Izrada strateške videoigre uloga u programskom alatu Unity

Štefičar, Dominik

Master's thesis / Diplomski rad

2023

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:683774

Rights / Prava: Attribution-NonCommercial 3.0 Unported/Imenovanje-Nekomercijalno 3.0

Download date / Datum preuzimanja: **2024-05-15**

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Repository / Repozitorij:

Faculty of Organization and Informatics - Digital Repository

zir.nsk.hr

dabar
DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Dominik Štefičar

Izrada strateške videoigre uloga u programskom alatu Unity

DIPLOMSKI RAD

Varaždin, 2023.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Dominik Štefičar

Matični broj: 46446

Studij: Informacijsko i programsko inženjerstvo

Izrada strateške videoigre uloga u programskom alatu Unity

DIPLOMSKI RAD

Mentor/Mentorica:

Doc. dr. sc. Mladen Konecki

Varaždin, rujan 2023

Dominik Štefičar

Izjava o izvornosti

Izjavljujem da je moj diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

Tema ovog diplomskog rada bavi se razvojem 3D računalne igre sa temom strateškog odlučivanja i igranja na poteze. Glavni dio igre sastoji se od povezane programske logike izrađene unutar alata *Unity*. Također je popratno korišten programski alat Blender zaslužan za izradu potrebnih 3D modela koji su iskorišteni za vizualnu reprezentaciju prototipa igre te u kombinaciji sa danom logikom tvore kompletnu igru. Sama igra izrađena je na principu naizmjeničnih igračevih i neprijateljskih poteza tijekom kojih se donose odluke o mogućim akcijama. Inspiracija za igru proizlazi iz šaha i kombinira elemente karata iz igra poput *Hearthstone* kartaške igre kako bi se nadodala šira apstrakcija figurama. Također nadodaje princip bacanja kocke i sličnih elemenata iz stolne igre uloga zvane *Dungeons & Dragons* što povećava razinu nepredvidljivosti i šansu da će igrat ponovno zaigrati igru ispočetka. Prototip igre reprezentira lekcija borbe protiv jednog unaprijed određenih setova karata neprijateljskih jedinica. Kod igre izrađen je na principu ponovnog iskorištenja koda i lakše proširivosti. Cilj prototipa je pobijediti sve neprijateljske jedinice prije nego one učine isto.

Ključne riječi: Unity, karte, 3D modeliranje, strateška igra, C# programiranje, razvoj videoigara, figure, igra na poteze

Sadržaj

Sadržaj iii
1. Uvod 1
2. Metode i tehnike rada 2
2.1. Unity 2
2.2 Blender 3
2.3. DoTween 5
2.4. Leonardo AI 6
3. Razrada teme 7
3.1. Kreiranje 3D modela 7
3.2. Postavljanje projekta u Unity alatu 9
3.3. Izvođenje i interakcija s igrom 10
3.4. Arhitektura projekta 15
3.4.1 Singleton uzorak dizajna 16
3.4.2 MonoBehaviour klasa 17
3.4.3 ScriptableObject klasa 19
3.4.4 Karte 20
3.4.4.1 Karte figura 22
3.4.4.2 Karte za napad 26
3.4.5 Efekti 28
3.4.6 Figure 30
3.4.7 Špil karata 31
3.4.8 Polje za karte 33
3.4.9 Polje na igraćoj ploči 35
3.4.10 Kontroleri 38
3.4.10.1 Kontroler za odabir karte napada 38
3.4.10.2 Kontroler borbe između karata 41
3.4.10.3 Kontroler za korisničko sučelje statusa figure 44
3.4.10.4 Kontroler za pojavu figura i karata 45
3.4.11 Menadžeri 47
3.4.11.1 Menadžer za karte 47
3.4.11.2. Menadžer za igraću ploču 52
3.4.11.3. Menadžer za igrača 56
3.4.11.4. Menadžer za kameru 62
3.4.11.5. Menadžer za neprijatelje 64
3.4.11.6. Menadžer za resurce 68
3.4.11.7. Menadžer za igru 69
4. Zaključak 71
Popis literature 72
Popis slika 73

1. Uvod

Kada se spominju igre uloga, vjerojatno će većina ljudi pomisliti na nešto što izgleda kao *Dungeons & Dragons* što je prva velika stolna igra koja je popularizirala ovaj žanr. Unutar igre borbeni sukobi i događaji rješavaju se upotrebom kockice, različitih atributa likova i slično. Igrači imaju sposobnost istraživanja unaprijed definiranih scenarije ili sami stvaraju nove koristeći priručnike za pravila, knjige i svoju maštu [1].

