IZRADA VIŠEKORISNIČKE KARTAŠKE IGRE U UNITY OKVIRU

Mamut, Mateo

SVEUČILIŠTE U SPLITU

SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Stručni diplomski studij Primijenjeno računarstvo

MATEO MAMUT

Z A V R Š N I R A D

IZRADA VIŠEKORISNIČKE KARTAŠKE IGRE U UNITY OKVIRU

Split, veljača 2025.

SVEUČILIŠTE U SPLITU

SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Stručni diplomski studij Primijenjeno računarstvo

Predmet: Primijenjena umjetna inteligencija

Z A V R Š N I R A D

Kandidat: Mateo Mamut

Naslov rada: Izrada višekorisničke kartaške igre u Unity okviru

Mentor: dr. sc. Toma Rončević, prof. struč. stud.

Split, veljača 2025.

Sadržaj

Sažetak

Cilj ovog završnog rada bilo je stvaranje lako proširive igre. Naglasak je bio na modularnosti i stvaranju gradivnih blokova što bi omogučilo daljnje širenje igre novim protivnicima, igračima i napadima. Rad pokazuje spoj kreativnog razmišljanja i vještina potrebnih za izradu kvalitetnog proizvoda s mogućnošću olakšanog širenja. Za izradu rada bilo je potrebno učenje novih vještina kao što su uzorci dizajna i modularno programiranje.

U ovom radu je prikazana izrada modularne igre, povezivanje ljudskih igrača preko mreže te stvaranje pametnih suigrača, tako da se može igrati i samostalno. Igra je izrađena u okruženju za izradu igara Unity (engl. *Unity3D game engine*) i u programskom jeziku C#. Naziv igre je *Catventure*, a žanr je kooperativna kartaška igra u kojoj igrači naizmjenično odigraju svoje akcije s ljudskim ili pametnim računalnim suigračima.

Ključne riječi: pogonski alat Unity, razvojno okruženje, 3D igra, C#, umjetna inteligencija.

Summary

Multiplayer card game

The goal of this dissertation was to create an easily expandable game with the focus being modularity and creating building blocks for easier expansion of game content, which includes attacks, enemies and players. This dissertation displays the creative thinking required to create a modular system and the skills to create it. For the creation of this work, it was neccessary to learn new skills such as modular programming with the help of design patterns.

This dissertation presents the creation of a modular game, a way to connect multiple players together over the internet and intelligent bots for solo players. The game was created using the *Unity game engine* and all scripts were written in *C#*. The games name is *Catventure* and it is a co-operative turn based card game.

Keywords: Game engine Unity, editor and C# library, Unity framework, 3D game, Artificial Inteligence.

1. Uvod

U radu je obrađena izrada višekorisničke kartaške igre u Unityju. Glavni razlog odabira teme jest bilo učenje stvaranja modularnih, i lako proširivih, igara kao sljedeći korak u ozbiljnom stvaranju igara. Najbolji primjer prikaza modularnosti je kartaška igra u kojoj se karte sastoje od različitih, unaprijed već napravljenih blokova, koji se samo slažu i povezuju.

Uz samu modularnost, stvaranje višekorisničke igre bio bi veliki korak ka usavršavanju znanja i vještina njihove izrade. Razlog tomu jest njihova velika proširenost u današnjem dobu, dok sama implementacija istih nije jednostavna. Uz ovo postoje i računalni suradnici, ako se ne želi igrati s drugim ljudima. Zanimljivost je u tome što računalni suradnici imaju iste mogućnosti kao i ljudski igrači, što im znatno povećava kompleksnost.

Unutar rada obradit će se ključna poglavlja za izradu višekorisničke modularne igre i svih sustava potrebnih za pravilan rad:

- sustav akcija i efekata
- sustav statusa
- osnovno ponašanje neprijatelja
- spajanje igrača

Karte su ručno izrađene, a svi ostali modeli besplatno su preuzeti s Untiy centra za kupovanje dodataka za igru (engl. *Unity Asset Store*).

Prvo poglavlje rada opisat će razvojno okruženje Unity, koje je korišteno za izradu rada, te C# programski jezik koji je korišten pri pisanju skripti i glavne logike igre. Također bit će objašnjeno što je razvojno okruženje u kontekstu razvijanja igara i glavne prednosti istih.

Drugo poglavlje govori o sâmoj igri i njenoj unutarnjoj strukturi. Bit će opisani glavni dijelovi igre kojima se postiže modularnost te će se dati objašnjenje zašto je tako napravljeno. Ujedno će se proći i sustav za praćenje odigranih poteza koji je ključan za rad računalnih suigrača. Isto tako, bit će dani prijedlozi za unaprjeđenje iskustva igrača.

U trećem poglavlju raspravlja se o računalnim suigračima. Prikazat će se na koji način su implementirani, koja pravila prate, kako raspoznavaju različita stanja ploče i kako biraju

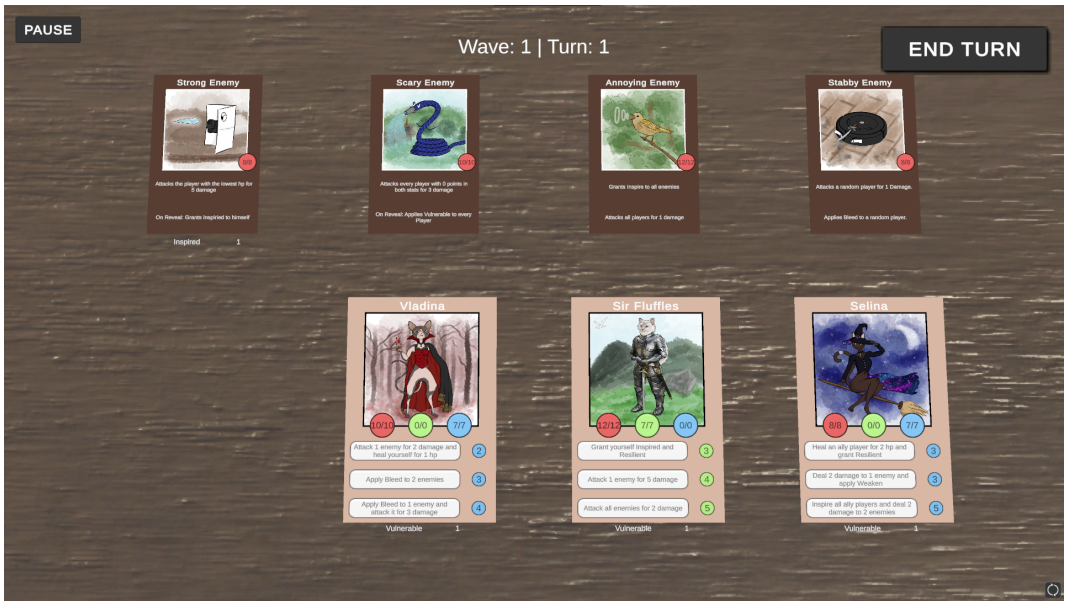
akcije. Prokomentirat će se potencijalni problemi te će biti ponuđeni prijedlozi za unaprjeđenje suigrača.

Zadnje poglavlje bavi se višekorisničkim aspektom igre: kako se igrači inicijalno povezuju, kako izgleda komunikacija igrača unutar igre te koje su promjene izvršeneda bi komunikacija bila moguća. Također, pokazat će se promjene unutar Unityja koje su bile potrebne objektima za rad na mreži.

U zaključku će biti ponuđeni primjeri za potencijalno daljnje unaprjeđenje igre i komentirat će se sveukupno iskustvo rada na projektu.

1.1. Opis i primjer igre

Igra se odvija kroz valove i krugove, a kada se pobijedi konačnog neprijatelja, igrači zajedno pobjeđuju. Ukoliko jedan od igrača izgubi sve životne bodove, svi suigrači gube i igra završava. Na početku svakog vala stvaraju se protivnici, osim u posljednjem valu kada se stvara konačni neprijatelj. Poslije stvaranja neprijatelja, igrači igraju tako da troše svoje resurse za iskorištavanje napada. Kada svi igrači iskoriste svoje dostupne resurse, protivnici napadaju te započinje sljedeći krug. Pobjedom nad trenutnim neprijateljima u valu, raste broj dostupnih resursa i životnih bodova igrača, nakon čega počinje novi val.



