# Razvoj igre u stvarnom vremenu temeljene na Blockchain-u

**Skok, Karlo**

Sveučilište Jurja Dobrile u Puli
Fakultet informatike

**KARLO SKOK**

# RAZVOJ IGRE U STVARNOM VREMENU TEMELJENE NA BLOCKCHAIN-U

Diplomski rad

Pula, rujan 2020.

Sveučilište Jurja Dobrile u Puli
Fakultet informatike

**KARLO SKOK**

# RAZVOJ IGRE U STVARNOM VREMENU TEMELJENE NA BLOCKCHAIN-U

Diplomski rad

**JMBAG:** 0246057991, redoviti student
**Studijski smjer:** Informatika

**Kolegij:** Dizajn i programiranje računalnih igara
**Znanstveno područje:** Društvene znanosti
**Znanstveno polje**: Informacijske i komunikacijske znanosti
**Znanstvena grana:** Informacijski sustavi i informatologija

**Mentor:** izv. prof. dr. sc. Tihomir Orehovački

Pula, rujan 2020.


## IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisan Karlo Skok, kandidat za magistra informatike, ovime izjavljujem da je ovaj Diplomski rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio diplomskog rada nije napisan na nedozvoljen način, odnosno da je prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi.

Student

_____

U Puli, _____, _____ godine


**IZJAVA O KORIŠTENJU AUTORSKOG DJELA**

Ja, Karlo Skok dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj diplomski rad pod nazivom „Razvoj igre u stvarnom vremenu temeljene na blockchain-u“ koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu s Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.
Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

Potpis

_____

U Puli, _____ (datum)

# SADRŽAJ

# SAŽETAK

Ciljevi ovog diplomskog rada su izraditi igru za više igrača u stvarnom vremenu na blockchain tehnologiji, istražiti i objasniti trenutnu primjenu blockchain tehnologije te predložiti mogućnosti primjene tehnologije u drugim sektorima, posebice kroz model pametnih ugovora te budućnost blockchain tehnologije.

Jedna od ideja kako pojednostaviti i ubrzati transakcije bez utjecaja treće stranke koja garantira povjerenje je pametni ugovor (eng. smart contract). Vrsta programa (ugovora) koji se pokreće na blockchainu i omogućuje provođenje poslovnih pravila kroz programski kod. Upotreba blockchaina kao tehnologije na kojoj se provode pametni ugovori moglo bi ubrzati i pojednostaviti transakcije u svim područjima poslovanja, te pojednostaviti proces određivanja autentičnosti i vlasništva.

Kroz igru za više igrača omogućujemo igraču potpuno vlasništvo nad likovima iz igre što ujedno stvara i osjećaj vrijednosti s obzirom da se likovi mogu smatrati igračevom imovinom. Imovina koju igrač posjeduje nalazi se na blockchain mreži na pametnom ugovoru i nitko je ne može oduzeti. Ono što izdvaja igru od ostalih je što likovi igre postoje u digitalnom obliku na pametnom ugovoru za razliku od ostalih igara kod kojih imovina igrača nije pod njihovim vlasništvom nego je pod vlasništvom centraliziranog poslužitelja, pružatelja usluge i sl.

**Ključne riječi:** igra, multiplayer, blockchain, pametni ugovori, real time, tehnologija, transakcije, Unity

## ABSTRACT

The objectives of the final thesis are to express real-time multiplayer game on blockchain technology, explore and explain current examples of blockchain technologies and suggest possibilities for applying the technology in other sectors, especially through the smart contract and the future of blockchain technologies.

One of the ideas on how to simplify and speed up transactions without the influence of a third party that guarantees a trust is a smart contract. The type of program (contract) that runs on the blockchain and allows the implementation of business rules through program code. The use of blockchain as a technology on which smart contracts are implemented could speed up and simplify transactions in all areas of business, and simplify the process of determining authenticity and ownership.

Through multiplayer, we allow the player complete ownership of the characters in the game, which also creates a sense of value since the characters can be considered the player's property. The property the player owns is on the blockchain network on a smart contract and no one can take it away. What sets the game apart from others is that the characters of the game exist in digital form on a smart contract, unlike other games where the player's property is not owned by them but by a centralized server, service provider, and the like.



**Keywords:** game, multiplayer, blockchain, smart contracts, real time, technology, transactions, Unity

# 1. UVOD

U radu će se govoriti o tehnologijama i mehanikama koje su korištene za izradu projekta. Igra za više igrača odvija se u stvarnom vremenu te koristi blockchain za slanje i primanje podataka. Korisnici se spajaju, točnije prijavljuju, u igru pomoću jedinstvenog privatnog ključa koji je izgeneraran u Ganache lokalnoj simulaciji. Sam rad se sastoji od dva dijela, teorijski i praktični.

Kao što je spomenuto, u prvom dijelu rada govoriti će se ponajviše o korištenim tehnologijama, mehanikama, ideji i sl. Svaka tehnologija biti će ukratko objašnjena i za svaku tehnologiju naveden je dio praktičnoga dijela. U ovom slučaju, cijeli rad je napravljen u C# programskom jeziku uz korištenje blockchain tehnologije, a projekt je rađen u Unity razvojnom sučelju.

U praktičnom dijelu opisuje se izrada projekta počevši od prijave i registracije igrača, glavnog izbornika te scene bitke. Naravno, prije samih formi igre, obrađen je dio u kojem je navedena neka od sličnih igri, zatim koncept igre te sama ideja za odabir okruženja.

Kroz izradu rada, cilj je bio izraditi te prezentirati igru koja će biti atraktivna po modernim standardima, uz korištenje raznih tehnologija koje nisu toliko česte, s potencijalom za monetizaciju. Motivacija za izradom rada na ovu temu je osobni interes prema industriji razvoja video igara. Odabran je specifično razvoj igre za više igrača s obzirom da su takve igre kompleksnije samim time što moraju biti sinkronizirane i odvijati se u jednakom vremenu, a s druge strane postoji neispunjeni prostor na tržištu video igara koji bi ovakva igra mogla popuniti uz buduće nadogradnje. S navedenim ciljem, istražene su najbolje prakse za razne elemente igre te proučene prakse koje se koriste u drugim igrama. Također, detaljno smo se dohvatili blockchain tehnologije, objasnili smo njegovu funkcionalnost te upotrebu pri spremanju podataka. Spomenuli smo kriptografske hash funkcije i njihovu ulogu u blockchain tehnologiji. Dotakli smo se načina stvaranja blokova i spremanja podataka u iste. Ujedno će se vidjeti prednosti i nedostaci u odnosu na klasično spremanje podataka, a na praktičnom primjeru će se pokazati i upotreba same tehnologije.

# 2. KORIŠTENE TEHNOLOGIJE

Ideja je bila upotrijebiti nekoliko različitih tehnologija u jedan veći i drukčiji projekt, a svaka od tih tehnologija biti će ukratko opisana u sljedećim odlomcima, a kasnije prikazane i na praktičnom dijelu.

## 2.1 C#

C# je objektno orijentirani programski jezik za brzi razvoj aplikacija (eng. rapid application development) RAD. Ovaj programski jezik je razvijen u sklopu Microsoftove .NET inicijative koja je trebala ublažiti probleme s kompatibilnosti i omogućiti pokretanje aplikacija na različitim operativnim sustavima (OS), ali također i pružiti mogućnosti koje programerima nude gotova rješenja i funkcionalnosti da bi ubrzala i pojednostavila razvoj aplikacija svih vrsta i oblika. C# je sličan jezicima C i C++ iz kojih je proizašao također ima jako velike sličnosti s Javom (naročito u ranijim verzijama) što je u vrijeme kada je C# izašao pobudilo dosta kontroverze ali također omogućilo lak prijelaz s jednog programskog jezika na drugi. Zapravo C# spaja moć i efikasnost C++-a, jednostavni i čisti objektno orijentirani dizajn Jave sa pojednostavljenim korištenjem kao npr. u Visual Basic-u.

C# je programski jezik koji se povodi za događajima (eng. event driven) u njemu se pišu programi koji reagiraju na događaje koje je napravio korisnik npr: klikovi mišem, pritiskom tipki na tipkovnici, dodirom na dodirnom ulaznom uređaju (touchpad) i ostalim gestama koje se rabe na računalima, pametnim telefonima i tabletima. Također C# omogućava asinkrono programiranje u kojem više zadataka može biti obavljeno u isto vrijeme što čini aplikacije više reaktivnima na korisnikovu interakciju s njima. Kao objektno orijentirani programski jezik važne karakteristike C#-a su: enkapsulacija, nasljeđivanje i polimorfizam.

## 2.2 Unity3D

Unity je višeplatformsko okruženje za razvoj video igara, koje je razvio Unity Technologies. Unity se često koristi za razvoj računalnih igara i simulacija, koje se mogu pokrenuti na računalu, igraćim konzolama i mobilnim uređajima. Značajni uspjeh Unity-ja se može pridodati širokom rasponu platformi koje ga podržavaju, ali i tome što se lako nauči zbog količine sadržaja dostupne na internetu te opširnoj dokumentaciji dostupnoj na web stranici Unity-ja. Unity je napisan u C i C++ jeziku dok za skriptiranje ponašanja objekata u igri se mogu koristiti C#, UnityScript (vrlo sličan JavaScript-u) i Boo. Skriptiranje unutar Unity-ja je izrađeno na Mono-u, koji je open-source implementacija Microsoft-ovog .NET Framework-a. Skripte koje korisnik napiše ili unaprijed napisane skripte koje dolaze sa Unity-jem se nazivaju komponentama. Skripte, odnosno komponente, se postavljaju na objekte u igri pomoću interaktivnog korisničkog sučelja čime se znatno ubrza razvoj igre. Unity dolazi s unaprijed definiranim komponentama među kojima su: Rigidbody, Collider, Transform itd. Određene komponente su unaprijed postavljene na svakom objektu koji se doda u igru ili će proširiti mogućnosti tog objekta na kojem su dodane [1]. Komponente mogu dodat mogućnosti kao što su: otkrivanje sudara dvaju objekata, utjecaja sile teže na objekte itd. Skripte u ovom završnom radu su napisane u C# programskom jeziku.

## 2.3 Nethereum

Nethereum je .Net integracijska biblioteka (eng. library) za Ethereum, pojednostavljujući upravljanje pametnim ugovorima i interakciju s čvorovima Ethereum bez obzira jesu li javni, poput Geth , Parity ili privatni, poput Quoruma i Besu .

Nethereum se razvija ciljajući netstandard 1.1, net451, a također i kao prijenosna knjižnica, stoga je kompatibilan sa svim glavnim operativnim sustavima (Windows, Linux, MacOS, Android i OSX) i testiran je na oblaku, mobilnom uređaju, radnoj površini, Xboxu, hololensima i prozori IoT [2].

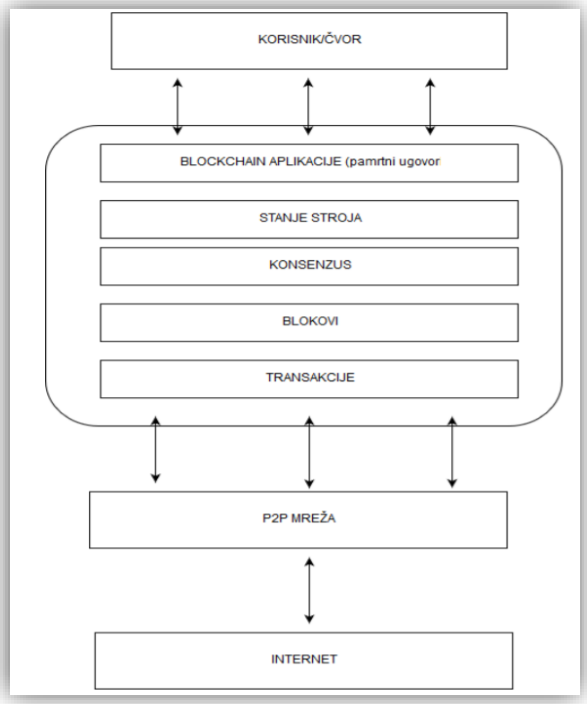
## 2.4 Photon

Photon je posredni softver (eng. middleware) koji omogućuje razvoj skalabilnih više platformskih igara za više igrača u stvarnom vremenu. Pruža pakete za razvoj programa (eng. software development kit, SDK) koji sadrže biblioteke za razne platforme, kao što su Unity, iOS, Android, Windows itd. S obzirom na to da se sve biblioteke spajaju na istu poslužiteljsku aplikaciju (eng. back-end) koristeći iste komande i logiku, što omogućuje više platformsko igranje. Postoje dva Photon servisa: Photon Server i Photon Cloud. Za potrebe ovog projekta i radi jednostavnosti, koristit će se Photon Cloud. Korištenjem Photon Cloud-a izbjegavamo pisanje kompleksne serverske logike. Photon Cloud je skalabilan, u smislu da se broj istovremeno spojenih igrača mijenja ovisno o potrebnom kapacitetu i to neprimjetno. Garantiran je niski odziv, jer Photon Cloud pruža servere u SAD-u, Europi i Aziji. Photon Cloud se prodaje kao, softver kao usluga (engl. software as a service) [3]. Dostupne ponude licenciranja servisa bazirana su na broju istovremeno povezanih igrača i plaćaju se mjesečno. Besplatna ponuda nudi 20 istovremeno povezanih igrača što je dovoljno u svrhu razvoja igre.

Kako bi mogli koristiti Photon Cloud u našem projektu, trebamo skinut Photon Unity Networking (PUN) sa Unity-jevog Asset Store-a. PUN se pokreće na Photon Cloud-u, te omogućuje uparivanje igrača nasumično ili ovisno o zadanim uvjetima pretraživanja, uz to ima i sve mogućnosti koje su navedene za Photon Cloud. Dostupna dokumentacija za PUN omogućava vrlo brzu i laku implementaciju u projekt.