Igre uloga žanr su video igre u kojoj igrač upravlja jednim ili više izmišljenih likova i preko njih stvara interakciju sa svijetom igre. *RPG* igre postoje dugi niz godina i danas je teško definirati cijeli žanr jer postoji veliki spektar hibridnih žanrova unutar RPG žanra. Neki od glavnih atributa koji definiraju ovakvu igru su naprimjer postojanje borbenog sustava za koji se koriste izbornici s različitim odabirom vještina, napada, akcija i slično. Nadalje može postojati neki centralni zadatak kroz priču igre i sporedni opcionalni zadaci ili postojanje klasa likova koje definiraju njihove karakteristike, vještine i druge mogućnosti. Većina današnjih igra uloga sadrži jedan ili dva takva elementa u kombinaciji sa nekim drugim žanrom igre. [2]

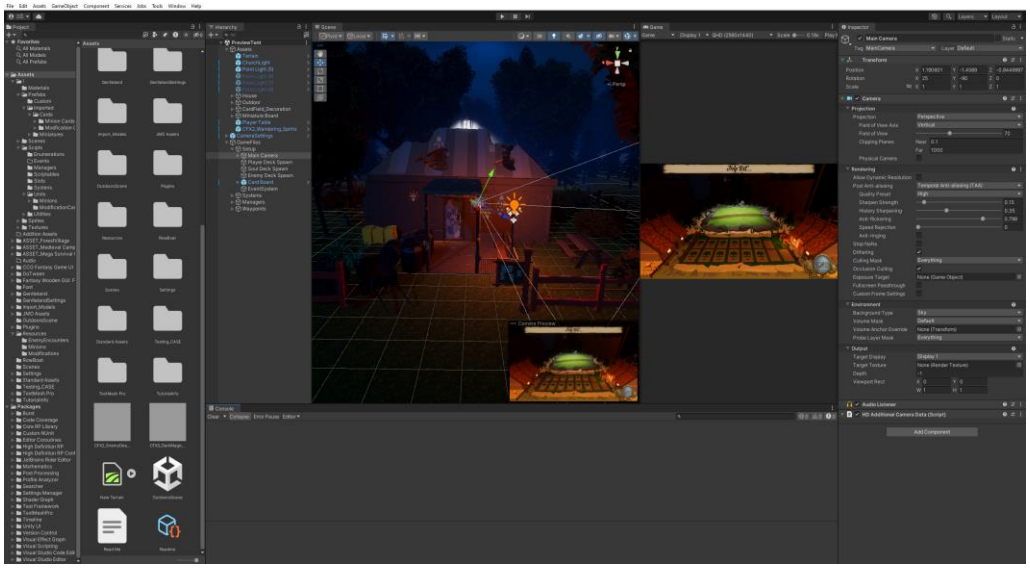
Strateške igre uloga jedan je od hibridnih žanrova, što je spoj taktičke igre na poteze i igre uloga. Pojam strateških igara postoji tisućama godina u fizičkom smislu kao igra na ploči, a prva digitalna verzija izašla je 1972. godine. To je vrsta igre u kojoj odluke igrača znatno utječu na ishod. Obično zahtjeva aktivno razmišljanje o odlukama i visoko svjesnost o situaciji. Primarno su takve igre orijentirane na strategiju, taktike, logistiku i upravljanje resursima. [3]

Ovaj projekt predstavlja spoj moje strasti prema strateškim igrama, *D&D-u*, 3D modeliranju i vještine programiranja. Kroz čitav razvoj ovog projekta, istraživao sam različite aspekte razvoja igre, naučio nova znanja i tehnike koje će mi pomoći u daljnjem razvoju, kako unutar tako i izvan *Unity* alata te sam unaprijedio svoje vještine u modeliranju. Cilj projekta bio je pokušati stvoriti svijet u kojem će igrači moći testirati svoje strateške sposobnosti i uživati u taktičkim izazovima koje pruža igra, izgraditi takvu igru sa svim pravima na materijale te iskoristiti znanje i potencijal izrađenog prototipa unutar ovog projekta za daljnji razvoj.