Slika 1: Prikaz pokrenute igre

Primjer prvog kruga (slika 1) bio bi da igrač br. 2, koji kontrolira lika "Sir Fluffles", iskoristi treću po redu akciju (koje će se objasniti dalje u radu) da napadne svakog neprijatelja

i oduzme svakom po 2 životna boda. Kako ta akcija ima cijenu od 5 bodova stamine, koja je jedna od potrebnih resursa za izvršavanje akcija i označena je zelenom bojom, igrač 2 više ne može koristiti akcije jer nema dovoljno stamine za platiti njihovu cijenu.

Poslije toga se igrač br. 3, koji kontrolira lika "Selina", sjeti da je mogao iskoristiti svoju treću akciju za davanje statusa *inspire* svim igračima, što povećava snagu svih napada za 1. To je moguće kako nema fiksnog reda igranja igrača. Na kraju je odlučio iskoristiti tu akciju i daje svakom suigraču status inspire, napada Strong i Stabby neprijatelje koji su trenutno na ploči i im oduzima 3 životna boda, što oba neprijatelja spušta na 3 životna boda. Oduzima im 3 životna boda zato što mu inspire status povećava snagu napada za 1. Poslije izvršene akcije, igrač br. 3 završava potez, jer nema dovoljno bodova za novu akciju.

Preostao je još igrač br. 1, koji kontrolira lika "Vladina", koji odlučuje napasti Strong neprijatelja s prvom akcijom te nakon Stabby neprijatelja s prvom akcijom. I jednom i drugom neprijatelju oduzima 2 životna boda, no ne liječi se kako su mu životni bodovi već puni. Nakon izvođenja dviju akcija, ostala su mu 3 boda mane (druga vrsta resursa potrebne za izvršavanje akcija uz staminu, i označena je plavom bojom) te odlučuje iskoristiti drugu po redu akciju za dodavanje *bleed* statusa na 2 neprijatelja. Ponovno bira Strong i Stabby neprijatelje. Bleed oduzima 2 životna boda na početku kruga, zbog čega će, na početku protivničkog kruga, životni bodovi za Stabby i Strong neprijatelja pasti na 0 te će biti pobijeđeni. Ovime igrač br. 1 također ostaje bez resursa.

Počinje krug protivnika, Strong i Stabby umiru zbog bleed statusa, a Scary napada Igrača 3 budući da je on jedini igrač koji je potrošio sve dostupne resurse. Annoying neprijatelj je sljedeći; daje status inspire svim protivnicima i napada sve igrače. Sljedeći krug počinje te se svi statusi uklanjaju s igrača i igrači sada ponovno igraju dalje.

2. Pogonski alat Unity

Ovo poglavlje je prilagođeni izvadak iz preddiplomskoga rada [1] te je dopunjeno novim funkcionalnostima i stavkama koje su dodane Unity razvojnom okruženju.

Unity je softver koji se koristi kao osnova za izradu igara. Daje programeru sve važne dodatke tako da bi mogao što god brže i efikasnije stvarati svoju viziju igre koju zamišlja. Omogućuje stvaranje cijelih svjetova, bilo 2D ili 3D, unutar jedinstvenog programa s jednostavnim, ali i efikasnim sučeljem. S pomoću njega se može stvoriti teren, dodati 2D ili 3D modele u svijet s već postojećom fizikom, dati im animacije, svijetlo, zvuk itd.

Rad unutar Unityja se može efektivno podijeliti na dva dijela. Jedna strana obuhvaća sam Unity te korištenje svih mogućnosti unutar njega, dok druga strana obuhvaća razvijanje programskih skripti. One se razvijaju van Unityja, i ovisno o operacijskom sustavu koristi se Microsoft Visual Studio ako se koristi Windows ili MonoDevelop ako se koristi Linux.

Prednost korištenja Unityja među ostalom je već postojanje fizike u radnom okruženju. To štedi nekoliko desetaka sati mukotrpnog stvaranja jednog od najvažnijih dijelova igre. Uz već postojanje same fizike, ona se isto može prilagoditi prema potrebama programera: kada je aktivna, koliko je snažna sila teže itd.

Unity omogućava jednostavno i bezbrižno dodavanje grafičkih elemenata u igru, bilo to 2D slike ili 3D modeli. Pomoću toga se mogu stvoriti realne 3D prizore kao i najljepša stilizirana 2D okruženja s mnoštvo detalja i pažnje.

Zvuk je među glavnim komponentama svake igre, pa se tako može i pomoću Unityja dodati. Sam zvuk, bilo to glazba ili zvučni efekti, se može stvoriti i urediti van Unityja pomoću aplikacija treće strane, a već gotov zvuk se može ukomponirati u igru.

Unity isto tako omogućava stvaranje grafičkog sučelja unutar same igre na jednostavan način. Tu se primjerice može stvoriti razne izbornike i prikazati raznu statistiku kao bodove, vrijeme i život.

Unity podržava programiranje u C#. Dolazi s unaprijed određenim skriptama koje omogućuju lagan i bezbolan početak rada te stvaranje prve igre. Uz to postoji mnoštvo primjera, Unityjev korisnički priručnik [2] i *tutoriala* napravljenih od strane korisnika.

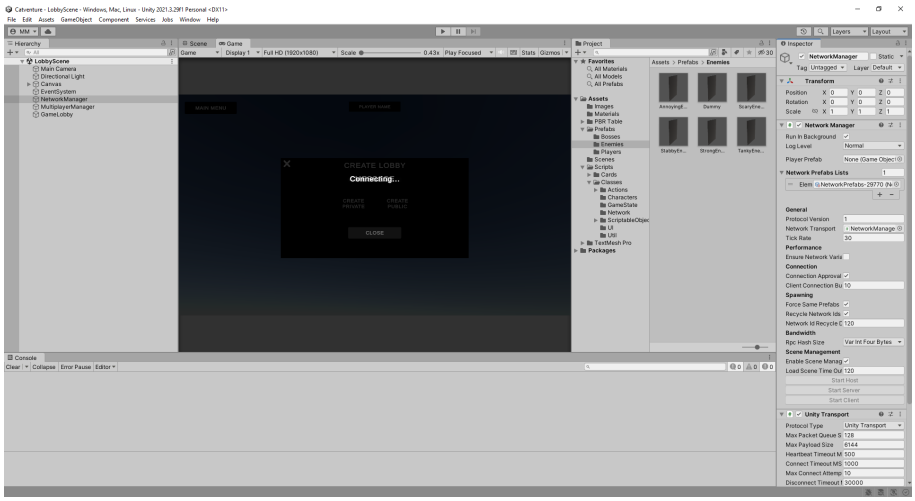
Pomoću Unityja se mogu stvarati više-platformske igre koje su namijenjene za više ope-

racijskih sustava. Ujedno se mogu stvarati više različitih vrsta projekata, uključujući:

- potpuno 3D igra,
- ortografska 3D igra,
- potpuno 2D igra,
- igra u 2D prostoru s 3D grafikom,
- igra u 2D prostoru s 2D grafikom, s nekoliko kamera različitih perspektiva.