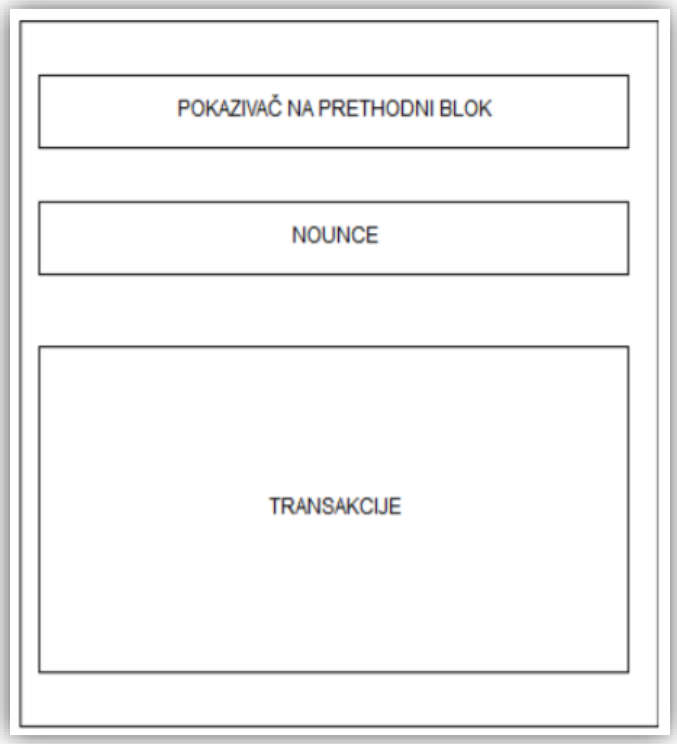
## 2.5 Blockchain

Blockchain možemo promatrati kao platformu na kojoj čvorovi mogu izmjenjivati vrijednosti preko transakcija bez potrebe centralnog autoriteta koji bi osiguravao ispravnost transakcije. Sadržano u samom imenu, blockchain čini skupina „blokova“ međusobno povezanih u lanac.




*Slika 1: Mrežni prikaz blockchaina [4]*

Možemo ga usporediti s vezanom listom. Svaki blok čini logičku organizaciju provedenih transakcija, a veličina bloka ovisi o načinu implementacije blockchaina. Svaki blok uključuje i heširani pokazivač na prijašnji blok u lancu, osim ako se radi o prvom bloku u lancu (eng. genesis block). Struktura bloka također ovisi o tipu blockchaina i načinu implementacije, a glavni atributi koje svaki blok mora sadržavati su: zaglavlje, pokazivač na prijašnji blok, vremenska oznaka, nonce (nasumičan broj koji se koristi pri izračunavanju novog bloka), brojač transakcija i transakcije [4].




*Slika 2: Struktura bloka [4]*

Ethereum je otvorena softverska platforma koja se temelji na blockchain tehnologiji, te developerima omogućava stvaranje i objavljivanje decentraliziranih aplikacija. Ethereum mreža djeluje slično kao i bilo koja druga blockchain mreža. Svaka nova transakcija se bilježi i zatim ispituje od strane različitih čvorova poznatih kao rudari. Navedeni rudari izvršavaju programski kod na svom računalu kako bi ažurirali transakciju u decentraliziranoj knjizi. Rudari dobivaju 3 etera za svaki kod koji izvrše ili za svaku transakciju koju dodaju u lanac. Rezultat koda svakog rudara dodaje se konsenzusu, koji se zatim ispituje kako bi se potvrdila autentičnost zadnje transakcije radi nesmetanih transakcija u cijeloj mreži. Umjesto "rudarenja" kao u slučaju Bitcoina, u blockchainu Ethereuma korisnici rade kako bi zaradili Ether, kriptovalutu koja održava širu mrežu. Osim što se radi o kriptovaluti kojom se može trgovati, programeri koriste Ether za plaćanje usluga na mreži Ethereuma [5].

Sličnost Ethereuma i Bitcoina ogleda se u činjenici, da su oba izgrađena uz pomoć tehnologije distribuirane javne mreže poznatije kao blockchain, ali tu sva sličnost prestaje. Bitcoin nudi jednu jedinu aplikaciju, peer-to-peer sustav elektroničke valute koji omogućava plaćanje putem Interneta. Prema svojoj biti, Bitcoin je valuta u svojem najčišćem obliku. Dok Bitcoin koristi blockchain tehnologiju kao platformu za praćenje podataka tko posjeduje Bitcoin, Ethereum koristi blockchain kao platformu za pokretanje gotovo svih decentraliziranih aplikacija [6].

### 2.6 Solidity

Solidity je programski jezik namijenjen programiranju pametnih ugovora. Nastao je pod utjecanjem drugih programskih jezika, kao što su C++, Python te najviše JavaScript-a, kako bi bio što poznatiji web programerima [8]. Napravljen je kako bi funkcionirao s EVM (eng. ethereum virtual machine). EVM je virtualni stroj na kojemu se pokreće kod Ethereum pametnog ugovora. Ethereum virtual machine radi samo s byte-kodom, koji se dobiva korištenjem ABI-a. ABI (eng. application binary interface) sluţi za kodiranje i dekodiranje Solidity-a u byte-kod i suprotno [7].

### 2.7 Ganache

Ganache je virtualni blockchain koji postavlja 10 zadanih Etheruem adresa, zajedno s privatnim ključevima i svime, i unaprijed ih učitava sa po 100 simuliranih etera. S Ganacheom se ne vrši "rudarstvo" - umjesto toga, on odmah potvrđuje bilo kakvu transakciju koja dolazi. To omogućuje iterativni razvoj - možemo napisati jedinstvene testove za svoj kod koji se izvršava na ovom simuliranom blockchainu, implementirati pametne ugovore, pozvati funkcije, a zatim sve to srušiti za daljnju simulaciju ili nove testove, vraćajući sve adrese na početne vrijednosti stanje 100 Etera [9].

Ovo je posebno napravljeno kako bi se smanjila ovisnost o Ethereum softveru novčanika, koji zahtijeva puno resursa za pokretanje lokalne instance blockchaina.

Korištenjem Ganachea možete pokrenuti lokalni blockchain u roku od nekoliko sekundi i on ne zahtijeva velike resurse CPU / memorije ili diska [10].

# 3. RAZRADA IDEJE

Prije same izrade igre veoma je bitno imati jasnu ideju i cilj igre. Pri izradi ovog rada koriste su razne metode za generiranje ideja, a neke od njih su oluja mozgova (eng. brainstorming**)** i traženje inspiracije u drugim igrama. Brainstorming je metoda rješavanja problema u kojoj se u obzir i razmatranje uzimaju sve ideje, pa čak i one besmislene. Ovom tehnikom često se koriste poslovnjaci u kriznim situacijama, umjetnici, ali i studenti kada se nađu pred zidom. Metoda je dobila ime po metodi „using the brain to storm a problem“ (doslovno: rabiti mozak za na brzo rješavanje problema).

Ideje se nalaze posvuda i prolaze naš unutarnji sustav odluke o tome koja ideja je dobra ili nije. Većinom sami odlučujemo o ideji koja će se prenijeti našem užem krugu te u skladu s njom postupati. Određen postotak takvih ideja prolazi kroz mogućnost povratne informacije koja ponekad ne bude jednaka onom početnom entuzijazmu osobe koja ju je smislila. Na tom koraku većina ljudi stane. Cilj je bio pronaći ideju koja bi mogla obuhvatiti nekoliko kolegija koje smo obradili na fakultetu te iz toga izvući ono najbolje. Slijedeći taj cilj, stigla je i sama ideja, a to je upravo ovaj projekt u kojem smo izradili igru za više igrača temeljenu na blockchainu.

## 3.1 Slične igre

Ideja za kreiranje ovog rada proizašla je iz već postojeće igre. Radi se o igri „Clash Royale“. To je jedinstvena igra za pametne uređaje. Ona kombinira elemente strategije u stvarnom vremenu, umrežavanje s više igrača, skupljane karata te dvoboj pomoću njih. Žanr igre nema točno definiran naziv, ali dosta ljudi ga naziva napad na toranj (eng. tower assault). Clash Royale daje mogućnost igračima da kreiraju klanove (eng. clans) i ratuju jedni protiv drugih. Glavni zadatak u igri je slanje vojnih jedinica sa zadatkom uništavanja protivničkih tornjeva. Igra je simpatična, zarazna i brza pa je savršeno prikladna za mobilne uređaje. Clash Royale je jedna od najpopularnijih mobilnih igara na svijetu i respektabilan elektronski sport (eng. esport) koji se sve vise razvija [28].


*Slika 3: Clash Royale [30]*

### 3.2 Koncept igre

Igra je zamisljena kao simulacija bitke između dva igrača. Svaki igrač kontrolira svoju vojsku te ima zadatak uništiti sve protivnikove kule. Igrači na početku bitke imaju po tri kule koje moraju zaštititi od protivnika. Ishod bitke može biti pobjeda ili poraz, a to ovisi o jačini likova (eng. characters) s kojima igrač barata te vještini i strategiji koju je pripremio za bitku. Za ulazak u igru igrač se mora registrirati te na početku igre dobiva pet osnovnih likova. Unutar igre moguće je kupovati nove likove. Da bi kupili lika mora se odabrati jedna od četiri škrinje koje su poredane po tipu rijetkosti. U igri postoje 20 vrsta likova, a svaki od njih ima svoja obilježja, od kojih su najznačajnija razina života, brzina, i jačina napada. Spomenuta obilježja direktno utječu na ishod bitke. Za ulaz u bitku igrač mora odabrati točno pet likova s kojima će se boriti. Sama bitka se odvija tako da igrači postavljaju svoje likove na teren s ciljem da unište sve tri protivničke kule. Kako bi se izbjeglo postavljanje likova pored protivničkih kula, ograničeno je postavljanje samo na igračevu polovicu,

odnosno polovicu terena na kojoj su igračeve kule. Ključna obilježja igre su ta da je ovo igra za više igrača te je povezana na pametni ugovor na kojem su pohranjeni svi podaci vezani uz igrača. Igra nema jedan žanr, nego je spoj triju različitih žanrova: MOBA (Multiplayer online battle arena), obrana tornjeva (eng. tower defense) i igra s kartama (eng. card game). Sve to je sastavljeno u jednu cjelinu te svaki aspekt igre bit će objašnjen u idućim poglavljima.

### 3.3 Odabir okruženja

Postoje mnogi razni alati za izradu video igara, neki od njih nude izradu 2D ili 3D igara, također, velika većina okruženja (eng. engine) podržava obje dimenzije pri kreiranju igre. Pri izradi ovoga rada za izbor okruženja odabrali smo Unity. Prednost je što je Unity besplatan pa mu svatko može pristupiti. Pomoću besplatne verzije Unity-a mogu se kreirati i prodavati igre koje podržavaju vise platformi. Neke od njih su: Windows, Linux, iOS, Android, BlackBerry, itd.

Iako Pro verzija Unity-a uključuje nekoliko dodatnih značajki kao sto je podrška za razvoj na PlayStation i Xbox platformama, besplatna verzija nam je dovoljna za učenje i izradu igara. Većina današnjih indie timova koristi besplatnu verziju Unity-a iz razloga sto su u takvoj situaciji da u početku ne mogu izdvojiti velike količine novaca za skupe licence drugih engine-a.

Još jedan od razloga za odabir Unity-a je njegova fleksibilnost, to objašnjava i njihov moto „Write once, deploy antwhere“, odnosno mogućnost da se s istim kodom i datotekama mogu izgraditi aplikacije za razne platforme. Spomenuta vrsta fleksibilnosti je jedinstvena i to je ono zbog čaga je Unity poseban.

Kako je razvoj igre poprilično kompliciran i težak proces, dobro je imati neku vrstu podrške. Bila ona u obliku dokumenta, video zapisa (eng. video tutorial), foruma. Još jedan razlog za odabrat Unity je njegova ogromna podrška. Osim izvrsne dokumentacije, Unity zajednica djeluje nevjerojatno aktivno. Stotine tisuća programera koristi Unity i mnogi od njih doprinose diskusijama na cijelom web-u.

Također, u Unity je ugrađeno nekoliko fenomenalnih značajki kao sto su: Collision detection, physics simulation, pathfinding, particle systems, draw call batching, shaders, the game loop, i mnogi drugi. Sve što trebate je napraviti igru i iskoristiti te mogućnosti na najbolji mogući način [11].

# 4. MEHANIKE IGRE

Mehanike igre su u osnovi način na koji igrač komunicira s igrom. Ovdje govorimo o pravilima, zapletu, ciljevima, izazovima i načinu na koji igrač treba komunicirati s njima. Mehanika igara pomaže u igranju pružanjem konstrukcije metoda ili pravila stvorenih za interakciju igrača. Iako postoje razne teorije u vezi s mehanikom igara, one su prisutne u gotovo svim video igrama. Glavna uloga dizajnera igara je osmisliti mehaniku igre koja je dovoljno angažirana da ili zabavi igrača ili mu pruži utjecajno i vrijedno iskustvo. Sve mehanike igre kreirane su u Unity-u. Interakcija igrača s igrom omogućena je pomoću ugradbenih komponenta unutar Unity-a. Unity komponente koje su korištene u ovom radu su kamera (eng. camera), audio (eng. sound), objekt u igri (eng. gameobject), paneli (eng. canvas), animacije (eng. animation), skripte i mnogi drugi. Pomoću kamere igraću se na ekranu prikazuje scena igre. Komponente za zvuk omogućuju mu da čuje zvukove u igri i uobičajeno se postavljaju na kameru igrača. Objekti sami po sebi su statički dio igre, ali kad se na njih doda skripta koja npr. reagira na korisnički input i upravlja tim objektom tada ona postaje dinamički dio igre s kojim igrač može vrišti interakciju. Sve komponente su smještene u scenu. U igri može biti vise scena te je uobičajeno podijeliti igru u vise scena. Promjena između scena odvija se preko skripte, a poziv na skriptu vrši se preko korisničkih ulaza (eng .input).

## 4.1 Prijava i registracija igrača

Za prijavu i registraciju igrača kreirana je nova scena (eng. scene). Na sceni su kreirani paneli korisničkog sučelja (eng. panel). Prvi panel je za prijavu igrača, a drugi za registraciju. Panel za prijavu na sebi sadrži oznaku (eng. label), tekstualni okvir (eng. textfield) koji je predvedeni za upis privatnog ključa pomoću kojeg će se svaki igrač prijaviti u igru i gumb (eng. button) koji služi za pozivanje funkcije za prijavu. Funkcija provjerava ispravnost privatnog ključa te šalje zahtjev na pametni ugovor, više o tome u idućem poglavlju. Osim panela na scenu su postavljeni modeli (eng. models) iz igre. Modeli su preuzeti s Unity trgovine te su poslagani po sceni da pojačaju doživljaj igre. Prvi dojam

igre je jako bitan te iz tog razloga je scena postavljena na način da vizualno uvede igrača u radnju igre. Na slici 4 prikazan je izgled scene kod prijave igrača u igru.