2. Metode i tehnike rada

Primarni dio razvoja igre odvijao se unutar programskog alata *Unity*, koristeći njegove mogućnosti za stvaranje interakcije između objekata i programiranja njihove logike. Dodatno, koristio sam alat Blender za izradu nekih 3D modela korištenih unutar igre. Blender je omogućio besplatan razvoj 3D modela preko alata otvorenog izvora koji je u potpunom vlasništvu kreatora te je bio korišten u svrhu izrade *Low-poly* modela, njihovog UV procesa projiciranja 2D slike na 3D površinu te ručne izrade i dodavanja već postojećih tekstura na modele. Kao glavni alat za programiranje same logike korišten je *Microsoft Visual Studio* koji je omogućio postizanje koherentnosti između vizualnost aspekta igre stvorenog unutar *Unity* alata i programskih mehanika implementiranih uz pomoć integriranog razvojnog okruženja.

2.1. Unity



Slika 1. Unity sučelje

Unity je 2D i 3D razvojni alat za igre koji potiče iz 2005. godine i razvija ga *tvrtka Unity Technologies*. Alat je stvoren kao odgovor na potrebu za korisnički pristupačnim i više platformskim alatom za izradu igara koji omogućava iskusnim i novim korisnicima razvojne mogućnosti stvaranja interaktivno i vizualno kvalitetnog sadržaja unutar igara. U vrijeme početka razvoja pružao je korisnicima jednostavnost, fleksibilnost i mogućnost istovremenog razvoja igre za više odabranih platforma. Osnivači *Unity* Technologies-a imali su želju pružiti korisnicima alat koji ne zahtijeva visoku razinu znanja za osnovan rad, ali i mogućnost naprednog korištenja za zahtjevnije potrebe. [4]

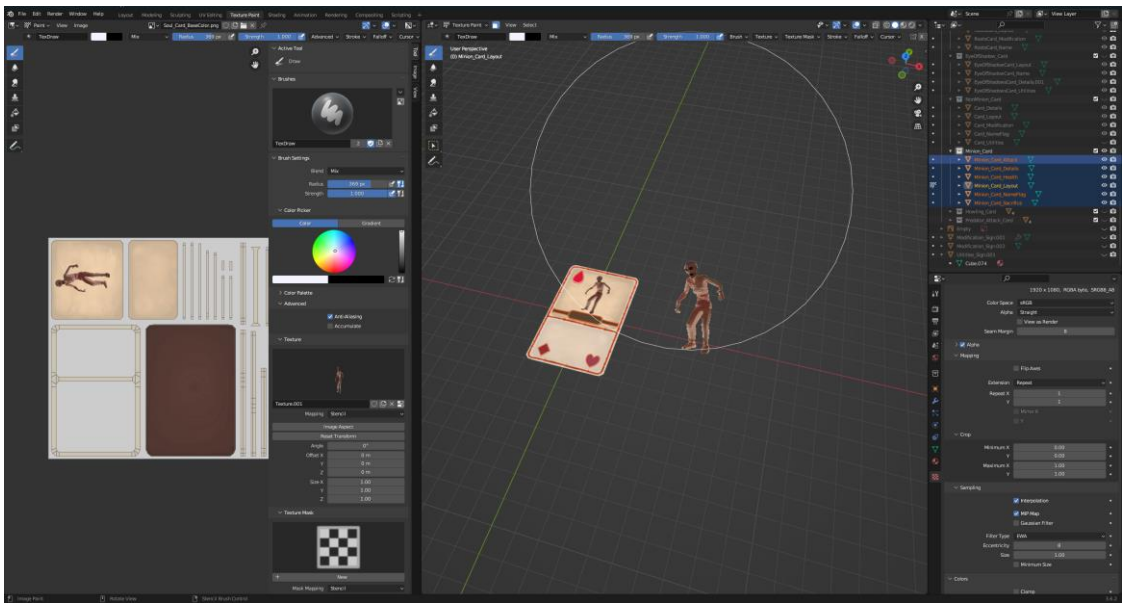
Danas je on korišten i u šire primjene te obuhvaća područje virtualne i proširene stvarnosti, implementaciju modernih tehnologija kao što je *RayTracing* i slično te konkurira vodećim alatima na tržištu poput *Unreal Engine* alata koji se koristi za razvoj *Triple-A* igara visoke kvalitete.