2.1. Upoznavanje sa sučeljem

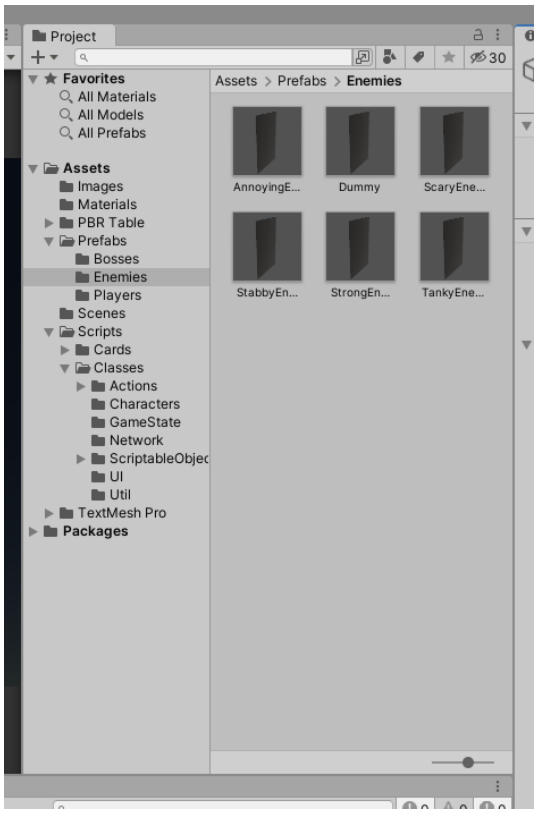
Prvo što se vidi kada se Unity pokrene je razvojno sučelje. To je ujedno jedno od najvažnijih mjesta gdje će se najviše vremena potrošiti tijekom razvojnog procesa. Glavni zaslon se sastoji od nekoliko dijelova i svaki dio se može premjestiti ili čak i maknuti prema vlastitim sklonostima i potrebama. Prikazan je na slici 2.



Slika 2: Radno sučelje

Projektni prozor (slika 3) prikazuje sveukupnu biblioteku dodataka koje su dio projekta. Tu se mogu vidjeti sve datoteke koje su vlastoručno dodane u projekt, ali i one koje su skinute s Unity online trgovine dodataka (engl. *Unity Asset Store*).

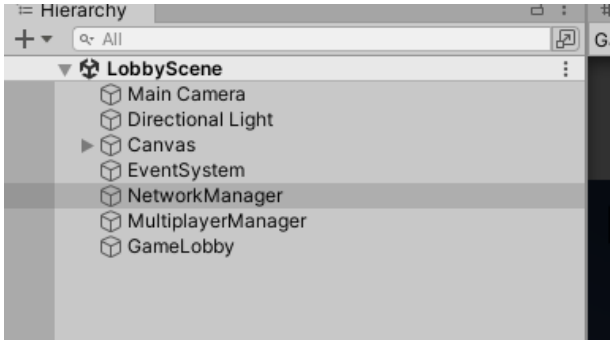
Prikaz scene je drugi najkorišteniji prozor unutar cijelog sučelja, jedino je hijerarhijski prikaz korišteniji od njega. Tu se može vidjeti sve što se do sada stvorilo u svijetu te kako izgleda. Uz to se može i manipulirati stvarima unutar svijeta, kao micati i rotirati objekte. Uz 2D prikaz, moguć je i 3D prikaz svijeta ovisno o vrsti projekta te specifičnoj komponenti



Slika 3: Projektni prozor

na kojoj se radi, npr. grafičko sučelje ili HUD (engl. *Heads-Up Display*) je uvijek u 2D perspektivi. Isto tako se može promijeniti osvjetljenje u sceni, ako projekt to podržava.

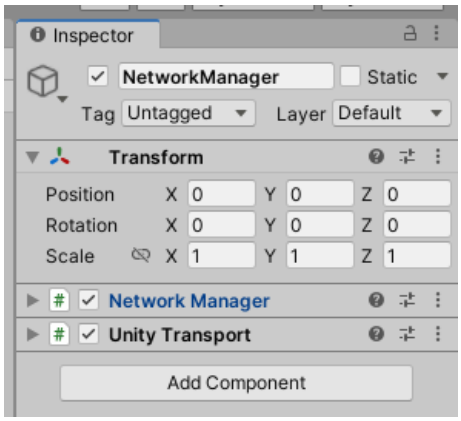
Hijerarhijski prikaz (slika 4) je najkorišteniji dio grafičkog sučelja. U prozoru su prikazani svi objekti unutar scene, a hijerarhija prikazuje kako su međusobno povezani. Unutar ovog prozora se objekti mogu izabrati, uređivati, postaviti im vidljivost, promijeniti veze te brisati i dodavati same objekte.



Slika 4: Hijerarhijski prikaz

Nadgledni prozor (slika 5) omogućuje prikaz i promjenu svojstva trenutno odabranog objekta unutar scene. Naravno, svaki objekt ne mora imati jednaka svojstva pa će se izgled

prozora promijeniti od objekta do objekta. Unutar prozora se može promijeniti veličina i položaj odabranog objekta, pogotovo ako su potrebne točne koordinate gdje objekt mora biti postavljen. Isto omogućava dodavanje fizike objektu, mijenjanje izgleda objekta, te i skripte za manipulaciju objektom.



Slika 5: Nadgledni prozor

2.2. Dodatci

Dodatci (engl. *Assets*) su svi elementi koji se mogu koristiti unutar projekta. Većinom su to modeli, ali mogu biti i glazba, slike, te bilo koja datoteka koju Unity podržava. Također se unutar Unityja mogu izraditi dodatci kao što su mikser zvukova (engl. *Audio Mixer*) i kontroler animacija (engl. *Animation Controller*). Unity također dolazi s već ugrađenim dodatcima kao što su modeli, postavke kamere, sistem čestica i efekata. Isto tako podržava sve standardne formate slika i audio datoteka.

Unutar Unityja također postoje primitivni tipovi objekata koji se dijele na 2D i 3D objekte. To su:

- Kocka, služi kao kocka, ali može služiti isto kao zid, pod te svakakve objekte, kao i senzor za pojedine stvari;
- Kugla, služi kao kugla, ali može isto služiti kao radijus za postavke pojedinih sistema, kao za sistem čestica;
- Kapsula, služi većinom za privremene modele igrača;
- Ploha, može služiti kao pod, ali isto kao i velike nevidljivi zidovi ili držači tekstura;
- Četverokut, 2D ekvivalent kocki, isto služi kao držač tekstura;

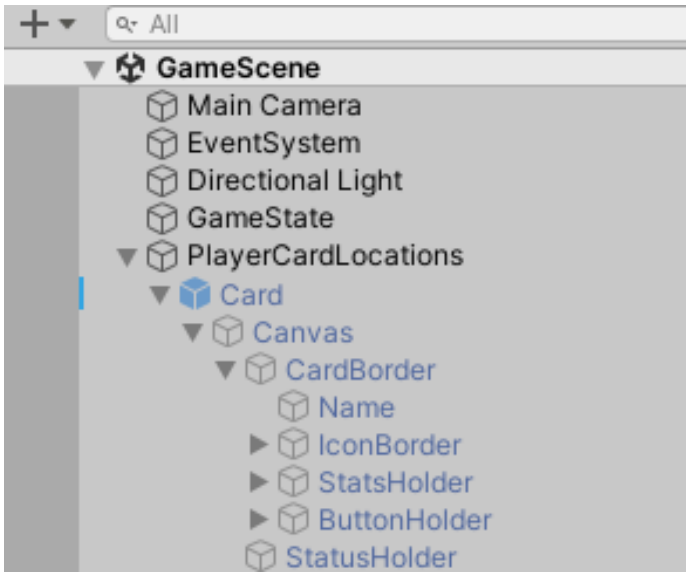
- Krug, 2D ekvivalent kugli, isto služi kao kolo;

Naravno postoje već napravljeni dodatci koje su drugi korisnici napravili. Njih možemo skinuti preko Unity trgovina za dodatke (engl. *Unity Asset Store*) te koristiti u našim projektima.

2.3. Početak rada

Za započeti novi projekt unutar Unityja nije potrebno imati godine iskustva, nego bilo tko uz malo uloženog vremena i truda može stvoriti divne stvari. Okruženje je veoma jednostavan za početnike, ali i dovoljno kompleksan za iskusnije programere. U ovom poglavlju će se pričati o najkorištenijim prozorima i glavnom pristupu izrade projekta.