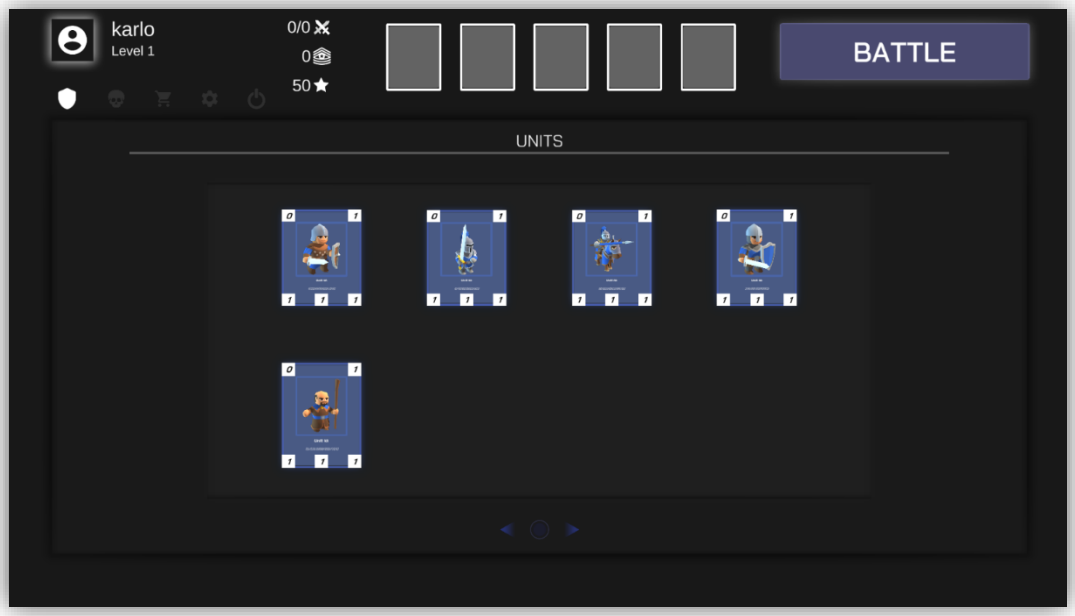

*Slika 4: Prijava igrača (Izvor: autor)*

Panel za registraciju sličan je panelu za prijavu. Ovaj panel uz tekstualni okvir za upis privatnog ključa sadrži okvir za upis igračevog imena. Prilikom registracije igrač mora popuniti sve okvire s podacima. Podaci se verificiraju te ukoliko su zadovoljavajući poziva se funkcija za registraciju. Registracija igrača je na način da se preko privatnog ključa i imena igrača kreira novi igrač na pametnom ugovoru. Ime koje igrač unese zapisat će se na pametni ugovor te igrač nema mogućnost naknadne promijene imena. Po privatnom ključu pamtiti će se kreirani igrač kako bi se izbjegla mogućnost ponovne registracije. Ekran registracije prikazan je na slici 5.


*Slika 5: Registracija igrača (Izvor: autor)*

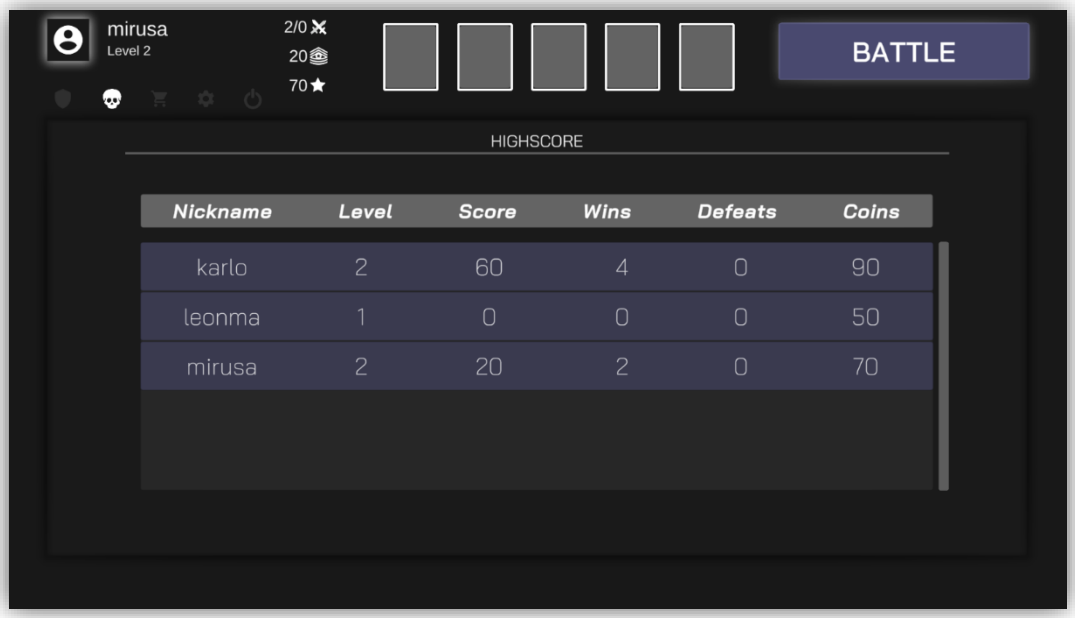
### 4.2 Glavni izbornik

Nakon prijave ili registracije, dolazi glavni izbornik. Ovaj izbornik je centar igre jer se preko njega može pristupati svom drugim scenama u igri. Funkcionalnosti glavnog izbornika su podijeljene po panelima. Paneli glavnog izbornika se dijele na prikaz svih likova (eng. units), liste najboljih rezultata (eng. highscore), kupnja novih likova i postavke igre. Prije učitavanja scene glavnog izbornika potrebno je povezati se na Photon poslužitelj. Povezivanje se ostvaruje putem Photon API koda koji je objašnjen u idućim poglavljima. Nakon povezivanja na poslužitelj, pozivaju se funkcije za interakciju s pametnim ugovorom. Te funkcije šalju zahtjev na pametni ugovor te dohvaćaju podatke s njega. Svaka funkcija ima svoju svrhu. Prva funkcija koja se poziva je za dohvaćanje igračevih statistika. Statistike igrača su ime igrača, razina, broj pobjeda i poraza, razina bodova i novac. Nakon dohvaćanja statistike se šalju na korisničko sučelje te se prikazuju na panelu na vrhu ekrana kao sto je prikazano na slici 6.




*Slika 6: Statistike igrača (Izvor: autor)*

Iduća funkcija je za dohvaćanje svih likova igrača. Igrač može imati neograničen broj likova. Svaki lik u igri ima svoju statistiku od koje su najznačajnije razina života , brzina i jačina napada. Navedena tri atributa utječu na sami ishod bitke. Igrač prije ulaza u bitku s drugim igračem mora odabrat pet likova s kojima će s boriti. Svaki lik ima svoje prednosti i mane te je jako ključno da se odabere pet likova koji daju dabar balans između atributa. Nakon što se učitaju svi likovi igrača potrebno je dohvatiti listu najboljih rezultata (eng. highscore). Za dohvaćanje se koristi slična funkcija te se podatci učitavaju na sučelje i prikazuju u tablici na panelu najboljih rezultata.

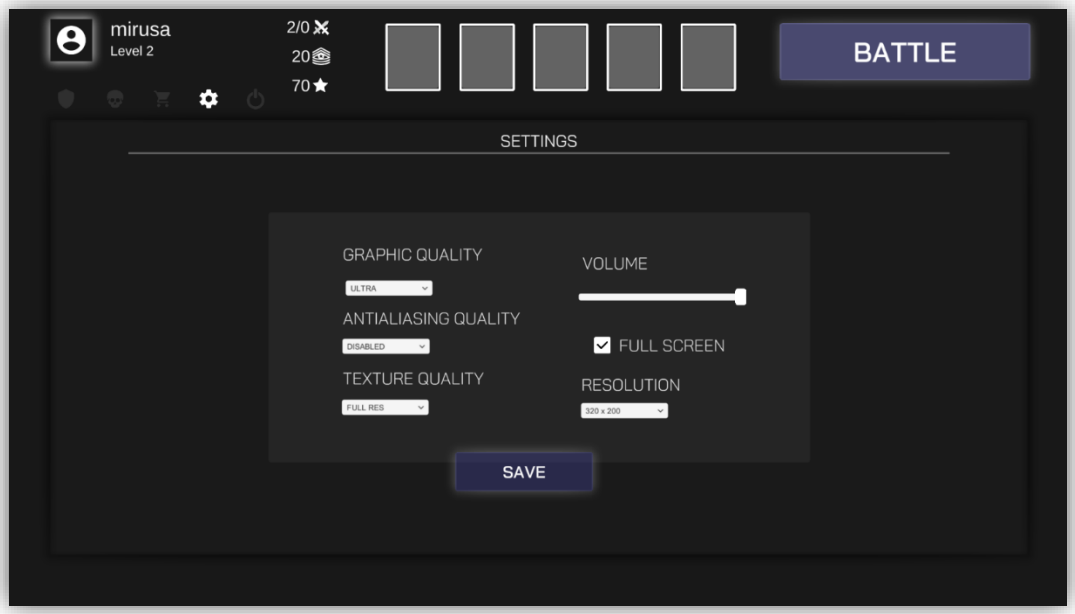



*Slika 7: Lista najboljih rezultata (Izvor: autor)*

Prethodne funkcije su dohvaćale podatke s pametnog ugovora i prikazivale ih u sučelju. Iduća mogućnost je kupnja novih likova. Za to je kreiran novi panel i na njega su poslagane sekcije sa škrinjama. Škrinje su poredane od slabih prema jačima. Svaka škrinja ima svoju cijenu, a ta je cijena prikazana ispod svake škrinje na panelu. Sve cijene su u Eterima jer se radi o pametnom ugovoru na Ethereum mreži. Pri kupnji škrinje poziva se funkcija koja kreira transakciju prema pametnom ugovoru. Spomenuta transakcija mijenja stanje ugovora pa iz tog razloga je potrebno izdvojiti dodatan iznos za gorivo koje će transakcija potrošiti. Vise o transakcijama i potrošnji goriva u idućim poglavljima.


*Slika 8: Prikaz škrinja za kupovinu (Izvor: autor)*

Zadnji panel u glavnom izborniku su postavke. Igrač ima mogućnost promijene grafike, kvalitete tekstura, promijene rezolucije i slično. Postavke se spremaju u osobne postavke (eng. player preferences) lokalno na računalo igrača. Postavke ostaju sačuvane prilikom izlaza iz igre te se učitavaju kad se igra pokrene. Dodatno na glavnom izborniku postoji tipka za izlaz iz igre na alatnoj traci pri vrhu ekrana. S alatne trake moguće je mijenjati panele koji su prethodno objašnjeni.




*Slika 9: Postavke (Izvor: autor)*

### 4.3 Scena bitke

Najviše vremena uloženo je u kreiranje scene bitke. Radi se o bitci između dva igrača. Svaki igrač ima svojih pet likova koje može postavljati na scenu te tri kule. Cilj bitke je uništiti sve tri kule protivnika, a pobjednik je onaj tko prvi uništi sve kule protivnika. Nakon pobjede igrač dobiva jednog novog lika. Za izradu scene bitke potrebno je kreirati komponentu terena (eng. terrain). Teren je Unity komponenta koja omogućuje kreiranje realističnih terena. Komponenta daje mogućnost crtanja tekstura, dodavanja drveća (eng. tree), trave (eng. grass), kreiranje reljefa i slično. Za potrebe ovog rada na teren su dodane teksture trave, blata i stijene. Na teren su poslagani modeli planina, drveća i ostalih rekvizita. Kreirani teren sa svim modelima izgleda kao na slici 10.

*Slika 10: Teren s modelima (Izvor: autor)*

Nakon kreiranog terena, potrebno je kreirati objekt koji će upravljati mehanikama igre. Za potrebe ovog rada kreirana je skripta koja će upravljati igrom. Kako je igra na mreži, odnosno igra za više igrača, svaki igrač će pokretati istu simulaciju igre. Preko mreže će se slati samo igračev ulaz i to samo postavljanje likova na teren. Igrač lika može postaviti na teren, ali samo na svoju polovicu. U tom slučaju simulacija igre mora biti deterministička jer se očekuje da se za svaki ulaz dobije isto stanje izlaza. Dolazimo do problema jer Unity nije determinističko razvojno okruženje. Iz tog razloga potrebno je u projekt uključiti dodatke za determinizam.


*Slika 11: Učitavanje podataka za determinizam (Izvor: autor)*

U nastavku je opisano nekoliko glavnih izazova pri kreiranju deterministične simulacije. Prvi je izbjegavati računanje s pomičnim zarezom (eng. floating point) jer ta računanja ne daju isti rezultat različitim računalima. To je potrebno izbjegavati što je vrlo teško jer su to temeljna numerička računanja koja se nalaze svuda u razvojnom okruženju. Sva računanja u igri moraju biti s cijelim brojevima za što postoje razne biblioteke koje se mogu uključiti u projekt. Zatim treba obratiti pažnju na nasumičnost. Sva nasumičnost u simulaciji mora biti preko istog sjemena (eng. seed) kako bi osigurali determinizam. Također, puno razvojnih okruženja koristi povratne pozive (eng. callbacks) koji nisu deterministički. Na primjer u Unity-u je potrebno osigurati da se logika igre izvršava u istom rasporedu u svim skriptama te se preporučuje izbjegavati korutine (eng. corutine) [12]. Za potrebe deterministične simulacije u projekt je uključena biblioteka za poziciju lika „SimpleVector“. Radi se o biblioteka za izračunavanje vektora. Pozicija lika na sceni reprezentirana je koordinatama x i y. Za sve promjene pozicije i izračunavanje udaljenosti koristi se ova biblioteka koja radi si cijelim brojevima kako bi se osigurao determinizam.

```
public struct SimpleVector2
{
    public int x;
    public int z;

    16 references
    public SimpleVector2(int x, int z)
    {
        this.x = x;
        this.z = z;
    }
    public readonly static SimpleVector2 zero = new SimpleVector2(0, 0);
    public readonly static SimpleVector2 one = new SimpleVector2(1, 1);
    public readonly static SimpleVector2 right = new SimpleVector2(1, 0);
    public readonly static SimpleVector2 left = new SimpleVector2(-1, 0);
    public readonly static SimpleVector2 forward = new SimpleVector2(0, 1);
    public readonly static SimpleVector2 back = new SimpleVector2(0, -1);
```

*Slika 12: Biblioteka za izračunavanje udaljenosti (Izvor: autor)*

Funkcija za dodavanje lika na scenu prikazana je na slici 13. U igri postoji red izvrdavanja korisničkih ulaza. Svaki zahtjev postavljanja novog lika na scenu ubacuje se u strukturu red te se tako i izvršava. Kod kreiranja lika postavlja mu se početna pozicija i dodaje se u grupu igrača koji ga je kreirao.

```
2 references
private void CreateHero(HeroWaitingList heroWaitingList, int groupIndex, Group myGroup, Group otherGroup)
{
    SimpleVector2 heroPosition = heroWaitingList.position;
    HeroManager newHero = CreateHero(heroWaitingList.index, groupIndex, heroPosition, otherGroup);
    myGroup.AddHero(newHero);
}
```