Unity igre podržavaju izgradnju na svim popularnim platformama kao što su *Android, iOS, Windows, MacOS, Linux, PS4, Xbox, HTML 5*. Postoji li potreba za materijalima poput raznih zvukova, korisničkog sučelja, slika, efekata, 3D modela i slično, *Unity* također održava stranicu za prodaju i kupnju raznih materijala za pomoć pri razvoju igre među kojima se mogu pronaći i neki besplatna sredstva za razvoj. Unutar ovog projekta tako su iskorišteni neki plaćena i besplatna sredstva za ubrzanje razvoja igre na određenu razinu kvalitete.

2.2 Blender

Blender je besplatan 3D grafički alat otvorenog koda koji služi za modeliranje, renderiranje, postavljanje tekstura i animiranje. On pruža široki raspon funkcionalnosti koje omogućuju korisniku stvaranje visoko kvalitetnih 3D modela te obradu istih. Alat ima široku primjenu u industriji kao što je izrada videoigara, animiranih filmova, kinematografija, arhitektura, medicina, inženjering i mnogim drugim sektorima

.



Slika 2. Blender sučelje

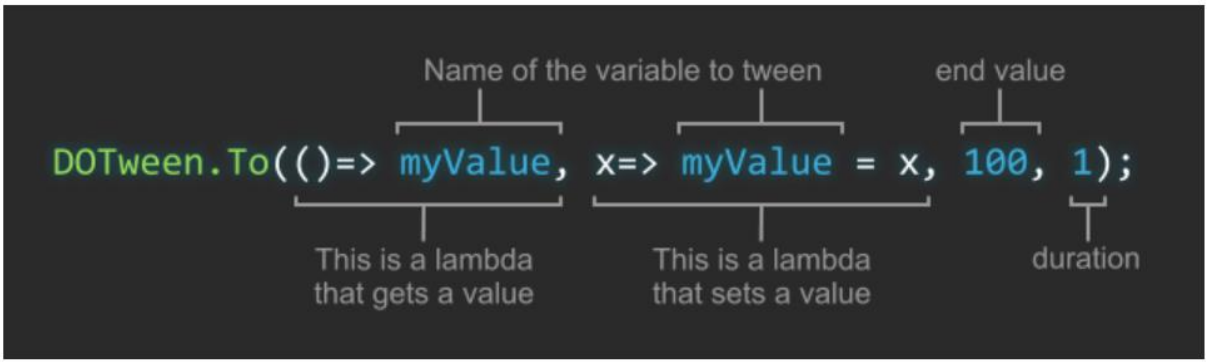
Razvijen je od strane *Blender Foundation* neprofitne organizacije te je prvi puta objavljen 1995. godine i od tada prošao je kroz mnogo verzija a danas je aktualan Blender 3.6 . Ključna karakteristika Blender-a je njegova besplatnost za korištenje te otvorenost koda koja pruža mogućnost vlastite nadogradnje alata svojim ili već kreiranim *plugin* modifikacijama što dodatno olakšava postizanje željenog cilja i olakšano korištenje [5]**.**

Glavne značajke Blendera:

- Modeliranje – Postoje različite alternacije za izradu 3D modela kao što je skulptiranje i poligonalno modeliranje
- Animacije – Nudi mogućnost izrade kompleksnih animacija putem ključnih okvira i deformacija kostiju
- Renderiranje – Podržava različite sinteze slike od kojih su najkorišteniji *Cycles* i *Evee* koji oboje omogućuju stvaranje foto realističnih PBR (eng. *Physically based render*) materijala
- Vizualni Efekti – Sadrži niz alata za stvaranje vizualnih efekata putem različitih simulacija poput simulacije vode, vatre, snijega, grmljavine i slično.
- Interaktivnost – Moguće je stvarati i interaktivnu 3D okolinu koristeći *Blender Game Engine*

2.3. DoTween

DoTween je popularna biblioteka za animacije unutar *Unity* razvojnog okruženja. Koristi se za jednostavno i efikasno stvaranje i upravljanje animacijama te ova biblioteka omogućuje glatke animacije transformacije, rotacije, skaliranja i mnogih sličnih parametara podržanih objekata [6].