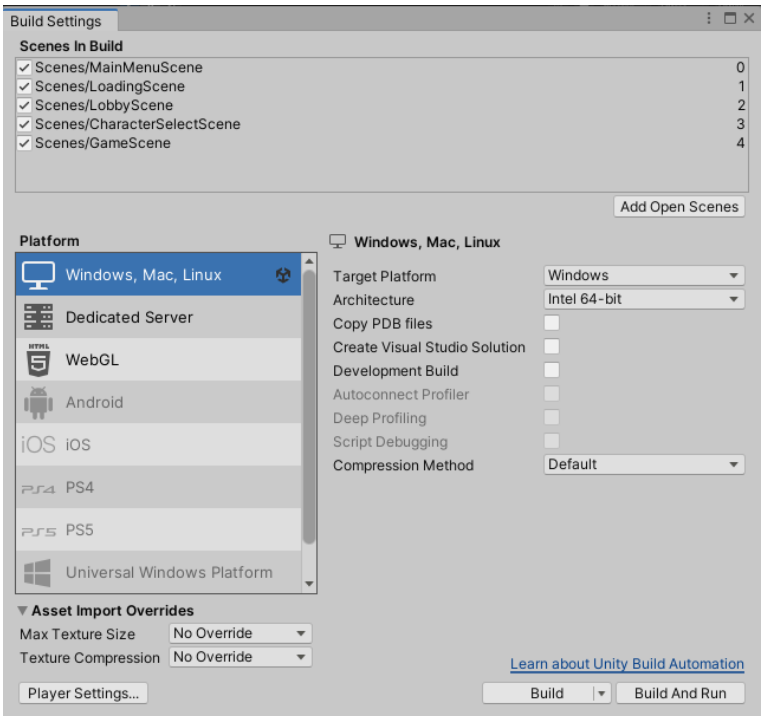
Unutar projektnog prozora mogu se vidjeti sve datoteke unutar projekta. Može ih se pretraživati prema imenu, ali čak i prema vrsti datoteke te i dodijeljenoj oznaci. Datotekama su pridijeljene ikone, ako nisu slike, radi lakšeg prepoznavanja vrsta datoteka, te se mogu iste ikone smanjiti i povećati. Može se promijeniti i način prikaza datoteka u listu i nazad. Datoteke se isto mogu označiti omiljenima (engl. *favourites*) radi lakšeg ponovnog pronalaženja. Isto tako radi lakšeg snalaženja i pronalaženja traženih datoteka, važno je da se datoteke sortiraju u svoje pojedine direktorije, kao primjerice sve zvukovne datoteke u direktoriji "Sounds". S lijeve strane je hijerarhijski prikaz svih direktorija te svih datoteka unutar direktorija. Tu se isto vide datoteke koje su označene omiljenima.



Slika 6: Prikaz odnosa roditelj dijete

U hijerarhijskom prozoru su prikazani svi objekti (engl. *Game Object*) unutar trenutne scene projekta o kojoj će se kasnije pričati. Ovdje se mogu vidjeti relacije između objekata unutar scene. Svaki objekt je dijete objekta scene, te svaki objekt može biti roditelj drugim objektima (prikazano na slici 6). Te relacije se veoma lagano dodaju pomoću *Drag and Drop* (povuci i spusti) sistema, objekt koji se želi postaviti kao dijete drugog objekta se jednostavno povuče preko željenog roditelja i ispusti. Isto tako se objekte može brisati i stvoriti u ovom prozoru, te se isto objekt može povući u projektni prozor. S time je stvoren tzv. uzorak (engl. *Prefab*). Uzorci se koriste kod objekta s posebnim svojstvima koji se mogu i programski i unutar Unityja riješiti. Koriste se tako da objektu dajemo sva željena svojstva pa se taj objekt s tim svojstvima ponovno instancira. Na taj način se može unutar Unityja pridodati efekte i svojstva, umjesto da ih ukomponiramo kôdom.

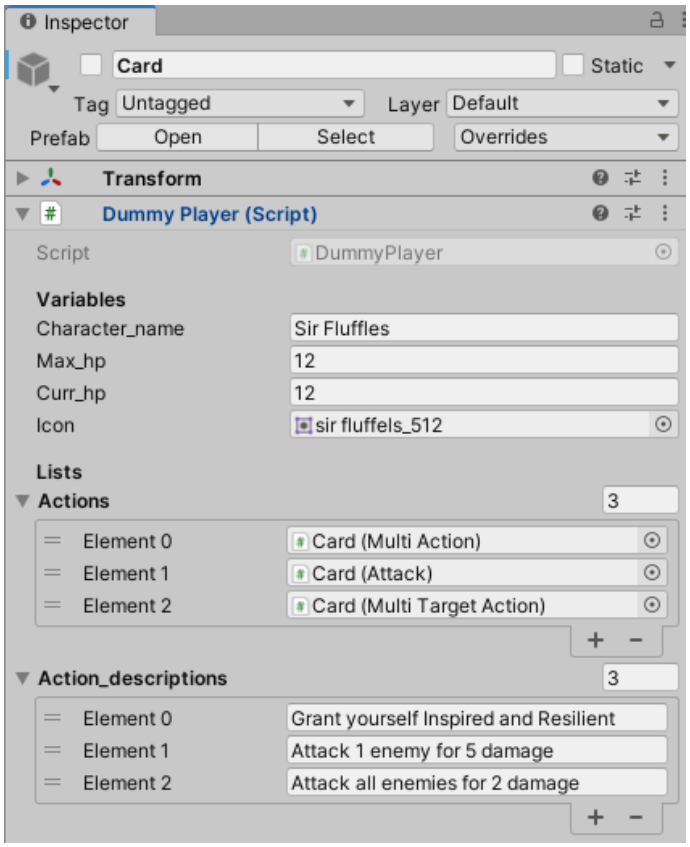
Prikaz scene pokazuje sve objekte unutar scene. Scenu se može zamisliti kao zasebnu razinu unutar igre, te isto tako ima svoje vlastite objekte. Može postojati više scena u projektu, ali samo jedna scena može biti aktivna u danom trenu. Sve stvorene i popunjene scene se moraju dodati u postavke građenja projekta (engl. *Build Settings*) i tu je svakoj sceni pridijeljen index pomoću kojeg se određena scena učitava u igru (prikazano na slici 7).



Slika 7: Postavke građenja igre

Svakom objektu u sceni se mogu mijenjati svojstva unutar nadglednog prozora (prikazan na slici 8). U to spada položaj objekta, ali i veličina i rotacija objekta, te i sam izgled i oblik

objekta. Uz ova svojstva, u nadglednom prozoru se mogu dodati druge komponente objektu, uključujući kruto tijelo (engl. *Rigid Body*)(u nastavku će se koristiti engleski naziv) i sudarač (engl. *Collider*) u svrhu mogućnosti utjecaja fizike i okoline na objekt. Na ovom mjestu se isto mogu dodati skripte koje je programer napisao, te se ovdje mogu vidjeti javne ili serijalizirane varijable i mijenjati im vrijednosti. Ovaj način mijenjanja vrijednosti varijabla se pretežno koristi radi mogućnosti mijenjanja istih tijekom testiranja projekta bez ponovnog kompiliranja kôda.



Slika 8: Nadgledni prozor sa skriptama

Uz normalni prikaz nadglednog prozora, postoji i prikaz uklanjanja greški (engl. *Debugging*). U tom načinu rada prikazane su sve informacije uvezi odabranog objekta, uključujući javne i serijalizirane varijable, ali su prikazane i privatne varijable. Vrijednost svih varijabli, isključujući privatnih, se isto mogu mijenjati u ovom prikazu radi jednostavnijeg otklanjanja poteškoća i pogrešaka.

2.4. Ključne stavke

Tijekom stvaranja igre koristit će se neke ključne stavke bez kojih bi proces stvaranja igre bio znatno teži. Dio njih je unutar samog kôda kao fizika, a dio stavki se nalazi unutar sučelja Unityja kao povezivanje objekta s kôdom.