*Slika 13: Dodavanje lika na scenu (Izvor: autor)*

Funkcija koja implementira listu čekanja kao argument prima indeks lika, njegovu poziciju i vrijeme čekanja. Vrijeme čekanja za kreiranje lika je dvije sekunde. Nakon isteka tog vremena lik se postavlja na teren te može napadati protivnika.

```
1 reference
private void AddMyHeroToWaitingList(int index, SimpleVector2 position, IEnumerator wait)
{
    if (!MyGroup.HeroesWaitingList.ContainsKey(GameTime + 1) && !MyGroup.HeroesWaitingList.ContainsKey(GameTime + 2))
    {
        uiManager.HeroCreateTime = 2.0f - secondsPass;
        StartCoroutine(wait);
        AddMyHeroToWaitingList(index, GameTime + 2, position);
    }
    else
    {
        long addTime = 3;
        while(MyGroup.HeroesWaitingList.ContainsKey(GameTime + addTime))
        {
            addTime++;
        }
        uiManager.HeroCreateTime = addTime - secondsPass;
        StartCoroutine(wait);
        AddMyHeroToWaitingList(index, GameTime + addTime, position);
    }
}
```

*Slika 14:Funkcija za implementaciju liste čekanja (Izvor: autor)*

Kada se lik postavi na teren tada on traži najbližeg protivnika. Kao cilj može biti lik protivnika ili njegova kula. U nastavku je prikazana funkcija za traženje najbližeg protivnika i postavljanje cilja lika. Ova funkciju poziva svaki aktivni lik na sceni.

```
2 references
private GameObjectManager GetTargetTowerOrHero(HeroManager newHero, SimpleVector2 heroPosition, Group otherGroup)
{
    GameObjectManager target = GetTargetTower(heroPosition, otherGroup.Towers, out int distance);
    HeroManager otherHero = GetTargetHero(newHero, heroPosition, otherGroup.Heroes, out int heroDistance);
    if (otherHero && distance > heroDistance)
    {
        distance = heroDistance;
        target = otherHero;
    }
    return target;
}
```

*Slika 15: Funkcija za traženje najbližeg protivnika (Izvor: autor)*

Iduća bitna funkcija je ona koja implementira primjenu pozicije lika. Lik mijenja poziciju na način da nade najbližeg protivnika te se pomiče prema njemu. Kako su koordinate pozicije cijelog broja, napravljene su animacije kretnje da se ne primijeti zaostajanje (eng. lagg) kod mijenjanja pozicije. Funkcija postavi novu poziciju na staru i okreće lika da gleda u smjeru nove pozicije.

```
1 reference
public void MoveTo(SimpleVector2 position)
{
    updtateTime = 0.0f;
    oldPosition = newPosition;
    Position = position;
    newPosition = new Vector2(Position.x, Position.z);
    lookAtPosition = new Vector3(TargetObject.Position.x, 0.0f, TargetObject.Position.z);
    if((lookAtPosition - transform.position).sqrMagnitude > 5.5f)
    {
        lookAtPosition = new Vector3(Position.x, 0.0f, Position.z);
    }
}
```

*Slika 16: Pozicija lika (Izvor: autor)*

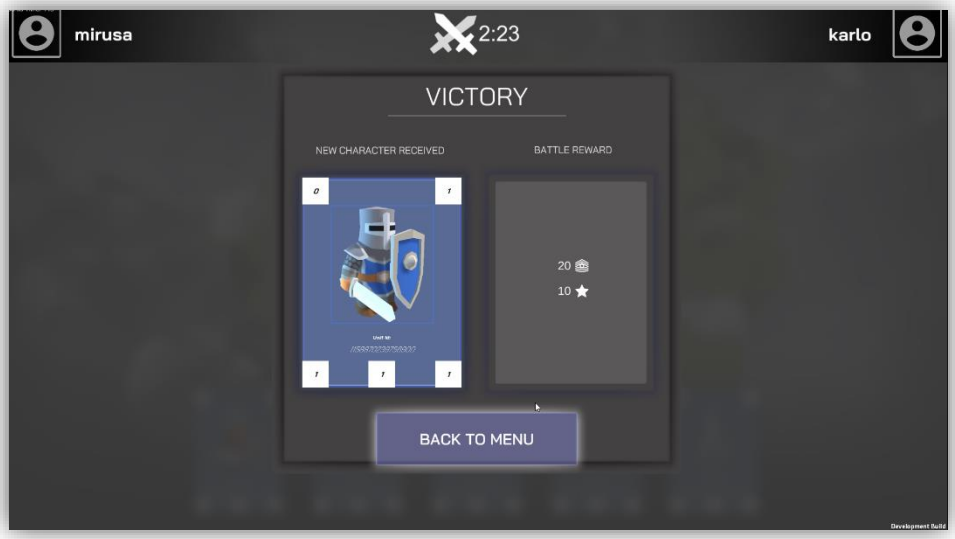
Kada lik dođe blizu protivnika, on ga napada. Napad je implementiran tako da se iz statistike lika dohvati jačina napada te se ta vrijednost oduzima od trenutne razine života protivnika, bio on lik ili kula.

```
if (hero.TargetObject.Damage(hero.Force))
{
    dynamicPositions.Remove(hero.TargetObject.Position);
    hero.TargetObject.RemoveFromGroup(group2);
    hero.RemoveTargetObject();
    if(group2.Towers.Count == 0)
    {
        gameIsEnd = true;
        StartCoroutine(OnWinOrLose(group2.Index != GroupIndex));
    }
}
```

*Slika 17: Napad na protivnika (Izvor: autor)*

Igra završava kad se unište sve kule protivnika. Nakon završetka igre na pametni ugovor se šalje transakcija. Igrač koji je pobijedio šalje transakciju u kojoj javlja kako je pobijedio te dobiva novog lika i povećavaju mu se statistike. Igrač koji je izgubio ne dobiva

novog lika nego mu se samo za mali iznos povećavaju statistike. Panel koji se prikaze kad igrač pobjedi u bitci prikazan je na slici 18.




*Slika 18: Pobjednički panel (Izvor: autor)*

# 5. UMREŽAVANJE

Razvijanje bilo koje vrste igre je izazovno, pogotovo ako se radi o igri putem mreže. Igra sa više igrača najčešće se igra preko interneta no može biti i lokalna preko LAN mreže. Umrežavanje je komplicirano, iz razloga jer ne postoji zlatna sredina arhitekture mreže i većina igara koristi kombinaciju različitih tehnika ovisno o igri i ograničenjima. Prva velika odluka koju treba donijeti prilikom dizajniranja kako će igra funkcionirati na mreži je „State vs. Input“. Da bi omogućili umrežavanje koje se temelji na mreži, podaci se moraju slati svim igračima na mreži. Dolazimo do pitanja što poslati? Postoje dva uobičajena pristupa i oni će znatno oblikovati razvoj igre.

Kod slanja stanja koristi se autorativni poslužitelj (eng. authoritative servers) koji prima stanja te ih šalje svim igračima na mreži. Na primjer. Igrač A pošalje pomak (2,1) na autorativni poslužitelj. Autorativni poslužitelj dobiva zahtjev i kaže igraču A da se pomakne na (2,1). Igrač A dobiva odgovor i pomiče se na (2,1). Ostali igrači dobivaju istu informaciju te na ekranu pomiču igrača A na poziciju (2,1) [13].

Kod slanja korisničkih ulaza, ne šalju se stajna nego se šalju ulazi. Na primjer: Igrač A šalje pomak (2,1) svim igračima, zatim igrač A dobiva odobrenje od svih igrača i pomiče se na (2,1). Na ekranu ostalih igrača igrač A se pomiče na poziciju (2,1).

U usporedbi s prethodnim primjerom glavna razlika je u tome sto umjesto slanja pozicije (stanja) igrača A jednostavno se šalje njegov potez (ulaz). U drugom primjeru, da bi igrači imali isto stanje igre, logika igre mora biti deterministička. Tj. s istim ulazom treba proizvesti isto stanje igre.

| | Slanje stanja | Slanja ulaza |
|---|---|---|
| Mrežni promet | Uglavnom visok | Uglavnom nizak |
| Ponovno spajanje | Relativno lako, igra samo treba obnoviti zadnje stanje igre od autorativnog poslužitelja | Relativno teško. Potrebno je pamtiti svaki ulaz i kad se igrač ponovno spoji tad se svi ulazi ponovno izvrše na igračevoj strani |
| Veličina datoteke ponovnog spajanja | Velika | Mala |
| Mogućnost varanja | Relativno teško, pogotovo ako je autorativni poslužitelj na fizičkom poslužitelju | Relativno lako |
| Deterministička logika | Nije potrebna | Potrebna |
| Tipični žanr igre | MMO | RTS |

*Tablica 1: State vs. Input [13]*

### 5.1 Photon PUN

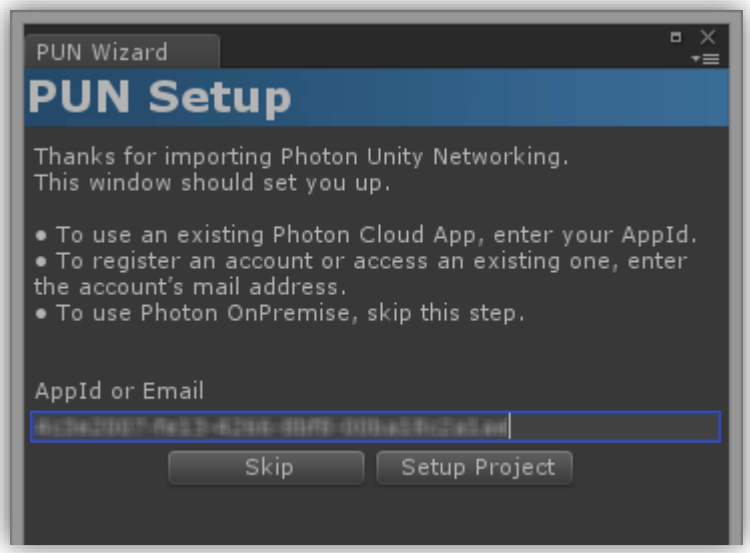
Photon je alat za razvoj igara za više igrača u stvarnom vremenu koji je brz, lagan i fleksibilan. Photon se sastoji od SDK-a poslužitelja i više klijentskih paketa za razne platforme.

Photon Unity Network (PUN) je Unity paket za kreiranje igara za vise igrača. Dostupan je puni izvorni kod, tako da se ovaj paket može prilagoditi tako da podržava bilo koju vrstu igre za više igrača. Ovaj je paket kompatibilan s uslugom Photon Cloud koja pokreće Photon poslužitelj. Struktura paketa je podijeljena u tri sloja: visoka razina (eng. high level), srednja razina (eng. second level) i niska razina (eng. low level). Visoka razina je PUN kod koji implementira komponente specifične za Unity, kao sto su mrežni objekti,

RPC i mnogi drugi. Druga razina sadrži logiku za rad s Photon poslužiteljem, povezivanje igrača u sobe (eng. rooms) i povratni pozivi (eng. callbacks). Najniža razina su DLL datoteke koje sadrže sterilizaciju, protokole i slično [14].

### 5.2 Integracija u Unity

Photon Unity Networking (PUN) je zaista jednostavno postaviti. Potrebno je skinuti i uključiti u Unity projekt paket PUN koji se može skinuti na assets store-u. Pri uključivanju PUN-a u projekt pojavit će se prozor kao na slici ispod.




*Slika 19: PUN Setup [14]*

Na prozoru je potrebno unijeti AppId. To je identifikacijski kod aplikacije na Photon poslužitelju. Aplikacija se može kreirati besplatnom registracijom na stranicama Photon-a. Nakon uspješnog postavljanja Photon-a u projekt, izgenerirat će se datoteka naziva „PhotonServerSettings“. Datoteka sadrži konfiguraciju poslužitelja te se u njoj mogu podešavati razne postavke [15].

### 5.2.1 Spajanje na Photon poslužitelja

Za spajanje na PUN poslužitelja potrebno je pozvati funkciju „ConnectUsingSettings“. PUN koristi povratne pozive (eng. callbacks) preko kojih daje odgovor da li je konekcija uspjela.

```
0 references
private IEnumerator Start()
{
    while (true)
    {
        UpdateConnecting();
        yield return new WaitForSeconds(1.0f);
    }
}
1 reference
private void UpdateConnecting()
{
    if (Application.internetReachability == NetworkReachability.NotReachable)
    {
        Debug.Log("No internet -> UpdateConnecting() -> LoadLogin");
        LoadLogin();
    }
    else if (Application.internetReachability != NetworkReachability.NotReachable && !PhotonNetwork.IsConnected)
    {
        PhotonNetwork.ConnectUsingSettings();
    }
}
```

*Slika 20: Spajanje na poslužitelja (Izvor: autor)*

### 5.2.2 Ulazak u sobe

Nakon povezivanja na PUN poslužitelja, može se pokušati pridružiti postojećoj sobi (eng. room) ili stvoriti vlastitu. Sljedeći isječci koda prikazuju pozive metoda za stvaranje i ulazak u sobu. Ulazak u sobu neće uspjeti ako nema slobodnih soba ili ne postoji niti jedna soba na poslužitelju. Metoda „OnJoinRandomRoom“ pokušava se spojiti s već postojećom sobom te ukoliko ne uspije poziva se funkcija „OnJoinRandomFailed“. Unutar te funkcije implementiran je dio koda za kreiranje nove sobe za dva igrača. Nakon kreiranja sobe igrač ulazi u sobu i čeka da se soba napuni kako bi igra mogla početi.

```
10 references
public override void OnJoinedLobby()
{
    PhotonNetwork.JoinRandomRoom();
}
14 references
public override void OnJoinRandomFailed(short returnCode, string message)
{
    string name = "Room_" + UnityEngine.Random.Range(1, 1000);
    PhotonNetwork.JoinOrCreateRoom(name, new RoomOptions() { MaxPlayers = 2 }, TypedLobby.Default);
    Debug.Log("OnJoinRandomFailed() Room name: " + name);
}
```

*Slika 21: Ulazak u sobe (Izvor: autor)*

### 5.2.3 RPC

Jedna značajka koja PUN izdvaja od ostalih Photon paketa je podrška za poziv udaljene procedure (RPC). RPC su pozivi metoda na udaljenim klijentima u istoj sobi. Da bi se omogućili pozivi udaljenih metoda, na funkciju se primjenjuje atribut [PunRPC]. RPC se ne izvršava na poslužitelju, nego on samo prosljeđuje poruku svim klijentima u sobi [16].

