позива клијентску даљинску процедуру којом захтева податке.

```
void ARobotController::RequestClientData_Implementation()
{
    if (!IsLocalController()) { return; }

    FInitRobotData initData;
    initData.ChosenRobotName =
    Cast<URoboClashGameInstance>(GetGameInstance())->ChosenRobotName;
    SendClientData(initData);
}

void ARobotController::SendClientData_Implementation(FInitRobotData initData)
{
    if (!HasAuthority()) { return; }

    ABattleGameMode* GameMode = Cast<ABattleGameMode>(GetWorld()-
    >GetAuthGameMode());
    GameMode->SpawnRobotForPlayer(initData.ChosenRobotName, this);

    ABattleRobotState* RobotState = GetPlayerState<ABattleRobotState>();
    RobotState->SetName(initData.ChosenRobotName);
    OnRep_PlayerState();
}
```

Блок кода 4.7.2.2. Имплементација позива удаљених процедура за иницијализацију података пред борбу

Сваки клијент пре конекције на сесију чува одабрани тип робота у инстанци класе *GameInstance*. Ова класа је искључиво клијентска, и једна од сврха јој је управо чување података између мапа, јер јој је животни век целокупно трајање процеса игре. Након што сервер захтева податке, на клијенту се изврши провера да ли смо у контролеру локалног играча, и којем случају извлачимо податке и сада позивомо серверску удаљену процедуру, где податке прослеђујемо као параметре функције. На серверу се прво врши провера да ли имамо

ауторитет, односно да ли смо у функцији сервера. Након тога, са прослеђеним подацима од клијента креирамо инстанцу робота и спајамо је са играчем. У овој методи се такође прослеђују подаци у *PlayerState* објекат који одговара играчу, који се користи у табели поена током борбе.

Реплицирана својства у базној класи робота су стање оружја (*WeaponState*) и тренутни број животних поена (*CurrentHealth*). Потребно је реплицирати стање оружја које нам говори да ли је оружје активно или не ка свим клијентима у свако време. Тако остварујемо исправне анимације активирања оружја свих противничких играча. Играч такође треба у сваком тренутку знати и колико је животних поена преостала сваком противнику. Ово својство реплицира се уз позив посебне методе која ажурира визуелну репрезентацију њихових животних поена. Ажурирање својства се дешава стриктно на серверу, како не би дошло до злоупотребе, где се такође увећавају и поени играча. Након тога се дешава процес репликације и сви клијенти постају обавештени о новим вредностима. Активирање визуелног ефекта успешног напада је оптимизовано да се не дешава као последица репликације, већ се свакако извршава при регистровању успешног напада на клијенту. Тако добијамо елиминацију потенцијалног кашњења ефекта, а активирање ефекта није критична акција те немамо проблем потенцијалног варања.

У класи *PlayerController* је придружени објекат типа *PlayerState* аутоматски означен као реплициран и већ садржи методу која се позива при извршењу репликације. Ова метода је преклопљена тако да се при добијању нове вредности од стране сервера, ажурира и табела поена играча.

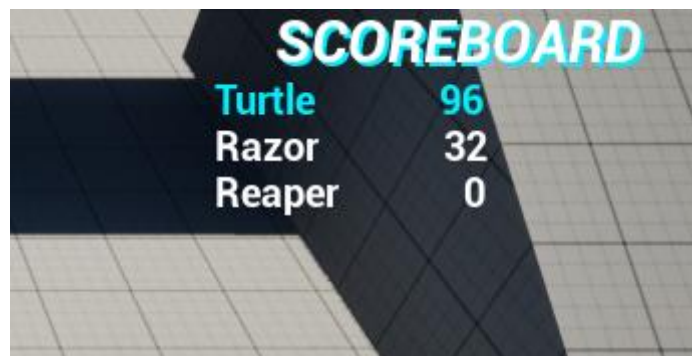
4.8. Кориснички интерфејс

У игри постоје два главна елемента корисничког интерфејса: главни мени и графички интерфејс током борбе робота (типа *HUD*). Главни мени садржи само *Blueprint* класу која одређује изглед самог менија и дефинише његову функционалност. Кликом на дугме за отпочињање борбе, мени извршава логику проналажења и креирања сесије, као и чување изабраног типа робота. Инстанца ове класе се креира из главне класе за почетни екран, *MainMenuGameModeBP*, која је и активира на екрану.