Objekt igre (engl. *GameObject*)(u nastavku samo objekt) jest glavno sredstvo pomoću kojeg se gradi igra. Sve što se vidi unutar Unityja je objekt i on se sastoji od različitih komponenta. Komponente mogu biti standardne komponente iz Unityja, kao što je *Transform* i *Sprite Renderer*, ali isto mogu biti skripte koje je programer napisao te se mogu vidjeti u nadglednom prozoru. Objekti se mogu referencirati unutar kôda, te ako je skripta komponenta objekta iz kojeg se žele dohvatiti određeni podaci, onda se to može bez reference na objekt napraviti naredbom `GetComponent<NazivKlase>()`.Primjer referenciranja unutar kôda se vidi u primjeru 1.

```
// Unutar Unityja se spaja objekt na varijablu
[SerializeField] private GameObject objekt;
DummyClass varijabla = objekt.GetComponent<DummyClass>();
```

Ispis 1: Primjer referenci unutar kôda

Prethodno spomenute reference se može isto uspostaviti preko Unityja ako se želi povezati objekt i skripta koja nije na traženom objektu. Unutar kôda se može dodati serijalizirana varijabla, pa se preko Unityja može unutar nadglednog prozora s pomoću *drag and drop* sistema stvoriti veza. Potrebno je samo povući željeni objekt i spustiti ga unutar željenog polja unutar komponente skripte. Ova vrsta veze se često koristi unutar projekta.

Uz uzorke postoje i skriptni objekti (engl. *Scriptable object*) i veoma su slični. Dok se uzorci koriste za instanciranje istih objekata, skriptni objekti služe za dijeljenje istih podataka kroz različite skripte. Može ih se promatrati kao javne varijable, te ih svaka skripta može čitati, ali se podatci ne mogu mijenjati. Također se, za razliku od uzoraka, skriptni objekti ne mogu instancirati unutar igre.

Fizika je stavka ugrađena unutar Unityja, te je važan dio svake igre. U potpunosti je konfigurabilna i veoma lagano se ugrađuje u igru. Objektu se dodaje fizika preko komponente

Rigid Body i Rigid Body 2D. Kao što se vidi, postoje dvije različite fizike, 3D fizika i 2D fizika. Veoma su slične te je jedina razlika nedostatak treće dimenzije. Toliko su slične da se može koristiti 3D fizika u 2D prostoru i obratno.

3. Izrada kartaške igre

U ovom poglavlju iznosi se proces stvaranja projekta, počevši od najmanjih dijelova sustava, šireći se, prema složenijim. Bit će ponuđena objašnjenja za glavne ideje i ciljevi pojedinih izbora, kako ih proširiti te koji su potencijalni problemi istih.

3.1. Akcije i efekti

Akcija je najmanja vrsta događaja koja se može dogoditi unutar igre, te se sastoji samo od meta podataka, bez ikakve logike ugrađene u njih. Akcije se nadalje šire na napade, uzastopne napade i efekte, od kojih svaki posjeduje svoju zasebnu logiku. Napadi oduzimaju životne bodove jednom liku u igri, dok uzastopne akcije mogu izvršiti jednu akciju na više likova ili mogu izvršiti više akcija na istom liku, a efekti postavljaju statuse na likove.

```
public abstract class Action : NetworkBehaviour {
   public int[] cost;
   public int numTargets;
   public bool targetAllies;
   public bool targetSelf;

   public abstract void do_action(List<Character> targets,
      Character source);
   public abstract void undo_action(List<Character> targets,
      Character source);
}
```

Ispis 2: Implementacija Akcije

Same akcije nemaju logiku biranja meta, već im se prosljeđuje niz meta na kojima se treba izvršiti akcija, što se može vidjeti unutar ispisa 2. Uvijek je niz meta, neovisno o tome postoji li samo jedna ili više meta radi modularnosti kôda. Zbog toga se sve akcije izvršavaju na jednaki način, neovisno o podvrsti akcije. Sve podvrste akcije implementiraju metodu `do action` i `undo action` te se unutar njih odvija cijela logika akcije. Tako se postiže polimorfizam, a njime modularnost igre, jer će igra sve podvrste akcija vidjeti samo

kao obične akcije, te će ih izvršavati na isti način, s pomoću `do action` metode. Primjer implementacije iste za napad se može vidjeti u ispisu 3.

```
public void do_action(List<Character> targets, Character source)
{
   // Always attack first target
   if (targets.Count == 0) return;
   int damage_done = official_damage;
   foreach (Status status in source.on_attack_statuses) {
      damage_done = status.modify(damage_done);
   }
   targets[0].take_damage(damage_done);
}
```

Ispis 3: Implementacija metode do action za napad

Akcije se dijele na dodatne podvrste, uključujući:

- uzastopne akcije na više meta,
- aktiviraj efekt na sve protivnike pa izvrši uzastopne akcije,
- statusi

Statusi su podvrsta efekata te se samostalno aktiviraju, naspram ostalih akcija koje se trebaju aktivirati od strane igrača ili protivnika. Dijele se na dvije vrste: statusi koji oduzimaju životne bodove i statusi koji modificiraju štetu koja se treba nanijeti liku u igri te ostaju vezani uz likove do određenog trenutka, najčešće početka idućeg kruga. Implementacija statusa se može vidjeti u ispisu 4.

```
public abstract class Status : Effect
{
   protected string statusName;
   protected bool removeOnTurnEnd;
   public abstract int modify(int value);
```

```
    public string StatusName => statusName;
    public bool RemoveOnTurnEnd => removeOnTurnEnd;
}
```

Ispis 4: Implementacija klase statusa

3.2. Dnevnik akcija

Sve akcije koje se dogode unutar igre bilježe se u dnevnik akcija. Tako se uvijek može provjeriti kako je došlo do trenutnog stanja igre i, što je još važnije, može se vratiti u prethodna stanja igre. Glavna korist pamćenje akcija jest vraćanje do prethodnih stanja, a uz to se može i ispisati prošlost akcija radi lakšeg razumijevanja tijeka igre. Prikaz članova klase se vidi u ispisu 5.

```
public class Event
{
    public int turn;
    public bool is_player_phase;
    public Character character;
    public Action action;
    public List<Character> targets;
}
```

Ispis 5: Implementacija klase za pamćenje akcija

Sve akcije prolaze kroz dnevnik akcija te se iste ne pokreću unutar dnevnika. Akcije se pokreću nakon zapisivanja u dnevnik. Za spremanje akcije u dnevnik potrebno je proslijediti samu akciju te podatke tko je započeo akciju i na kome se akcija izvodi. Te se informacije spremaju u niz, kronološkim redoslijedom, što znači da je prvi zapis prva akcija i zadnji zapis je uvijek zadnja akcija koja se dogodila. Kako se sprema objekt klase `akcija`, može se poslati bilo koja podklasa iste, zbog svojstva poliformizma.

Za vraćanje u prethodna stanja igre, potrebno je implementirati mogućnost poništavanja akcija unutar akcije. Zbog toga je potrebno da sve akcije implementiraju metodu `undo`

`action` koja se poziva pri poništavanju akcije. Implementacija zavisi o vrsti akcije, npr. vraćanje životnih bodova ili uklanjanje statusa koji je aktivan na liku, no ujedno se i razlikuje, ovisno o tome koliko se uzastopnih akcija izvršava i na koliko likova. Kod uzastopnih akcija se obrnutim redom izvršavanja poništavaju akcije kako se ne bi došlo do drugačijeg stanja igre nego što je bio cilj. Primjer implementacije metode `undo action` je prikazan u ispisu 6. Informacija nad kojim akcijama je potrebno izvršiti poništavanje akcije se nalazi unutar samog dnevnika akcija te se pri poništavanju akcija ista briše iz dnevnika.

```
public void undo_action(List<Character> targets, Character source);
{
   foreach (Character target in targets) {
      target.on_attack_statuses.Remove(weak);
   }
}
```

Ispis 6: Implementacija metode undo action

Mogućnost za širenje igre bilo bi dodavanje grafičkog sučelja (engl. *User Interface*) za prikaz povijesti poteza, gdje bi bile vidljive sve akcije izvršene dosad sa svim potrebnim informacijama. To obuhvaća osnovne podatke uključene u zapisu dnevnika, ali i podatke o krugu i valu u kojima se odvila akcija. Grafičkim elementima olakšalo bi se razlikovanje igračevih akcija naspram protivničkih akcija.