Na slici x prikazana je metoda za dodavanje lika u listu čekanja. Svi igrači u sobi pozivaju istu metodu na svojoj simulaciji. Metoda kao argumente prima indeks lika koji će se kreirati, vrijeme kad je pozvana metoda, i poziciju lika na koju će se postaviti na teren.

```
[PunRPC]
0 references
public void AddUnitToWaitingList(int index, long addTime, int x, int z)
{
    OnAddOtherHeroToWaitingList?.Invoke(index, addTime, new SimpleVector2(x, z));
}
```

*Slika 22: Dodavanje lika u listu čekanja (Izvor: autor)*

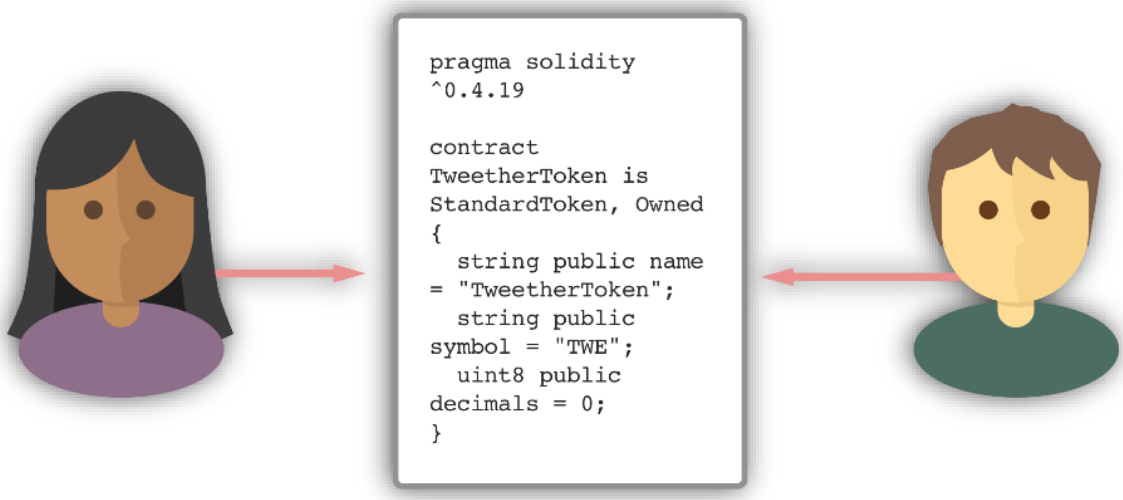
Izuzetno je važan da poredak kreiranja likova bude sinkroniziran u vremenu kako se ne bi desilo kašnjenje u simulaciji. Kašnjenje je izbjegnuto tako da igrači preko mreže šalju proteklo vrijeme igre koje tada upravitelj igre uspoređuje i sinkronizira. Igra se sinkronizira tako da se dohvati vrijeme protivnika te ako je vrijeme manje od igračevog tada se igra na igračevoj strani pauzira na trenutak. Isto tako ako je vrijeme veće tada se pauzira na protivnikovog strani. Na taj način se osigurava da su igrači usklađeni.

```
[PunRPC]
0 references
public void Ping(float gameTime)
{
    OnPing?.Invoke(gameTime);
}
2 references
public void OnPing(float otherGameTime)
{
    this.otherGameTime = otherGameTime;
}
```

*Slika 23: Razmjena vremena između igrača (Izvor: autor)*

# 6. PAMETNI UGOVOR

Pametni ugovor je samo izvršavajući ugovor, pri čemu su uvjeti između kupaca i prodavatelja izravno zapisani u linije koda. Kod i ugovori sadržani u njima postoje u distribuiranoj mreži blockchain. Kod je taj koji kontrolira izvršavanje, a transakcije se mogu pratiti, ali ne i mijenjati. Pametni ugovori dopuštaju obavljanje transakcija i sporazuma između različitih, anonimnih stranaka bez potrebe za središnjim tijelom vlasti ili trećom stranom. Pametne ugovore prvi je put predložio 1994. Nick Szabo, američki informatičar koji je 1998. godine izumio virtualnu valutu zvanu "Bit Gold", i to punih 10 godina prije izuma bitcoina. Szabo je pametne ugovore definirao kao računalni protokol transakcija koji izvršava uvjete ugovora. Želio je proširiti funkcionalnost elektroničkih metoda transakcija, poput POS-a (prodajnog mjesta), na digitalno područje.




*Slika 24: Pametni ugovor [17]*

## 6.1 Kreiranje Pametnog ugovora

Prije same izrade pametnog ugovora bitno je shvatiti kako se oni izvršavaju na Ethereum blockchain platformi. Za izvršavanje koda na Ethereum je razvio EVM „Ethereum Virtual Machine“. To je virtualna mašina koja pokreće kod napisan u pametnom ugovoru. Kako bi se omogućilo pisanje koda za EVM, kreiran je Solidity. To je jezik za pisanje pametnih ugovora na Ethereum-u. Objektni jezik opće namjene razvijena na samom vrhu Ethereum virtualne mašine. Bas kao i drugi objektni jezici, Solidity koristi klase (ugovore) i metode koje ga definiraju. Solidity može poslužiti za proizvoljne proračune, ali glavna mu je svrha slanje i primanje digitalnih tokena i stanja kroz transakcije. Na Ethereum mreži svaki pametni ugovor obraduje jedan rudar (eng. miner) i rezultat te operacije je blok koji je dodan u Ethereum blockchain. Kako bi rudari bili nagrađeni za svoj trud, izvršavanje bilo kojeg pametnog ugovora na EVM zahtjeva fiksno plaćanje, zvano gorivo (eng. gas). Za svaki pametni ugovor treba se odrediti količina goriva koju će potrošiti pri samom kreiranju. Sto je pametni ugovor složeniji to će mu trebat vise goriva.  Za potrebu ovog rada kreiran je pametni ugovor naziva „EpicCryptoBattle“. U tom ugovoru definirana je lista igrača i njihovih likova (eng. characters) [18].

### 6.1.1 Reprezentacija igrača u ugovoru

Svaki igrač reprezentiran je putem strukture. Struktura u pametnom ugovoru je prilagođena vrsta koja u svojoj definiciji može sadržavati stanja i ponašanja. U radu je kreirana struktura „Plater“ sa stanjima. Dakle, ima svoje ime „name“, trenutni level „level“, broj pobjeda „wins“, poraza „defeats“, broj ostvarenih bodova „score“ i novac „coins“. Kako ta struktura reprezentira pojedinog igrača, idući korak je kreirati vise igrača. To se radi na način da se u ugovoru definirana lista igrača. Primjer koda prikazan je u nastavku.

```
struct Player
{
    string name;
    uint32 level;
    uint32 wins;
    uint32 defeats;
    uint score;
    uint coins;
}
Player[] private players;
```

*Slika 25: Struktura statistike igrača (Izvor: autor)*

Također, svaki igrač ima svoje likove (eng. characters). Pomoću njih igrač može stupiti u borbu s ostalim igračima. Svaki lik ima svoj jedinstven identitet „id“, level „level“, tip „unitType“, rijetkost „rareType“ i statistiku: razina života „health“, brzina „speed“ i jačina napada „attack“. Isto kao i za igrače definirana je lista likova te svaki lik pripada točno jednom igraču.

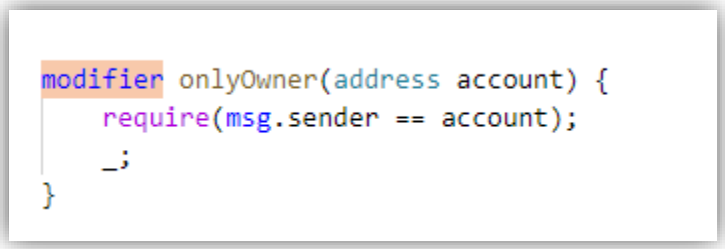
```
struct Unit
{
    uint id;
    uint32 level;
    uint unitType;
    uint rareType;
    //skills
    uint32 health;
    uint32 speed;
    uint32 attack;
}
Unit[] private units;
```

*Slika 26: Struktura statistike lika (Izvor: autor)*

### 6.1.2 Modifikatori i mapiranje

Modifikatori se koristiti za mijenjanje ponašanja funkcija. Na primjer, mogu automatski provjeriti stanje prije izvršavanja funkcije. Oni su nasljedna svojstva ugovora i mogu se nadjačati izvedenim ugovorima.

Sama definicija modifikatora vrši se putem ključne riječi „modifer“. To su funkcije koje se uglavnom definiraju jednom te se koriste na drugim željenim mjestima, odnosno funkcijama. U ovo radu definiran je jedan modifikator koji se koristi u više funkcija. U niže navedenom primjeru funkcija provjerava da li adresa pošiljatelja „msg.sender“ odgovara adresi koja je predana kao argument funkcije. Ukoliko ne odgovara prekida se izvođenje funkcije, te ako adresa odgovara onda se tok izvođenja funkcije nastavlja nakon simbola „_“. To je poseban simbol koji nam označava da nakon njega dolazi blok funkcije iz koje je pozvan taj modifikator [19].




*Slika 27: Modifikator provjere vlasnika (Izvor: autor)*

Mapiranje djeluje kao hash tablica (eng. hash table) koje se sastoji od tipova ključeva i odgovarajućih parova vrijednosti. Oni su definirani kao bilo koji drugi tip varijable u jeziku Solidity. Mapiranja su nevjerojatno korisna za preslikavanje i često se koriste za povezivanje jedinstvenih adresa s pridruženim vrstama vrijednosti [20]. Na primjer, u ovom radu bilo je potrebno da se mapiraju adrese s odgovarajućim igračem, te s brojem likova koje igrač posjeduje.

```
mapping (address => uint) public ownerToAccount;
mapping (address => uint) ownerUnitCount;
```

*Slika 28: Mapiranje igrača i njegovih likova (Izvor: autor)*

### 6.1.3 Interne funkcije

Interne funkcije (eng. internal function) su funkcije koje se mogu pozivati te izvršavati samo ako su pozvane direktno s pametnog ugovora ili iz nasljednih ugovora. Definiraju se tako da se modifikator vidljivosti funkcije postavi na „private“ ili „internal“. U ovom radu kreirane su dvije interne funkcije, za kreiranje novog lika i za kreiranje prvih 5 likova na početku igre kada se registrira novi igrač.

```
function createCharacter(uint  _id, uint _rareType) private returns (uint){
    uint unitType = randUnitId();
    uint id = units.push(Unit(_id, 1, unitType, _rareType, 1, 1, 1)) - 1;

    unitToOwner[id] = msg.sender;
    ownerUnitCount[msg.sender]++;
    unitCount++;
    return id;
}

function createFirstFiveCharacter() private {
    for(int i = 0; i < 5; i++){
        uint randId = generateRandomId();
        randId = randId - randId % 100;
        createCharacter(randId, 0);
    }
}
```

*Slika 29: Kreiranje igrača (Izvor: autor)*

Prva funkcija prima dva argumenta; „id“ i „rareType“. Zatim kreira novog lika te ga sprema u listu svih likova. Nakon toga se zapamti indeks novo kreiranog lika te se on mapira preko adrese pošiljatelja, odnosno igrača koji ga je stvorio. Druga funkcija se poziva samo jednom za svakog igrača i to samo pri registraciji novog igrača, nakon registracije igrač dobiva pet osnovnih likova s kojim može početi igru.

### 6.1.4 Eksterne funkcije

Eksterne funkcije (eng. external function) su funkcije koje se mogu pozvati samo od treće strane, odnosno izvan ugovora. Ne može se pozvati iz samog ugovora ili bilo kojeg ugovora koji iz njega proizlazi. Eksterne funkcije imaju prednost što mogu biti učinkovitije zbog činjenice da njihove argumente nije potrebno kopirati u memoriju. Dakle, tamo gdje je moguće, preporučljivo je zadržati logiku kojoj samo vanjska strana može pristupiti eksternoj funkciji. U ovom radu postoje nekoliko eksternih funkcija, neke od njih su funkcije koje mijenjaju stanje ugovora, a neke samo čitaju stajne s ugovora. Prva funkcija sluzi za kreiranje novog igrača. Igrač se kreira tako da se pošalje transakciju u kojoj se kao podatak šalje željeno ime novog igrača. U igri se ime upisuje u tekstualni okvir (eng. textbox) te pritiskom na gumb se taj podatak šalje kroz transakciju na adresu ugovora. Na slici 30 može se primijetiti da se prije kreiranja novog igrača provjerava postoji li igrač s adresom pošiljatelja.

```
function CreateAccount(string memory _name) public {
    require(!exists(msg.sender));

    uint id = players.push(Player(_name, 1, 0, 0, 0, 50)) - 1;

    accountToOwner[id] = msg.sender;
    ownerToAccount[msg.sender] = id;
    accountCount++;
    createFirstFiveCharacter();
}
```

*Slika 30: Kreiranje profila (Izvor: autor)*

Iduća funkcija je za dohvaćanje igračevih statistika. One se dohvaćaju tako da se pomoću mapiranja dobije indeks igrača u listi te se statistika igrača vrati kroz funkciju pošiljatelju. Izvorno je funkcija trebala vratiti statistiku igrača kao strukturu, ali kako se svi podaci u transakciji zapisuju u niz heksadekadskih znakova bilo je teško iščitati podatke. Iz tog razloga podaci se vraćaju kao odvojene vrijednosti radi lakšeg iščitavanja i pretvaranja u osnovni tip podataka.

```
function getPlayerStats(address account) external view returns (string memory
                                                                ,uint32
                                                                ,uint32
                                                                ,uint32
                                                                ,uint
                                                                ,uint){
    uint id = ownerToAccount[account];
    Player storage p = players[id];
    return (p.name, p.level, p.wins, p.defeats, p.score, p.coins);
}
```

*Slika 31: Dohvaćanje igračevih statistika (Izvor: autor)*

Slika 32 prikazuje funkciju koja ima sličnu strukturu kao i prethodna, samo sto ova funkcija dohvaća statistiku lika po indeksu u listi svih likova.

```
function getUnitStats(uint id) external view returns (uint
                                                      ,uint32
                                                      ,uint
                                                      ,uint
                                                      ,uint32
                                                      ,uint32
                                                      ,uint32){
    Unit storage p = units[id];
    return (p.id, p.level, p.unitType, p.rareType, p.health, p.speed, p.attack);
}
```