Током борбе, изнад сваког робота налази се графички приказ његових животних поена. Подразумева бели правоугаоник са црвеним пуњењем, које означава процентуално стање животних поена робота. На почетку борбе, роботу су поени максимални и правоугаоник је у потпуности испуњен црвеном бојом. Попуњена боја се постепено смањује како се губе

животни поени током борбе. Ова компонента је додата директно на робота унутар његове *Blueprint* класе, тако да увек прати позицију возила. Конфигурацијом је подешена да увек буде окренута ка камери. Имплементирана је посебна функција унутар компоненте која као аргументе прима број тренутних животних поена и њихов проценат, и на основу тога попуњава правоугаоник и исписује текст.

Игра треба редовно ажурирати табелу поена играча. За ту потребу креиране су две компоненте 2Д корисничког интерфејса, сама табела, и улаз у табелу. Потребно је засебно ажурирати сваки улаз, тако да класа табеле осим визуелног изгледа, омогућује и дохватање сваког улаза појединачно. Укупно има четири улаза, за максимални број конектованих играча, а уколико је број играча мањи, одређени улази се не приказују на екрану. Након дохватања улаза који одговара играчу, направљене су функције којима можемо поставити поене, име, као и истаћи улаз (обојити га плавом бојом, што представља локалног играча). Имплементација ових функција се налази унутар *Blueprint* класе улаза, а позивамо их из *C++* кода при синхронизацији поена са сервера.



SCOREBOARD	
Turtle	96
Razor	32
Reaper	0

Слика 4.8.1. Изглед табеле играча током борбе

5. ОПИС РАДА СИСТЕМА

Резултат пројекта је извршни фајл игре коју можемо покренути на *Windows* оперативном систему. Ради лакшег покретања већег броја инстанци, можемо користити опцију вишеструког покретања из *Unreal Engine* окружења. Подразумевана мапа која се учитава при уласку у игру у мапа главног менија. Садржи равну површину са платформама на којима се налазе модели свих типова робота и радијални извор светлости који их осветљује. 2Д графички кориснички интерфејс на екрану исписује име игре великим словима, и испод робота исписује име селектованог робота. Главни елемент представља дугме *Play* којим отпочињемо игру. Ово дугме је на почетку онемогућено док не изаберемо тип робота. Кликом левог тастера миша на неки од робота, чинимо га тренутно изабраним. То се осликава активирањем одговарајућег рефлектора изнад робота, као и исписом имена робота на кориснички интерфејс. Рефлектор изнад сваког робота сија другачијом бојом, одговарајући типу робота. Када одаберемо тип робота, дугме постаје омогућено и можемо отпочети игру.



Слика 5.1. Почетни екран када је изабран тип робота *Razor*

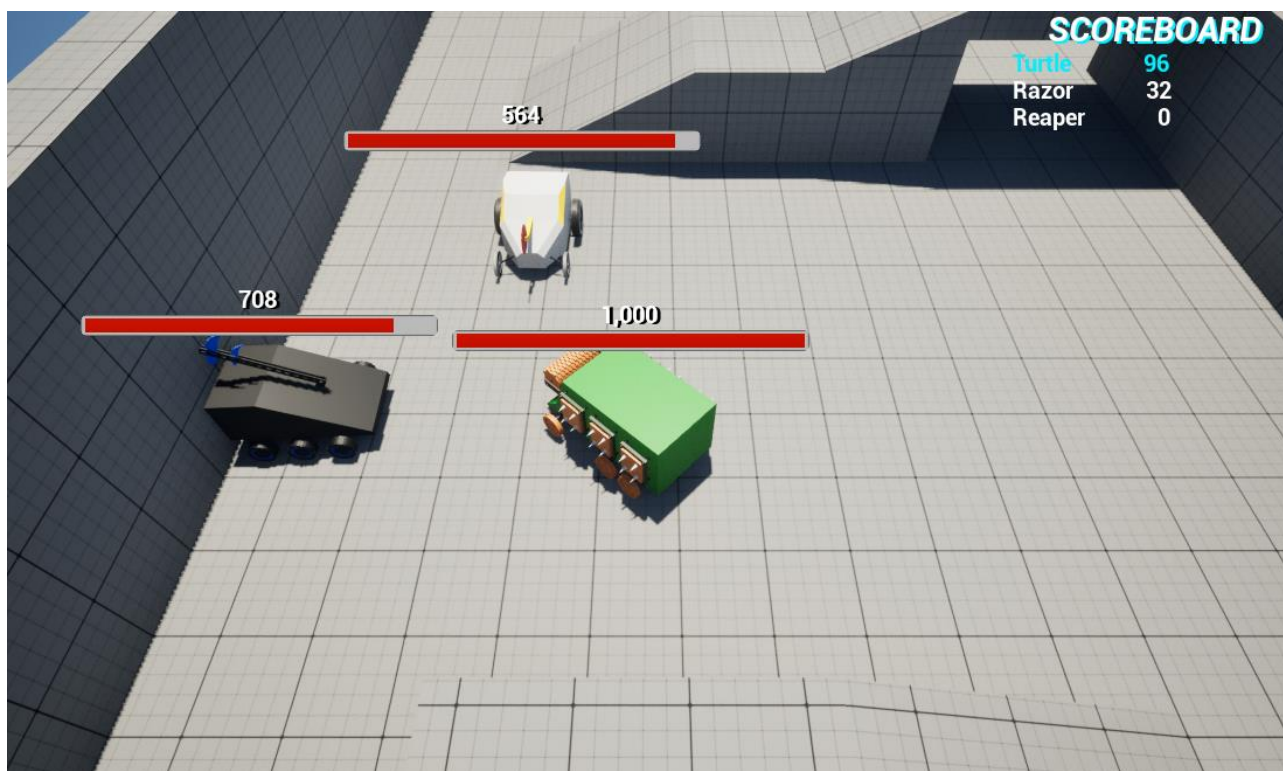
Сваки тип робота има јединствене карактеристике описане у следећој табели:

Табела 5.1. Типови робота и њихови описи

Назив	Изглед	Оружје	Предности	Мане
Razor (Жилет)	Троугони облик, четири точка, карактеристична боја црвена	Кружна тестера	Брзина, јачина оружја, мала површина	Нестабилност, мали почетни животни поени
Reaper (Косач)	Мат црна боја, шест точкова, карактеристична боја плава	Коса	Стабилност, јачина оружја, окретност	Средњи почетни животни поени, велика површина
Turtle (Корњача)	Зелено – браон боја, шиљци, карактеристична боја зелена	Дробилица	Велики почетни животни поени, велика површина оружја	Слаба јачина оружја, спорост, слаба окретност

Након уласка у сесију, добијамо контролу над типом робота који смо изабрали и отпочињем борбу са једног од четири предефинисана места у мапи. Максималан број умпрежених играча је четири. Камера је позиционирана из птичије перспективе и прати позицију робота. Робота контролишемо преко тастатуре помоћу тастера: W – гас, кретање унапред, S – кретање уназад, A – скретање улево, D – скретање удесно. Могуће је симулирати ручну кочницу притиском тастера *Space*. Активирање оружја врши се притиском тастера E. Код робота типа *Reaper* и *Turtle*, притиском тастера оружје се врти и остаје активно до поновног притиска. Код робота типа *Reaper* притиском тастера врши се један замах оружјем.

Игра нема дефинисан крај већ је циљ константно прикупљање поена и одржавање позиције на табели приказаној у горњем десном углу. Сваки тип робота креће са максималним животним поенима дефинисаним у својој класи и анулираним поенима у табели. Током игре, потребно је оружјем начинити штету другим играчима. Сваки успешни ударац скида одређен број животних поена противнику и награђује нападача повећавањем поена.



Слика 5.2. Исечак из игре током борбе свих типова робота

Потребно је водити рачуна о својим животним поенима, јер уколико дођемо до нуле, губимо одређен број поена, робот нам се уништава и поново започињемо игру са једне од подразумеваних тачака. Исто тако уништавајући роботе других играча смањујемо поене противника.

6. ЗАКЉУЧАК

Израдом овог рада потврђене су широке могућности и неограничен потенцијал *Unreal Engine* радног оквира при креирању видео игре. Пројекат је успешно имплементирао кључне делове игре као што су ток игре, правила игре, механика робота, кориснички интерфејс и умрежавање више играча. Креирана су три типа робота, сваки са уникатном механиком, визуелном репрезентацијом и оружјем. У вишекорисничком окружењу роботи исправно и благовремено синхронизују своје атрибуте између различитих играча, демонстрирајући високу ефикасност *Unreal Engine* подсистема за умрежавање. Отворени код алата *Unreal Engine* омогућио је дубоко залажење у најситније детаље имплементације самог радног оквира. Истражени су детаљи имплементације основних класа попут класе *Actor* и *Pawn*, као и детаљи архитектуре подсистема за умрежавање.

Да би вишекорисничка игра одржала интересовање играча, потребно ју је редовно одржавати и унапређивати. Игра се може бесконачно ширити додавањем нових типова робота и нових мапа. Захваљујући скалабилној архитектури система, тај процес је праволинијски. Тренутно постоји мали број специјалних ефеката, на чему треба радити да се овај број повећа, јер би побољшало доживљај играча. Комплексност борбе је тренутно ниска, и може се значајно закомпликовати додавањем слабих тачака робота. Односно, делова робота који би примали већу штету уколико су погођени оружјем. Ова механика би додатно раздвојила искусне играче од почетника, и натерала почетнике да боље разумеју игру и посвете јој више времена. Такође је добра пракса стално радити на мрежној архитектури и побољшавати је, како би остварили што мање кашњење и реалистичнију борбу.