3.3. Igrači i protivnici

Igrači i protivnici središnjica su igre te su u mnogočemu slični. I protivnici i igrači nasljeđuju klasu lika koja u sebi ima sva potrebna svojstva i metode za normalan rad (ispis 7). To uključuje životne bodove i metode za oduzimanje životnih bodova te gubljenje igre. Glavna razlika između njih je način na koji se dodatno šire različitim metodama i svojstvima.

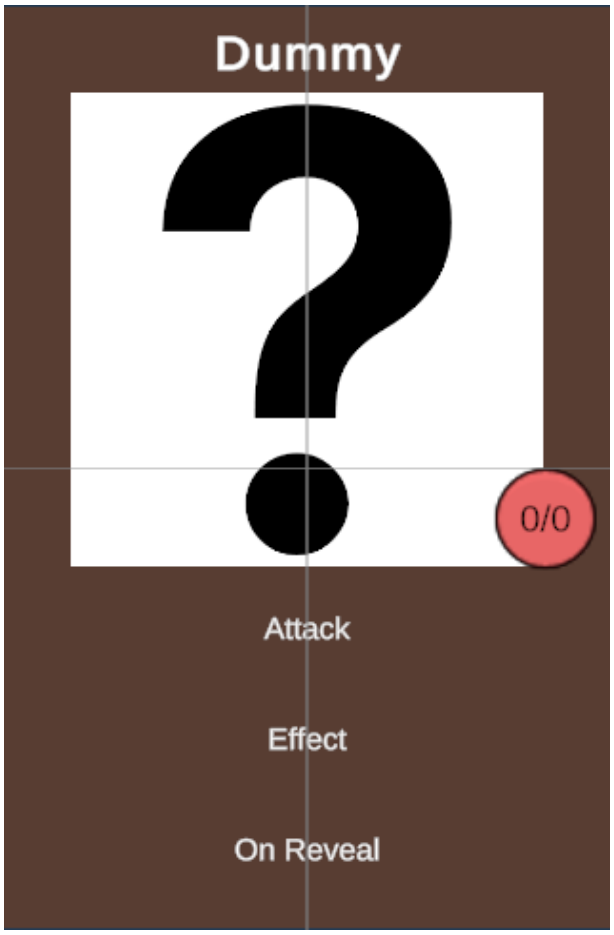
```
public abstract class Character : MonoBehaviour
{
   [SerializeField] protected string character_name;
   [SerializeField] protected int max_hp;
```

```
    [SerializeField] protected int curr_hp;
    [SerializeField] protected Sprite Icon;

    public Action[] actions;
    public string[] action_descriptions;
    public List<Status> on_hit_statuses;
    public List<Status> on_attack_statuses;
    public List<Status> on_turn_start_statuses;

    public void Awake() { curr_hp = max_hp; }

    public abstract void die();

    public int take_damage(int val) { ... }
}
```

Ispis 7: Implementacija klase lika

Igrači uz životne bodove imaju i obnovljive resurse: staminu i manu. One su potrebne za vršenje akcija. Svaka akcija ima svoju cijenu stamine i mane, a ako igrač ne može platiti cijenu, ne može ni odigrati akciju. Kada se akcija odvije, smanjuje se trenutna vrijednost dostupne stamine ili mane, ovisno o cijeni akcije. Resursi se na početku kruga osvježavaju te se na kraju vala povećava maksimalni broj dostupnih resursa, i različiti likovi imaju različite maksimalne vrijednosti istih.

Protivnici su glavni neprijatelji protiv kojih se igrač bori, a dolaze u valovima. Protivnici se sastoje isključivo od akcija koje moraju odigrati, a koje se ne mijenjaju. Bolje rečeno, isti protivnik uvijek radi istu akciju, a na njemu samome pišu uvjeti prema kojima bira metu za akciju. Primjer toga jest akcija koja oduzima 1 životni bod igraču s najvećem broju maksimalnih životnih bodova. Protivnička karta je prikazana na slici 9. Svaki protivnik u pozadini ima istu akciju za napad, a jedina je razlika implementacija biranja meta. Iznimke su naravno stavljanje statusa na likove, no ideja je još uvijek ista. S time se postiže smanjivanje potrebnog vremena za izradu novih neprijatelja kako se samo mora implementirati biranje meta.



Slika 9: Prikaz protivničke karte

Pri kraju igre dolazi i konačni protivnik koji je naprednija verzija normalnog protivnika. Dok normalni protivnici imaju maksimalno po jedan napad i jednu akciju koja dodaje statuse, glavni protivnici mogu imati više istih akcija. Uz to, ima i više života, što znači da ga je teže pobijediti. Implementacija samog glavnog protivnika je ista kao i implementacija normalnih protivnika, uz razliku da se klasificira kao *boss*, to jest kao glavni protivnik te se nalazi u drugom špilu.

Sve igračeve i protivničke karte izrađene su unutar Unityja. Napravljene su pomoću 3d objekta na kojemu se nalaze razni elementi korisničkog sučelja (engl. *User Interface*) kao što su slika lika, dostupne akcije i trenutno dostupni resursi. Sve karte su kopije jednog te istog objekta te se pomoću skripti dinamično postavljaju podatke ovisno o tome na kojeg je lika povezana karta. Primjer karte igrača se vidi na slici 10.

Slika 10: Prikaz igračeve karte

3.4. Računalni suigrači

Računalni suigrači (nadalje *botovi*) ponašaju se kao normalni igrači te unutar same igre nema nikakve razlike između njih i normalnih, živih igrača. Botovi koriste iste likove kao i igrači, a i mogu koristiti sve nove likove koji bi se dodali u igru jer ne zahtijevaju posebnu implementaciju za pojedinog lika. Jedna je implementacija za sve likove te se dijeli na dva dijela:

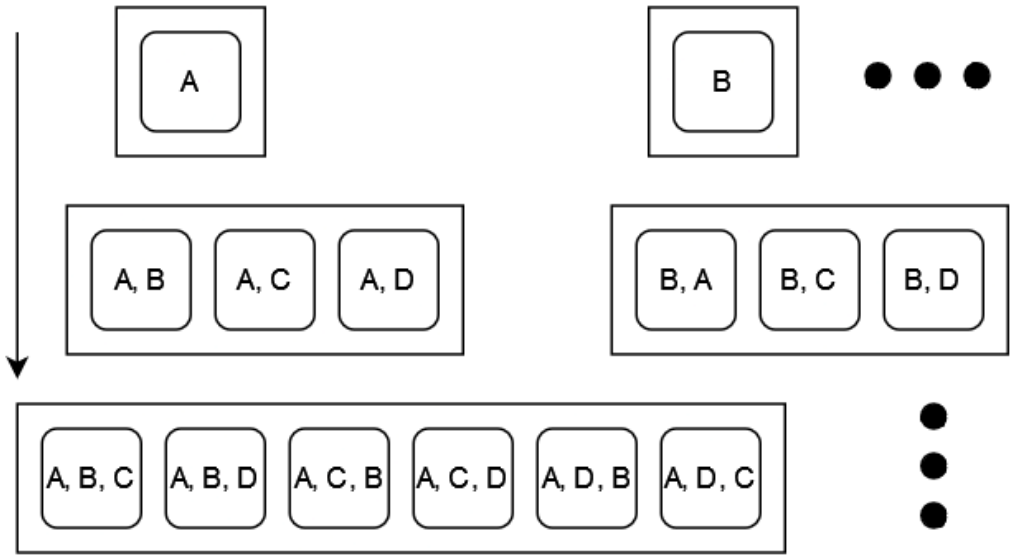
- pronalaženje svih mogućih poteza,
- pronalaženje najboljeg poteza

3.4.1 Pronalaženje svih mogućih poteza

Prvi korak za pronalaženje najboljeg poteza jest pronalaženje svih mogućih poteza. U niz se dodaje svaka akcija s odgovarajućim metama. To je jednostavno ako akcija zahtjeva samo jednu metu, no stvari postanu kompliciranе kada akcija zahtijeva više meta. Ako akcija

zahtjeva samo jednu metu, onda se akcija doda toliko puta koliko je meta i to svaki put s posebnom metom. No, ako akcija zahtjeva dvije ili više meta, onda se akcija treba dodati sa svim mogućim kombinacijama meta.