*Slika 32: Dohvaćanje statistika lika (Izvor: autor)*

Posljednja funkcija koja je i jedna od najznačajnijih i najbitnijih u pametnom ugovoru je funkcija koja se poziva nakon bitke. Navedenu funkciju pozivaju oba igrača, te se ovisno o pobjedi ili porazu unaprjeđuje statistike igrača. Logika funkcije je takva da ako je igrač pobijedio u bitki, on dobije novog lika, poveća mu se broj pobjeda, osvojeni bodovi i razina novca. Slično se dešava ako igrač izgubi samo sto je povećanje u manjoj mjeri i ne dobije novog lika. Te za kraj provjeri trenutno stanje bodova igrača te ako je potrebno poveća mu nivo (eng. level) za jednu razinu.

```
function AfterBattle(bool isWin) public{
    Player storage myPlayer = players[ownerToAccount[msg.sender]];
    if(isWin){
        myPlayer.score += 10 * myPlayer.level;
        myPlayer.wins++;
        myPlayer.coins += 10;

        //get new character
        uint randId = generateRandomId();
        randId = randId - randId % 100;

        uint rareType = randRareType();
        uint unitId = createCharacter(randId, rareType);
        emit NewUnit(unitId);
    }
    else{
        myPlayer.score += 2 * myPlayer.level;
        myPlayer.defeats++;
        myPlayer.coins += 1;
    }
    uint256 levelExpMax = pow(2,myPlayer.level) * 10 * (myPlayer.level);
    if(myPlayer.score >= levelExpMax){
        if(myPlayer.level < 100)
            myPlayer.level++;
    }
}
```

*Slika 33: Funkcija poziva nakon bitke (Izvor: autor)*

### 6.1.5 Pure funkcije

Funkcije se mogu deklarirati kao pure, u tom slučaju one neće čitati ili mijenjati stanje ugovora. Te funkcije mogu koristiti druge funkcije kao sto su: revert i require za vraćanje potencijalnih promjena stanja kada se dogodi pogreška. Vraćanje promjene stanja ne smatra se "izmjenom stanja" jer se poništavaju samo promjene stanja učinjene prethodno u kodu koji nije imao modifikator vidljivosti view ili pure. Taj kod ima mogućnost vraćanja i ne nastavljanja dalje s izvršavanjem koda.

```
function generateRandomId() private pure returns (uint){
    uint rand = uint(keccak256(abi.encodePacked(now, msg.sender, units.length)));
    return rand % idModulus;
}
```

*Slika 34: Pure funkcija (Izvor: autor)*

U ovom radu definirana je funkcija za nasumično generiranje identiteta pojedinog lika. Kako je zahtjev da svaki lik mora biti jedinstven tako se u obzir pri nasumičnom generiranju identiteta uzima vise izvora entropije. Kao izvor entropije uzima se trenutno vrijeme, ukupna količina likova na pametnom ugovoru i adresa pošiljatelja, te se s tim izvorima dobije dobar izvor nasumičnosti. Zatim se koristi funkcija keccak256, koja je ugrađena u Solidity. Funkcija generira ključ (eng. hash) u duljini od 256 bitova, te se taj generirani identitet pretvara u tip podatka cijeli broj (eng. integer). Na kraju se pomoću modalnog operatora ograničava duljina identiteta na maksimum 16 znakova, odnosno brojeva.

### 6.1.6 Owner funkcije

To su funkcije u pametnom ugovoru koje može pozivati samo vlasnik (eng. owner) ugovora. Vlasnik ugovora je onaj tko je kreirao ugovor. Pri kreiranju ugovora u konstruktoru se postavlja varijabla „_owner“ na osobu koja je kreirala taj ugovor

„msg.sender“ . Ostale funkcije prikazane ispod može pozivati samo vlasnik ugovora sto je realizirano dodavanjem modifikatora „onlyOwner“. To su uglavnom funkcije vezane za podizanje iznosa s ugovora i promjenu vlasnika ugovora.

```
address private _owner;
constructor() public {
    _owner = msg.sender;
}

function withdraw() external onlyOwner(msg.sender){
    address(uint160(_owner)).transfer(address(this).balance);
}

function withdraw_ETH(uint eth_wei) external onlyOwner(msg.sender){
    address(uint160(_owner)).transfer(eth_wei);
}

function transferOwnership(address new_owner) external onlyOwner(msg.sender){
    _owner = new_owner;
}
```

*Slika 35: Funkcije za podizanje iznosa s ugovora i promjenu vlasnika (Izvor: autor)*

### 6.1.7 Eventi

Eventi (eng. event) su važan dio arhitekture Ethereum-a. Preko eventa mogu se emitirati događaji iz bilo koje funkcije pametnog ugovora pokrenute transakcijom. Oni olakšavaju komunikaciju između pametnih ugovora i njihovih korisničkih sučelja. U tradicionalnom se razvoju web-stranica pruža odgovor poslužitelja kao povratni poziv prema sučelju. U Ethereum-u, kad se transakcija izvrši, pametni ugovori mogu emitirati događaje i zapisati podatke u blockchain koji se preko sučelja (eng. frontend) može potom obraditi [21]. U dolje navedenom primjeru prikazana je deklaracija eventa. Event se deklarira s ključnom rijeci event te nakon nje naziv eventa. Eventi su slični kao i ostale funkcije te isto tako mogu primati argumente. Poziv eventa omogućuje se s kljunom rijeci

emit unutar druge funkcije. U ovome radu event „NewUnit“ emitira se pri kreiranju novog lika iz vise funkcija.

```
//deklaracija
event NewUnit(uint unitIndex);


uint rareType = randRareType();
uint unitId = createCharacter(randId, rareType);
//emitiranje eventa
emit NewUnit(unitId);
```

*Slika 36: Pokretanje evenata (Izvor: autor)*

### 6.2 Dostupne mreže za postavljanje ugovora

Blockchain mreža je infrastruktura koja aplikacijama pruža usluge knjiženja i pametnog ugovora. Prvenstveno se pametni ugovori koriste za generiranje transakcija koje se potom distribuiraju na sve ravnopravne čvorove u mreži gdje su nepromjenjivo zabilježene na mreži. Korisnici aplikacija mogu biti krajnji korisnici koji koriste klijentske aplikacije ili mrežne administratori blockchaina. U većini slučajeva, više organizacija okuplja se kao ravnopravni članovi da bi se stvorila mreža, a njihova dopuštenja određuju se nizom politika koje se dogovore s članovima kada je mreža izvorno konfigurirana [22].

Blockchain mreža može se podijeli u dvije kategorije: javna i privatna. Javna mreža dostupna je svima na svijetu. Mogu se čitati ili izvršavati transakcije na javnom blockchain-u i potvrđivati transakcije koje se izvršavaju na blockchainu.

Privatnom mrežom zapovijeda središnjim tijelom u mreži, međutim, transakcija je potpuno transparentna za sve članove u mreži. Privilegije čitanja također se mogu prilagoditi.

### 6.2.1 Glavna Ethereum mreža

Glavna mreža (eng. mainnet) je izraz koji se koristi za opisivanje kada je blockchain protokol u potpunosti razvijen i implementiran, što znači da se transakcije kripto valuta emitiraju, provjeravaju i bilježe. Za razliku od glavne mreže, pojam testna mreža (eng. testnet) opisuje kada blockchain protokol ili mreža još nije pokrenuta i ne radi u svom punom kapacitetu. Testnu mrežu koriste programeri za testiranje i rješavanje svih aspekata i značajki blockchain mreže prije nego što se uvjere da je sustav siguran i spreman za pokretanje na glavnoj mreži. Drugim riječima, testna mreža postoji samo kao radni prototip za blockchain projekt, dok je glavna mreža potpuno razvijena blockchain platforma za korisnike za slanje i primanje kripto valuta ili bilo koje druge vrste digitalnih podataka koji se bilježe na distribuiranoj mreži. Obično se prije pokretanja glavne mreže, blockchain tim postaviti početnu ponudu kovanica (eng. initial coin offering) ili ICO, početnu ponudu za razmjenu (eng. initial exchange offering) ili IEO te bilo koje drugo sredstvo koje može pomoći projektu da prikupi sredstva i poveća svoju zajednicu. Prikupljena sredstva obično se koriste za razvoj prototipa blockchain mreže koja se zatim testira tijekom testne faze. Nakon što izvrši ispravke programskih pogrešaka i ovisno o izvedbi testne mreže, tim će tada pokrenuti glavnu verziju blockchain-a, koja je u potpunosti postavljena i funkcionalna.

2017. godine mnogi su blockchain startupi odlučili izvesti ICO događaje masovnog financiranja (eng. crowdfunding). Da bi to učinili, većina njih odlučila je izdati vlastiti token ERC-20 na mreži Ethereum. Spomenuti su tokeni podijeljeni u novčanike investitora prema njihovom doprinosu tijekom ICO faze.

Nakon što se ICO financiranje dovrši i blockchain bude u potpunosti raspoređen, tim može objaviti svoju glavnu mrežu koja će imati vlastiti izvorni novčić, a ne prethodno izdani ERC-20 token. U ovom trenutku odvija se proces poznat kao zamjena matične mreže, gdje se ERC-20 tokeni zamjenjuju za kovanice novog blockchaina. Nakon završetka zamjene glavne mreže preostali tokeni se obično uništavaju tako da se mogu koristiti samo novi novčići.

Unatoč popularnosti Ethereuma i ERC-20 standarda, postoje mnoge druge blockchain platforme koje podržavaju izdavanje digitalnih tokena (npr. Stellar, NEM, NEO, TRON i Waves) [23].

### 6.2.2 Testna net

Trenutno se koriste tri testne mreže, a svaka se ponaša slično kao blockchain mreža gdje se koriste etheri i tokeni. Programeri mogu imati osobne preferencije ili omiljenu testnu mrežu, a projekti se obično razvijaju samo na jednom od njih. Valuta plaćanja Ether (ETH) na testnoj mreži nema stvarnu vrijednost i služi samo za kolaborativno testiranje na mreži.

Tri su testne mreze:

- Ropsten - dokaz o radu (eng. proof-of-work), najsličnija glavnoj mreži
- Kovan - dokaz o vlasti (eng. proof of authority), ether se ne može rudariti, nego se mora zatražiti
- Rinkeby - dokaz o vlasti (eng. proof of authority), ether se mora zatražiti

Ropsten: Blok vezan uz rad koji najviše podsjeća na Ethereum; lako možete minirati faux-Ether.

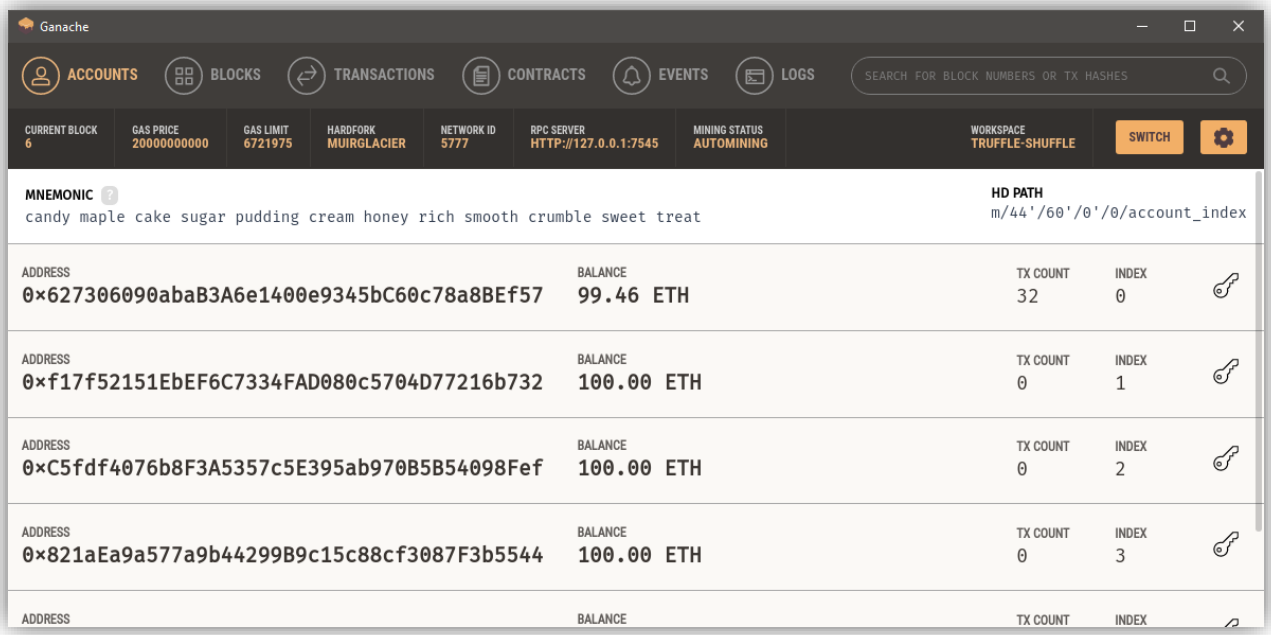
Kovan: Blokator dokaza o autoritetu, koji je započeo tim Parity. Eter se ne može iskopati; mora se tražiti.

Rinkeby: Blok vezan za potvrdu autoriteta, pokrenuo Gethov tim. Eter se ne može minirati; mora se tražiti [24].

### 6.2.3 Lokalna simulacija

Za brzi razvoj distribuiranih aplikacija koristi se Ganache. On se može koristiti tijekom cijelog razvojnog ciklusa te omogućuje razvoj, primjenu i testiranje decentraliziranih aplikacija u sigurnom i deterministčkom okruženju. U ovom radu koriti

se kao podloga za testiranje i pokretanje pametnog ugovora. Kako Ganache sadrži deset početnih računa pri čemu svaki sadrži po 100 ETH, idealan je za testiranje funkcionalnosti pametnog ugovora.