Резултат овог пројекта је мрежна, вишекорисничка, борбена, брза, хаотична игра са фер и уникатним роботима. Садржи систем рангирања по бодовима који подспешује играче на конкуритивност и посвећеност. Резултат имплементације пројекта је снажна база мрежне и играчке логике која се може искористити као основа за комплексније игре сличног типа. Циљ овог документа је да прикаже целокупни процес израде једне 3Д мрежне вишекорисничке игре креиране у *Unreal Engine* радном оквиру. Детаљно су описани изазови и проблеми током развоја видео игре, као и приложена и објашњена имплементирана решења. Објављене су корисне информације и решења за наишле проблеме ради информисања будућих пројеката.

ЛИТЕРАТУРА

- [1] <https://cedric-neukirchen.net/docs/multiplayer-compendium/network-in-unreal> (последњи пут приступљено септембра 2024.)
- [2] Званична *Unreal Engine* документација <https://dev.epicgames.com/documentation/en-us/unreal-engine/nanite-virtualized-geometry-in-unreal-engine> (последњи пут приступљено августа 2024.)
- [3] Званична *Unreal Engine* документација <https://dev.epicgames.com/documentation/en-us/unreal-engine/lumen-global-illumination-and-reflections-in-unreal-engine> (последњи пут приступљено августа 2024.)
- [4] Званична *Unreal Engine* документација <https://dev.epicgames.com/documentation/en-us/unreal-engine/world-partition-in-unreal-engine> (последњи пут приступљено августа 2024.)
- [5] Званични *Unreal Engine* услови коришћења <https://www.unrealengine.com/en-US/eula/unreal> (последњи пут приступљено септембра 2024.)
- [6] Гинисова књига рекорда <https://www.guinnessworldrecords.com/world-records/most-successful-game-engine> (последњи пут приступљено септембра 2024.)
- [7] <https://unrealcommunity.wiki/live-compiling-in-unreal-projects-tp14jcgs> (последњи пут приступљено јула 2024.)
- [8] Линк ка алату <https://chromium.github.io/vs-chromium/> (последњи пут приступљено јуна 2024.)
- [9] Линк ка алату <https://www.toptal.com/developers/gitignore> (последњи пут приступљено јуна 2024.)
- [10] Joanna Lee, „Learning Unreal Engine Game Development“, доступно на: <https://subscription.packtpub.com/book/game-development/9781784398156/1/ch011v11sec12/the-components-of-unreal-engine-4> (последњи пут приступљено септембра 2024.)

СПИСАК СЛИКА

Слика 2.1. Визуелизација колизионе мреже свих карактера и окружења.....	8
Слика 3.1.1. Исечак из игре <i>Unreal</i> објављене 1998. године, прве игре израђане на <i>Unreal</i> радном оквиру.....	10
Слика 3.1.2. Исечак из игре <i>Black Myth: Wukong</i> објављене 2024. године, израђене на <i>Unreal</i> радном оквиру	11
Слика 4.2.1.1. Поједностављена стандардизована архитектура <i>Unreal Engine</i> пројекта.	21
Слика 4.3.1.1. Изглед једног од модела у алату <i>Blender</i> након процеса моделовања.	23
Слика 4.3.2.1. Текстура модела.	24
Слика 4.3.2.2. Финални изглед модела са текстуром.	24
Слика 4.5.1.1. Класни дијаграм архитектуре возила.....	26
Слика 4.6.1. Изглед арене где се дешава борба робота.....	31
Слика 4.8.1. Изглед табеле играча током борбе.....	36
Слика 5.1. Почетни екран када је изабран тип робота <i>Razor</i>	37
Слика 5.2. Исечак из игре током борбе свих типова робота.	39

СПИСАК ТАБЕЛА

Табела 4.2.2.1. Табела логичких целина пројекта и њихов опис	22
Табела 5.1. Типови работа и њихови описи.....	38

СПИСАК БЛОКОВА КОДА

Блок кода 4.1.1.1. Стабло структуре фолдера и фајлова.....	18
Блок кода 4.5.1.1. Функција за рачунање тренутног угла оружја робота типа <i>Reaper</i>	28
Блок кода 4.5.1.2. Функција која се позива при међусобном контакту два робота.....	29
Блок кода 4.5.1.3. Функција која проверава спремност оружја робота типа <i>Reaper</i>	30
Блок кода 4.7.2.1. Функција која се позива при уласку новог играча у сесију	34
Блок кода 4.7.2.2. Имплементација позива удаљених процедура за иницијализацију података пред борбу	34

.