Problem je riješen rekurzijom gdje se postepeno dodaju mete. Primjer rješenja vidi se na slici 11. Proces počinje s jednom metom koja se stavlja u popis. Početni popis se duplicira i u svaku kopiju se dodaje nova različita meta. Taj proces dupliciranja i dodavanja novih, različitih meta izvodi se sve dok se ne iskoriste sve dostupne mete ili se ne dođe do traženog broja meta.



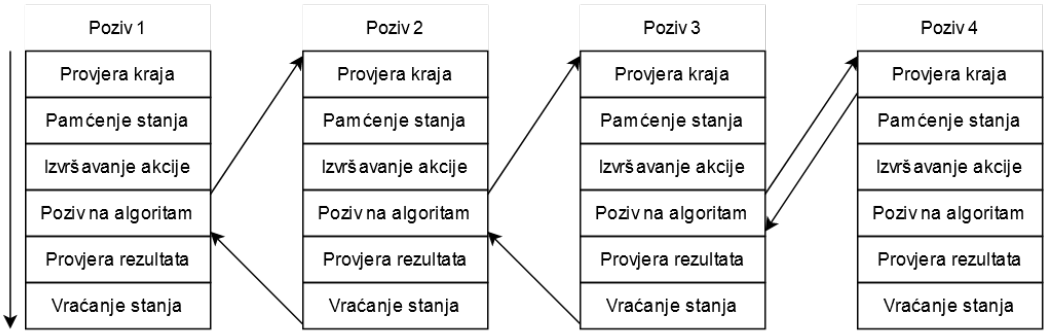
Slika 11: Primjer pronalaženja svih kombinacija meta

3.4.2 Pronalaženje najboljeg poteza

Algoritam pronalaženja najboljeg poteza izvršava se rekurzivno. Razlog tome jest pronalaženje najboljeg stanja ploče nakon što su svi potezi odigrani te pamćenje izvršenih akcija do tog stanja. Sam algoritam je podijeljen na 3 dijela, pri čemu prvi dio služi za zaustavljanje rekurzije. Tu se provjerava se da li je došlo do jednog od konačnih stanja rekurzije, što su kraj igre ili nestanak protivnika.

Drugi dio algoritma služi za simuliranje ponašanje bota. U tom dijelu bot vrši i poništava akcije u potrazi za najboljom. Bot prvo pamti trenutno stanje igre, a potom vrši sljedeću akciju. Nakon izvršenja akcije, rekurzivno se poziva algoritam i, pri završetku izvršavanja te instance, igra se vraća u početno stanje te se provjerava da li je vraćeni potez iz te instance

bolji od trenutno najboljeg poteza. Ako jest, onda ta akcija postaje nova najbolja akcija. Ta petlja se izvršava za svaki mogući potez i zaustavlja se tek kada se ostane bez mogućih poteza, odnosno, ako se ostane bez resursa potrebnih za izvršavanje akcija. Grafički prikaz se može vidjeti na slici 12.



Slika 12: Illustracija pronalaženja najbolje akcije

Zadnji dio algoritma simulira protivnički krug te taj dio algoritma ne nastavlja rekurziju. Jednako kao i u drugom dijelu, prvo se pamti početno stanje igre te se pritom oponaša početak neprijateljske faze. To uključuje aktiviranje statusa, napadanje igrača i na kraju aktiviranje statusa na samim igračima na početaku kruga. Potvrdu da uistinu postoje neprijatelji nudi prvi dio algoritma koji zaustavlja algoritam ako oni trenutno ne postoje. Tada se vraća stanje igre izraženo brojem bodova i igra se vraća u prvobitno stanje.

Algoritam trenutno ne uzima u obzir sljedeće krugove igre, već se se pri kraju trenutnog kruga zaustavlja. Razlog tomu jest mogući eksponencijalni rast vremenskog trajanja algoritma. Pri sve većem simuliranju krugova i mogućih poteza, produljuje mu se i vrijeme za izvršavanje. Za unaprjeđenje algoritma, pokušalo bi se uključiti i daljnje krugove uz nužne optimizacije za ubrzavanje pretrage akcija. Drugo moguće unaprjeđenje algoritma jest da se botovi međusobno uzimaju u obzir, odnosno, da zajedno igraju i zajedno traže najbolju akciju od svih mogućih akcija. To bi omogućilo bolje sekvenciranje akcija botova, no i dalje ostaje problem povećanja vremenske zahtjevnosti algoritma.

4. Pretvorba u višekorisničku igru

Sljedeći korak jest implementacija značajki potrebnih za višekorisnički rad. To uključuje pretprostor za igrače (nadalje *lobby*) i dodavanje logike unutar same igre koja omogućuje komunikaciju među igračima.

To je postignuto pomoću Unityjevih gotovih rješenja *Lobbies* i *Netcode for Gameobjects*. Lobbies služi za pojednostavljeno spajanje igrača preko Unityjevih servera i servisa, dok Netcode for gameobjects služi za dodavanje mrežnih funkcionalnosti objektima.

4.1. Lobby

Lobby je scena unutar igre gdje se različiti ljudi povezuju i biraju svoje likove. Sam lobby se sastoji od prikaza igrača, izbora mogućih likova i statusa spremnosti. Uz to, svaki lobby ima svoj identifikacijski broj i nekoliko postavki kao maksimalni broj igrača i privatnost lobbyja. Sama implementacija lobbyja potječe od Unityja, a spajanje igrača odvija se preko njihovih servera.

Za samo kreiranje lobbyja potrebna je dodatna scena koja ujedno služi i za spajanje na postojeće lobbyje (slika 13). Pri odabiru stvaranja lobbyja, igraču se nudi izbor naziva lobbyja i privatnosti istog. Za opciju privatnosti nudi se javno i privatno, pri čemu javno označava da bilo koja osoba može ući u lobby, dok se privatnom lobbyju može pristupiti isključivo pomoću identifikacijskog koda lobbyja.

Slika 13: Prikaz scene za pronalaženje lobbyja

Poslije stvaranja lobbyja, igrač koji ga je stvorio automatski ulazi u lobby. Ovisno o odabranoj opciji privatnosti, drugi igrači ulaze u lobby putem izbornika ili unosom koda u

tražilicu

Unutar lobbyja vide se svi trenutno povezani igrači i dodani botovi. Igrači su prikazani ikonama izabranih likova koje žele igrati, pri čemu je vidljivo i ime svakog igrača. Ako je igrač bot, onda će samo pisati "bot". Vlasnik lobbyja jedini ima mogućnost dodati i maknuti botove u lobby i mijenjati izabranog lika botovima. Vlasnik može, proizvoljno, na isti način dodati ili ukloniti igrače iz lobbyja.