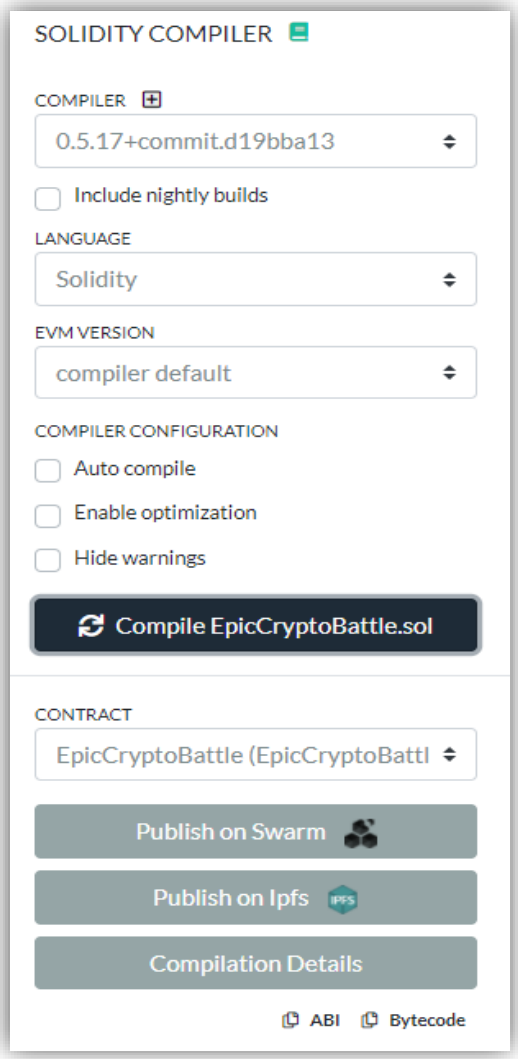

*Slika 37: Prikaz simulacije u Ganache-u [31]*

## 6.3 Postavljanje ugovora na mrežu

Na raspolaganju su nekoliko alata za pisanje, testiranje i postavljenje pametnih ugovora. U ovom radu korišten je Remix razvojno okruženje (eng. integrated development environment). Remix je prevodilac (eng. complier) u pregledniku (eng. browser) te omogućuje izgradnju pametnih ugovora u jeziku Solidity.

### 6.3.1 Prevođenje ugovora

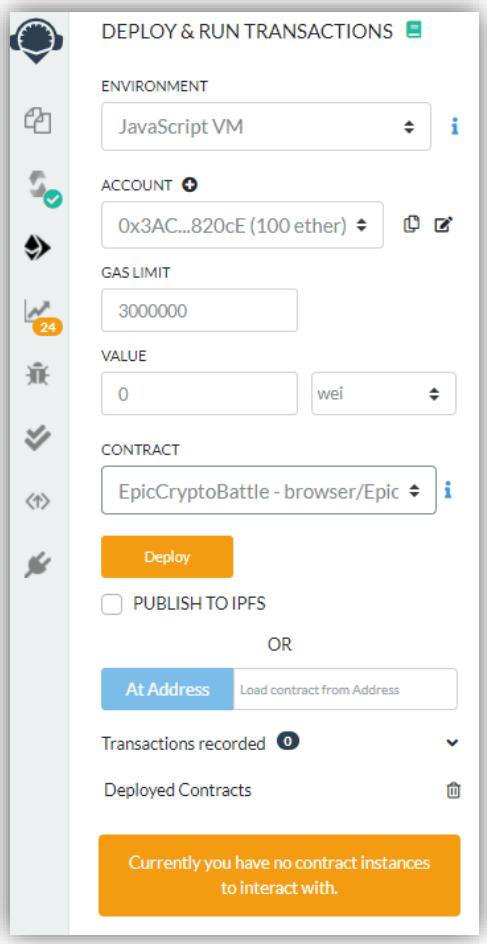
Pametni ugovor možemo prevoditi lijepljenjem (eng. copy) u uređivač (eng. editor). Prema zadanim postavkama Remix automatski prevodi kod kad se promijeni, ali to se može učiniti i ručno klikom na gumb s oznakom "Compile" [25]. Ako prevoditelj pronađe pogrešku kod prevođenja ili upozori na dio koda koji nije ispravan, prikazat će to u prozoru s lijeve strane.




*Slika 38: Prevođenje ugovora u Solidity-u (Izvor: autor)*

### 6.3.2 Postavljanje i testiranje

Nakon uspješnog prevođenja ugovora, on se može postaviti na mrežu pomoću tipke (eng. button) Deploy. Remix nam omogućuje odabir okruženje u kojem će se ugovor postaviti, odnosno daje tri mogućnosti za postavljanje ugovora: JavaScriptVM, InjectedWeb3, Web3Provider.




*Slika 39: Postavljanje ugovora na mrežu (Izvor: autor)*

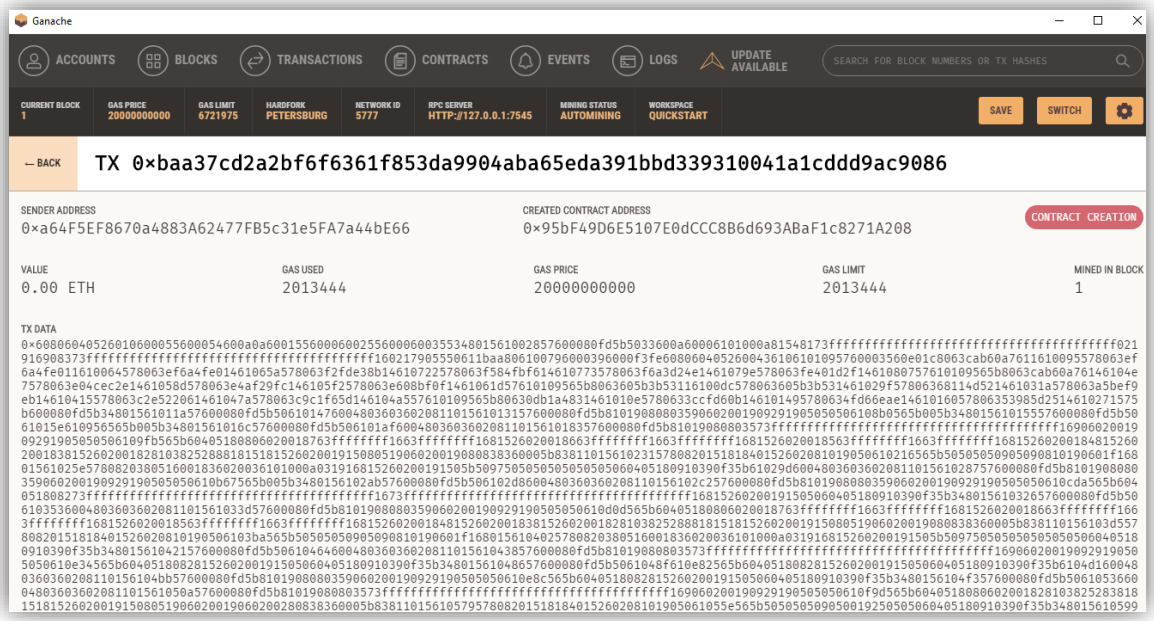
JavaScript VM nam omogućuje pokretanje ugovora izravno u pregledniku pomoću JavaScript implementacije Ethereum virtualnog stroja (EVM). Ovo okruženje je idealno za

jednostavno testiranje, ali se u njemu ne može komunicirati s drugim ugovorima ili članovima mreže.

Injected Web3 je sučelje za interakciju s Ethereum mrežom. U slučaju da se koristi MetaMask, on se sluzi implementacijom Web3 za svaku web stranicu. Navedena će opcija omogućiti implementaciju za postavljanje na testnu mrežu ili glavnu Ethereum mrežu.

Web3 Provider se izravno povezuje s Ethereum mrežom putem HTTP-a. Ako se koristi lokalna simulacija mreže ova opcija će omogućiti povezivanje na tu lokalnu mrežu.

Za početni razvoj i testiranje pametnog ugovora, koristio se JavaScript VM te u kasnijim fazama nakon testiranja ugovora korištene su i druge opcije. Nakon uspješnog testiranja i postavljanja ugovora na mrežu, potrebno je omogućiti drugim članovima da komuniciraju s pametnim ugovorom. Za to su potrebne dvije stvari: adresa ugovora (eng. contract address) i binarno sučelje ugovora (eng. application binaty interface) ili skraćeno ABI. Adresa ugovora se dobiva kad se ugovor postavi na mrežu, bila testna mreža ili glavnu Ethereum mreža. Upravo preko te adrese se može komunicirati s ugovorom. ABI je ono što nam govori koje su funkcije dostupne, koje argumente primaju i što vraćaju. ABI se generira nakon uspješnog prevođenja koda u Solidity-u, a Remix ga prikaze na sučelju kako bi ga mogli koristiti u daljnjem radu. Nakon sto smo pokrenuli lokalnu simulaciju mreže u aplikaciji Ganache, možemo prevesti i postaviti ugovor na mrežu. Uobičajeno je da se za svaku transakciju čeka neko izvjesno vrijeme da se blok napravi i postavi na mrežu, ali kad se radi na lokalnoj simulaciji tada se transakcije izvršavaju u najkraćem vremenskom razdoblju, odnosno odmah. Na slici u nastavku prikazan je blok na lokalnoj simulaciji koji sadrži podatke vezane uz postavljanje ugovora na mrežu. Možemo primijetiti da se nalaze adresa pošiljatelja, odnosno stvaralac ugovora, vrijednosti o potrošenom gorivu te adresa pametnog ugovora. Kao sto se može iščitati iz slike, tu se nalazi još podataka koji su napisani u heksadekadskom obliku te su najčešće to zapisnici (eng. log) ili eventi koji su pridruženi toj transakciji.




*Slika 40: Prikaz detalja transakcije (Izvor: autor)*

# 7. INTERAKCIJA S PAMETNIM UGOVOROM IZ IGRE

Pametni ugovori sami po sebi samo su pola priče. Da bi aplikacije bile cjelovite, pametni ugovori trebaju ih pozivati putem udaljene procedure (eng. remote procedure call) ili skraćeno RPC. Za to postoje Web3 klijenti. Klijent Web3 jednostavno je sučelje koje omogućuje interakciju s lokalnom ili udaljenom Ethereum mrežom putem HTTRP-a, IPC ili WebSocket-a. Za .NET jedini način za komunikaciju s Ethereum mrežom je Nethereum. Nethereum je složenica od .NET i Ethereum. Ova biblioteka pokušava replicirati sve funkcionalnosti koje pruža Web3 za JavaScript aplikacije koje se pokreću u pregledniku ili na JavaScript serveru [29].

## 7.1 Nethereum .NET

Nethereum je .NET integracijska biblioteka (eng. library) za Ethereum. Ona pojednostavljuje upravljanje pametnim ugovorima, poziv funkcija, transakcija, filtriranje evenata te dekodiranje i interakciju s Ethereum mrežom. Glavni razlog njenog korištenja u ovom projektu je taj sto postoji integracija za Unity te je kompatibilan sa svim glavnim operacijskim sustavima, a neki od njih su: Windows, Linux, MacOS, Android i OSX. Također, testiran je na mobilnom ureduju, osobnom računalu, Xbox i u oblaku (eng. cloud). Prije samog početka rada s Netereum-om u Unity-u potrebno je odraditi nekoliko koraka. Prvi je imati .NET SDK instaliran. Netereum koristi .NET Core ili .NET Framework od verzije 4.5.1 prema gore. Drugi korak je skinuti i uključit dodatke (eng. plugin) u Unity projekt. Svi dodatci se mogu pronaći na Internet stranicama Netereum-a [2].

## 7.2 Integracija pametnog ugovora s Netereum-om

Netereum pruža Unity-u mogućnost da postavi ugovor na mrežu i definira sve funkcije za interakciju s Ethereum mrežom, zajedno s korutinama (eng. coroutines) i dodatkom UnityWebRequest. Da bi započeli s pisanjem koda preko kojeg će se pozivati funkcije u ugovoru potrebno je deklarirati sve dodatke i definicije preko kojih će se

komunicirati s pametnim ugovorom. Primjer dodataka potrebnih u ovom radu je na slici ispod [26].

```
using Nethereum.ABI.Encoders;
using Nethereum.ABI.FunctionEncoding.Attributes;
using Nethereum.ABI.Model;
using Nethereum.Contracts;
using Nethereum.Hex.HexConvertors.Extensions;
using Nethereum.Hex.HexTypes;
using Nethereum.RPC.Eth.DTOs;
using Nethereum.Signer;
using System.Collections.Generic;
using System.Numerics;
```

*Slika 41: Prikaz korištenih dodataka (Izvor: autor)*

Postoje još i dodatne klase (eng. class) koje služe za postavljanje ugovora na mrežu, ali nisu korištene jer je to odrađeno pomoću ostalih alata. Idući korak u deklariranju pametnog ugovora je postavljanje adrese ugovora i ABI kod te s tim podatcima se kreira objekt ugovora kao sto je prikazano u nastavku.

```
public class LeeContract
{
    public static string contractABI = ...;

    private static string contractAddress = "0x95bF49D6E5107E0dCCC8B6d693ABaF1c8271A208";
    private Contract contract;


    4 references
    public LeeContract()
    {
        this.contract = new Contract(null, contractABI, contractAddress);
    }
```

*Slika 42: Kreiranje objekta ugovora (Izvor: autor)*

S novo kreiranim objektom ugovora možemo dohvatiti sve funkcije koje su definirane u njemu. Sve funkcije se dohvaćaju preko imena funkcije. Neke od metoda koje su dostupne preko Netereuma su :

- SetGasPriceFromGwei koji postavlja cijenu goriva i pretvara ga u Wei
- CreateCallInput stvara poziv koji se može koristiti za dohvaćanje podataka ili slanje transakcija
- CreateTransactionInput kreira transakcije koji se može koristiti za slanje ili potpisivanje transakcije
- DecodeInput dekodira podatke o transakciji u text
- DecodeTransactionToFunctionMessage dekodira cijelu transakciju u text
- GetCallData vraća kodirane podatke poziva sa svim informacijama o funkcijama i parametrima
- DecodeTransactionToDeploymentMessage dekodira cijelu transakciju u poruku o implementaciji [2]

### 7.3 Poziv funkcije

Pozivi funkcija upućuju se na sličan način kao bilo koji drugi Unity RPC poziv, ali je potrebno da zahtjev bude kodiran. Na primjer, da bi se moglo pozvati funkciju za dohvaćanje statistika igrača, potrebno je kreirati poziv funkcije „CallInput“. Uobičajeno je da se funkcija poziva preko adrese igrača, iz tog razloga kao argument poziva funkcije šaljemo adresu igrača.

```
2 references
public Function Function_getPlayerStats()
{
    return contract.GetFunction("getPlayerStats");
}
2 references
public CallInput CreateCallInput_getPlayerStats(
    string addressFrom
    )
{
    var function = Function_getPlayerStats();
    return function.CreateCallInput(addressFrom);
}
```

*Slika 43: Poziv funkcije (Izvor: autor)*

Nakon sto je definiran poziv funkcije, pomoću korutina šaljemo zahtjev (eng. request) na pametni ugovor. Ovdje šaljemo zahtjev pomoću EthCallUnityRequest klase na adresu mreže na koju smo povezani. U ovom slučaju to je lokalna simulacija mreže te je njena adresa „HTTP://127.0.0.1:7545“. Ukoliko je transakcija prošla tada se provjerava postoji li iznimka (eng. exception) te ako je sve u redu tada ulazimo u blok selekcije za dobivanje rezultata.

```
2 references
public IEnumerator GetOutputFunction_getPlayerStats()
{
    var address = playerEthereumAccount;

    var request = new EthCallUnityRequest(networkUrl);
    var callInput = leeContract.CreateCallInput_getPlayerStats(address);

    yield return request.SendRequest(callInput, Nethereum.RPC.Eth.DTOs.BlockParameter.CreateLatest());

    if (request.Exception == null)
    {

        //dohvacanje podataka
```

*Slika 44: Zahtjev na pametni ugovor (Izvor: autor)*

Nakon što smo uspješno dohvatili rezultat, koristimo funkciju za dekodiranje rezultata. No, vise o tome u nastavku.

### 7.4 Kreiranje transakcije

Prije nego pošaljemo transakciju na sličan način kao i poziv funkcije, potrebno je kreirati transakcijski poziv. Transakcija se emitira na mrežu, zatim je obraduju rudari (eng. miners) te ako je valjana zapisuje se na blockchain. To je operacija koja će utjecati na sve druge račune u mreži ili mijenjati stanje blokchaina. Za nju je potrebno izdvojiti neku količinu Ether-a i platiti potrošeno gorivo za njenu provedbu. Na slici ispod prikazana je funkcija za kreiranje poziva transakcije. Funkcija sluzi za kreiranje novog igrača tako da se kao argument pošalje željeno ime igrača, adresu s koje se šalje transakcija, količinu goriva i maksimalnu količinu goriva koju transakcija može potrošiti. U nastavku je prikazano kako se ova funkcija koristi za mijenjanje stanja ugovora.

```
1 reference
public Function Function_CreateAccount()
{
    return contract.GetFunction("CreateAccount");
}
1 reference
public TransactionInput CreateTransactionInput_CreateAccount(
    string addressFrom,
    string name,
    HexBigInteger gas = null,
    HexBigInteger valueAmount = null)
{

    var function = Function_CreateAccount();
    return function.CreateTransactionInput(addressFrom, gas, valueAmount, name);
}
```

*Slika 45: Funkcija za mijenjanje stanja ugovora (Izvor: autor)*

Svaku transakciju potrebno je potpisati (eng. signing). Potpisivanje transakcija jedini je način da se blockchain-u dokaže tko šalje transakciju. To je temeljni koncept u radu s Ethereum-om. Postoji nekoliko načina za potpisivanje transakcija. Prvi način je korištenjem otključanih računa. Već je spomenuto da Ganache ima deset testnih računa. Navedeni se računi generiraju otključano i pomoću njih se mogu pozivati transakcije prema pametnim ugovorima. Otključane račune mogu se generirati i pomoću specijaliziranih davatelja usluga Web3, poput pružatelja usluga novčanika (eng. wallet). Drugi način je pomoću privatnog kljuca (eng. private key). Iz sigurnosnih razloga nužno je biti oprezan s privatnim ključem jer ako dospije u krive ruke tada može doci do ogromnih problema. Ako privatni ključ dospije u krive ruke tada će taj netko imati pristup računu, moći će potpisivati transakcije predstavljajuću se kao vlasnik kljuca i upravljati svim novcem pod tim računom.

Kao i kod poziva funkcije koristi se korutina što znači da se potpise transakcija privatnim ključem igrača te se takva šalje na blockchain. Ako je transakcija izvršena tada dobijemo hash transakcije. Također, transakcije mogu imati i povratne vrijednosti. Sve povratne vrijednosti iz funkcija moraju se dekodirati o čemu je napisano u nastavku.

```
1 reference
public IEnumerator Set_CreateAccount(string _name)
{
    var transactionInput = leeContract.CreateTransactionInput_CreateAccount(playerEthereumAccount, _name, gasLimit);

    var transactionSignedRequest = new TransactionSignedUnityRequest(networkUrl, playerEthereumPrivateKey);

    yield return transactionSignedRequest.SignAndSendTransaction(transactionInput);
    if (transactionSignedRequest.Exception == null)
    {
        UnityEngine.Debug.Log("Transfered tx created: " + transactionSignedRequest.Result);

        //transakcija izvrsena
```

*Slika 46: Kreiranje transakcije prema ugovoru (Izvor: autor)*

## 7.5 Dekodiranje rezultata

Funkcije pametnih ugovora mogu vratiti jednu ili više vrijednosti u jednom pozivu. Za dekodiranje povratnih vrijednosti koristimo FunctionOutput. Primjer za to je sljedeća implementacija koja se koristi za dekodiranje povratne vrijednosti funkcije „getPlayerStats“. Funkcija nam vraća statistiku igrača te preko parametra definiramo tip izlazne vrijednosti.

```
[FunctionOutput]
4 references
public class OutputDecoder_getPlayerStats
{
    [Parameter("string", "", 1)]
    5 references
    public string Name { get; set; }

    [Parameter("uint32", "", 2)]
    8 references
    public BigInteger Level { get; set; }

    [Parameter("uint32", "", 2)]
    5 references
    public BigInteger Wins { get; set; }

    [Parameter("uint32", "", 2)]
    5 references
    public BigInteger Defeats { get; set; }
    [Parameter("uint256", "", 2)]
    5 references
    public BigInteger Score { get; set; }
    [Parameter("uint256", "", 2)]
    5 references
    public BigInteger Coins { get; set; }
}
```

*Slika 47: Parametri dekodiranja (Izvor: autor)*

Nakon definiranja klase za dekodiranje povratne vrijednosti funkcije za dohvaćanje statistike igrača, potrebno je kreirati funkciju koja će upravljati klasom za dekodiranje izlaza. Funkcija je prikazana u nastavku. Ona prima izlaznu vrijednost funkcije „getPlayerStats“ kao niz heksadekadskih znakova u obliku string te se taj niz dekodira i vrijednosti izlaza se spremaju kao parametri klase.

```
2 references
public OutputDecoder_getPlayerStats DecodeDTO_getPlayerStats(string result)
{
    var function = Function_getPlayerStats();
    return function.DecodeDTOTypeOutput<OutputDecoder_getPlayerStats>(result);
}
```

*Slika 48: Funkcija za dekodiranje povratne vrijednosti (Izvor: autor)*

Kasnije je moguće pristupati svim vrijednostima na načina da se pozove funkcija za dekodiranje te se povratna vrijednost spremi u objekt.

```
if (request.Exception == null)
{
    //dekodiranje izlaza
    var output = leeContract.DecodeDTO_getPlayerStats(request.Result);

    UnityEngine.Debug.Log(output.Name
                + ", " + output.Level
                + ", " + output.Wins
                + ", " + output.Defeats
                + ", " + output.Score
                + ", " + output.Coins);
```

*Slika 49: Dohvaćanje dekodiranih vrijednosti (Izvor: autor)*

Na isti način može se dekodirati izlaz svake funkcije u pametnom ugovoru. Samo je potrebno definirati klasu za dekodiranje te funkciju koja će upravljati tom klasom.

# 8. ZAKLJUČAK

Premda je nastao prije više od 10 godina, u posljednje vrijeme riječ blockchain se može čuti gotovo svakodnevno. Razlog tome su njegova revolucionarna svojstva i primjene, koje predvode internet u novu eru. Svjedoci smo nastajanju decentraliziranih aplikacija čiji je temelj upravo blockchain. Decentralizirane aplikacije još su u ranoj fazi, ali postoji sve veća i veća potreba za spremanjem podatka na blockchain.

U ovom diplomskom radu objašnjeno je funkcioniranje blockchaina te nastajanje blokova. Na praktičnom primjeru uspješno je prikazano rješenje spremanja i čitanja podataka u i iz Ethereum blockchaina kroz igru za više igrača. Trenutno postoji nekolicina ideja kojima bi se poboljšalo spremanje podataka u blockchain, ali svaka ideja ima svoje prednosti i mane te je na korisniku da odluči koja mu platforma najviše odgovara.

Blockchain nezaustavljivo raste, u 2015. godini tržište blockchaina je vrijedilo 315,9 milijuna, a pretpostavlja se da će do 2024. vrijediti 20 milijardi američkih dolara, s godišnjim rastom od 58.7% [27].

Uz sve pozitivne primjere blockchaina, postoji i velika negativna strana, a to je velik utrošak energije na rudarenje, odnosno spremanje podataka. Kako blockchain konstantno napreduje, ovaj problem se namjerava riješiti u bliskoj budućnosti, prelaskom s „proof of work“ načinom rudarenja, na „proof of stake“ način, gdje će manje ljudi sudjelovati u rudarenju, što će se pozitivno odraziti na okoliš.

Blockchain je još relativno nova tehnologija, koja je daleko od savršenstva, ali konstantno se radi na njenom poboljšavanju te je nedvojbeno kako će blockchain imati veliku ulogu u budućnosti informacijske tehnologije.

Za blockchain tehnologiju možemo s pravom reći da je još uvijek u svojim začecima, jer njenu prvu pravu implementaciju vidimo tek 10-ak godina unazad. Digitalne valute su svjedočile početku blockchain tehnologije, no sam potencijal je vidljiv i mnogo dalje. Samim time kako je manipulacija podacima unutar lanca onemogućena dobrom podlogom od strane tehnologije, u budućnosti bi se u školstvu i raznim drugim institucijama mogla naći ova tehnologija kao sastavni dio svakodnevice. Primjerice, kroz

povijest mnogih država zna se kako je bilo poteškoća i manipulacija u procesima prebrojavanja izbornih rezultata, blockchain će u potpunosti ukloniti tu mogućnost. Također kao primjer može se uzeti kako medicinskoj industriji dijeljenje ili gubitak zdravstvenih informacija dovodi do problema i nepovjerljivosti između korisnik i pružatelja usluga, pomoću blockchaina razmjena informacija između korisnika bi bila u potpunosti sigurna i lišena svih takvih ili sličnih anomalija u sustavu. Ovo su dakle samo trivijalni primjeri u kojima spomenuta tehnologija nudi revolucionarna i trenutno optimalna rješenja, no sama budućnost blockchaina i vrijeme koje će biti potrebno da se ono implementira u razne industrijske sustave, to će biti vidljivo u narednim godinama.

## LITERATURA


1. Unity internet stranica, https://unity3d.com/
2. Nethereum Documentation, http://docs.nethereum.com/en/latest/
3. Photon internet stranica, https://www.photonengine.com/en-US/Photon
4. Bashir I., (2017) Mastering Blockchain. Birmingham UK: Packt Publishing.Ltd
5. UpGrad blog, What is Ethereum Blockchain?, https://www.upgrad.com/blog/what-is-ethereum-blockchain/
6. Admiral Markets, Što je Ethereum?, https://admiralmarkets.com/hr/education/articles/cryptocurrencies/sto-je-ethereum
7. Solidity, https://solidity.readthedocs.io/en/v0.4.25/
8. Stark, J., Making Sense of Blockchain Smart Contracts, (2016), http://www.coindesk.com/making-sense-smart-contracts
9. Škvorc, B., Codementor Community, Developing for Ethereum: Getting Started with Ganache, https://www.codementor.io/@swader/developing-for-ethereum-getting-started-with-ganache-l6abwh62j
10. Packt, Local blockchain with Ganache, https://subscription.packtpub.com/book/data/9781839218262/5/ch05lvl1sec33/local-blockchain-with-ganache
11. Gibson, J., Introduction to game design, prototyping and development, https://ptgmedia.pearsoncmg.com/images/9780321933164/samplepages/9780321933164.pdf
12. Rouyu Sun's, Game Networking Demystified, Part 2: Deterministic (2019), https://ruoyusun.com/2019/03/29/game-networking-2.html
13. Rouyu Sun's, Game Networking Demystified, Part 1: State vs. Input (2019), https://ruoyusun.com/2019/03/28/game-networking-1.html
14. Photon Engine, PUN Intro, https://doc.photonengine.com/en-us/pun/current/getting-started/pun-intro
15. Photon Engine, Setup And Connect, https://doc.photonengine.com/zh-tw/pun/v2/getting-started/initial-setup
16. Photon Engine, RPCs and RaiseEvent, https://doc.photonengine.com/zh-tw/pun/current/gameplay/rpcsandraiseevent

17. Ludu Assets, https://ludu-assets.s3.amazonaws.com/lesson-content/26/jD5XUWTqI1aDU8oCJWTQ
18. Solidity, Introduction to Smart Contracts, https://solidity.readthedocs.io/en/v0.4.24/introduction-to-smart-contracts.html
19. Solidity, Contracts, https://solidity.readthedocs.io/en/v0.4.24/contracts.html
20. Crecenzi, D., Mappings in Solidity Explained in Under Two Minutes, https://medium.com/upstate-interactive/mappings-in-solidity-explained-in-under-two-minutes-ecba88aff96e
21. Kauri.io, Listening for Ethereum Smart Contract Events in Java, https://kauri.io/listening-for-ethereum-smart-contract-events-in-ja/760f495423db42f988d17b8c145b0874/a
22. Hyperledger Fabric, Blockchain Network, https://hyperledger-fabric.readthedocs.io/en/release-2.2/network/network.html
23. Binance Academy, Mainnet, https://academy.binance.com/glossary/mainnet
24. Compound, Hayes, G., The Beginners Guide to Using an Ethereum Test Network, https://medium.com/compound-finance/the-beginners-guide-to-using-an-ethereum-test-network-95bbbc85fc1d
25. Program the Blockchain, Testing and Deploying Smart Contracts with Remix, https://programtheblockchain.com/posts/2017/12/19/testing-and-deploying-smart-contracts-with-remix/
26. Wintellect, Steward, B., Interfacing .NET and Ethereum Blockchain Smart Contracts with Nethereum, https://www.wintellect.com/interfacing-net-ethereum-blockchain-smart-contracts-nethereum/
27. Data Shows the Growth in Value of Blockchain Markets, https://www.prnewswire.com/news-releases/data-shows-the-growth-in-value-of-blockchainmarkets-676985293.html
28. Igre.games, Clash Royale, https://igre.games/clash-royale/
29. DataSeries, Ali, Z., A Step-by-Step Guide to Nethereum, https://medium.com/dataseries/a-step-by-step-guide-to-nethereum-b98311cbe0f5
30. Miro Medium, Crash Royal, https://miro.medium.com/max/1810/1*M8nYvfG_oX7p0BAW0BjEKw.jpeg

31. Truffle Suite, Ganache Window, https://www.trufflesuite.com/img/ganache-window.png

## POPIS SLIKA

## POPIS TABLICA



## LINK NA GITHUB REPOZITORIJ: https://github.com/karlozap/Epic-CryptoWars