Važno je naglasiti da dva igrača ne mogu izabrati istog lika za igrati. Kada igrač ili bot izabere lika za igrati, odabir lika se zaključa te ga više nije moguće izabrati. Također sam izbor likova nije ručno napravljen, nego se pri ulasku u lobby ekran povlači popis svih likova s njihovim informacijama. Primjer postavljanje izbora likova se vidi u ispisu 8. S time se postiže lakše širenje igre kako se novi lik mora samo dodati u jedan popis da bi bio dodan u igru.

```
public override void OnNetworkSpawn()
{
   playerCharacters = MultiplayerManager.Instance
      .GetAllPlayerCharacters();
   addBotButton.gameObject.SetActive(IsServer);

   foreach (PlayerCharacter character in playerCharacters)
   {
      var selectButtonInstance = Instantiate(selectButtonPrefab,
         characterHolder);
      selectButtonInstance.SetCharacter(character);
   }
}
```

Ispis 8: Metoda za postavljanje izbora likova

Kako bi se igra pokrenula, svaki igrač treba izabrati kojeg lika želi igrati i potom potvrditi da je spreman. Igrač može povući spremnost jer se izabranog lika može mijenjati samo kada igrač nije spreman. Botovi su automatski spremni te im se dodaje prvi trenutno slobodan lik čim se dodaju u lobby.

4.2. Promjene unutar igre

Da bi igra mogla raditi preko mreže, potrebno je omogućiti objektima korištenje mreže. To rješava Unityjev Netcode for gameobjects koji objektu daje upute kako komunicirati preko mreže. Da bi objekt mogao komunicirati putem mreže prvo mora naslijediti novu klasu, `NetworkBehaviour`, i mora imati dodano *Network object* kao svojstvo, unutar Unityja. Pri tome, ne mogu se svi objekti instancirati na mreži, nego samo oni koji su dodani u listu mrežnih objekata, što povećava sigurnost igre i smanjuje mogućnost ubacivanja zlonamjernih objekata u igru. Popis se može vidjeti na slici 14.

Slika 14: Popis objekata koji se mogu instancirati na mreži

Za komunikaciju između igrača koriste se RPC pozivi (engl. *Remote Procedure Calls*). RPC predstavljaja način pokretanja kôda na drugim računalima. Postoje dvije različite vrste RPC-ova: serverski i klijentski. Serverski RPC-ovi služe za izvođenje metoda na serverskoj instanci igre, odnosno, u instanci igre vlasnika lobbyja, dok klijentski RPC-ovi služe za izvođenje metoda na svim klijentskim instancama igre, što uključuje i serversku instancu u ovoj igri.

Prva metoda koja se pretvorila u RPC jest metoda za dodavanje akcije u povijest. Dodavanje akcije u povijest vrši se na serveru preko serverskog RPC-a, dok se sama akcija izvršava na svakom klijentu pomoću klijentskog RPC-a. Pri pozivanju serverskog RPC-a vrši se sva potrebna provjera valjanosti upita, čime se postiže veća otpornost igre prema zlonamjernim napadima.

```
[ClientRpc]
private void do_actionClientRpc(
   NetworkBehaviourReference characterRef,
   NetworkBehaviourReference actionRef,
   NetworkBehaviourReference[] targetsRef) {

   if (!characterRef.TryGet(out Character character)) { return; }

   if (!actionRef.TryGet(out Action action)) { return; }

   List<Character> targets = new List<Character>();
   foreach (NetworkBehaviourReference targetRef in targetsRef) {
      if (!targetRef.TryGet(out Character temp_target)) {
         return;
      }

      if (temp_target != gameCharacter) {
         targets.Add(temp_target);
      }
   }

   event_backlog.Add(new Event(character, action,
      targets, turn.Value, is_player_phase.Value));
   action.do_action(targets, character);
}
```

Ispis 9: Klijentski RPC za pokretanje akcije

Potreban je i RPC za dodavanje igrače i protivnike u svoje popise. Za igrače je potreban samo RPC, jer se igrači neće mijenjati tijekom igre, dok je za protivnike potreban i serverski RPC koji će pokretati klijentski iz sigurnosnih razloga. Uz to su dodani i RPC-ovi za propagiranje trenutnih vrijednosti resursa pojedinih igrača. Sastoje se od serverskog i klijentskog RPC-a.

Uz RPC-ove, postoje i mrežne varijable koje se dijele preko mreže svim igračima te su svakom igraču iste. Kako se koristilo Unityjevo mrežno rješenje, podatci koji se mogu slati mrežom ne smiju biti prazni, to jest, ne smiju biti *null*. Nije važno je li podatak zapravo null, važno je samo može li biti postavljen na null. To znači da se ne mogu slati objekti preko mreže, budući da isti mogu biti postavljeni na null. Zbog toga ostaju samo jednostavni podatkovni tipovi kao što su brojevi, booleani, slova i slično, ali i reference pomoću kojih se može pokušati pristupiti objektu. Jedina iznimka jest što se tekst može slati preko mreže.

Uz RPC-ove i mrežne varijable također se može provjeriti status instance unutar kôda, odnosno je li je pokrenuta instanca server. Glavna petlja igre pri početku provjerava je li trenutna instanca server, a ako je, nastavlja se izvršavati. S time se osiguralo da će se petlja samo jednom po krugu izvesti, te da server ima kontrolu nad podatcima. Unutar petlje se izvršavaju akcije botova i neprijatelja pomoću otprije spomenutih RPC-ova te se tako stanje igre propagira igračima.

Zadnja stvar potrebna za rad na mreži jest pridruživanje likova unutar igre njihovim vlasnicima. Pri instaciranju objekata, u ovom slučaju likova, na mreži, može se postaviti tko je vlasnik objekta unutar. Korist pridruživanja vlasnika za lika je to što se može limitirati interakciju igrača s tuđim kartama. Pri svakom pokretanju akcije lika se provjerava jeli je vlasnik te karte započeo akciju te ako nije, izvršavanje akcije se zaustavlja.

5. Zaključak

Cilj ovog projekta bio je stvoriti modularnu i lako proširivu igru, koja ujedno pruža i mogućnost igranja s prijateljima. Zbog iskustva na prethodnom radu, odabrano je Unity razvojno okruženje, koje je korišteno kroz čitavu izradu projekta. U radu je opisano pomoću kojih je funkcionalnosti izrađen projekt, te kako se ostvarila modularnost.

Kao razvojno okruženje, Unity omogućuje jednostavnu i brzu izradu igre, što se postiže „familijarnošću“ korisničkog sučelja i sveukupnog okruženja. Unity omogućuje da se sve radi kroz njega, što uključuje logiku igre kroz skripte, ali i animacije i stvaranje samih modela. S time se smanjuje broj potrebnih alata za rad, što je dodatan razlog pri odabiru ovog okruženja za rad.

Najproblematičniji dio implementacije bilo je stvaranje računalnog suigrača. Problem proizlazi iz činjenice korištenja rekurzivne implementacije, čime je znanto otežan proces otkrivanja pogrešaka. Uz to je trebalo paziti da se akcije poništavaju pri vraćanju igre u prvobitno stanje, što prvotno nije bio slučaj. Poteškoće su stvarale i beskonačne petlje koje bi nastajale kada bi se zaboravilo izmijeniti uvjet ili oduzeti resurs nakon izvršene akcije.

Unutar igre djelomično se obradila tema postavki, no kako je naglasak na višekorisničkom aspektu igre i samoj modularnosti, nije se temeljito obradila. Također su se spominjale postavke za mijenjanje rezolucije igre te postavke za mijenjanje jačine zvuka, koje bi se dodale nakon dodavanja zvuka projektu.

Sljedeći korak za projekt bio bi unaprjeđenje računalnog igrača tako da vidi dalje u krug te na taj način postane pametniji. Po potrebi bi se igra optimizirala na način da računalni igrač treba odigrati svoj krug u zadanom vremenu. Uz to bi se sama igra uljepšala te bi se dodale prethodno spomenute postavke i glazba.

Literatura

[1] M. Mamut, “Proceduralna igra u unity razvojnoj okolini,” https://urn.nsk.hr/urn:nbn:hr:228:082940 (posjećeno 15.8.2024.).

[2] Unity, “Unity user manual,” https://docs.unity3d.com/Manual/index.html (posjećeno 21.3.2024.).