Slika 3. DoTween funkcija

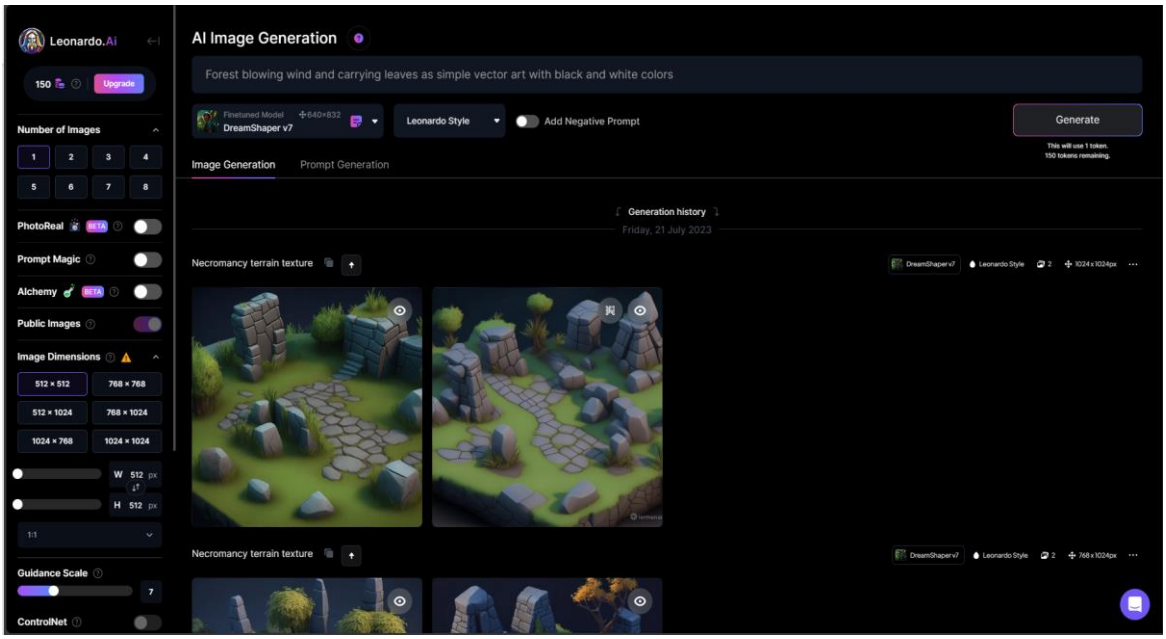
Glavne značajke:

- Podrška za *Unity* komponente – jednostavno se integrira sa *Unity* komponentama što omogućuje animiranje različitih objekata poput glavnog igrača, kamere, efekata i slično.
- Fleksibilnost – nudi precizno definiranje parametara koji sudjeluju u kreaciji animacije što olakšava prilagodbu animacije potrebama
- Grafika putem koda – pomoću biblioteke korisnik stvara složene animacije putem programskog koda što daje kontrolu nad svim aspektima animacije u bilo kojem trenutku
- Nizanje animacija – moguće je lako postavljati nizove animacija koje prate svoje sljedbenike i time postići kompleksne scenarije
- Interpolacija vrijednosti – pruža mogućnost korištenja linearnih ili interpolacijskih funkcija za postizanje željenih rezultata tijekom animacije, gdje interpolacijom je moguće postići prilagođene vrijednosti u trenutku što omogućuje prilagodbu brzine animacije u trenutcima unutar vremenskog perioda

2.4. Leonardo AI

Umjetna inteligencija doživljava ubrzanu evoluciju koja transformira brzinu i načine na koji rješavamo složene probleme u nastajanju. U posljednjem desetljeću, razvoj AI-a dosegnuo je nevjerojatne visine, a stvoren je također velik broj AI alata koji revolucioniraju različite sektore industrija.

Jedan takav alat je Leonardo AI, što je alat koji služi za razvoj slika u digitalnom obliku putem umjetne inteligencije. Vrlo je popularan novonastali alat za izradu slikovnog materijala potrebnih za izradu računalnih igara, no vrlo je moćan alat i može poslužiti za kreativno stvaralaštvo slika potrebnih za bilokakav korisnički projekt. Trenutno postoji besplatna verzija alata koja korisniku nudi 150 kredita za generiranje slika po danu i pruža različite tehnologije i mogućnosti kod generiranja te svakim dodatkom naplaćuje dodatni kredit za izradu. Za korisnike sa intenzivnijom potrebnom generiranja slika postoji i plaćeni plan za veću dozvoljeno količinu generiranog sadržaja po danu. [7]

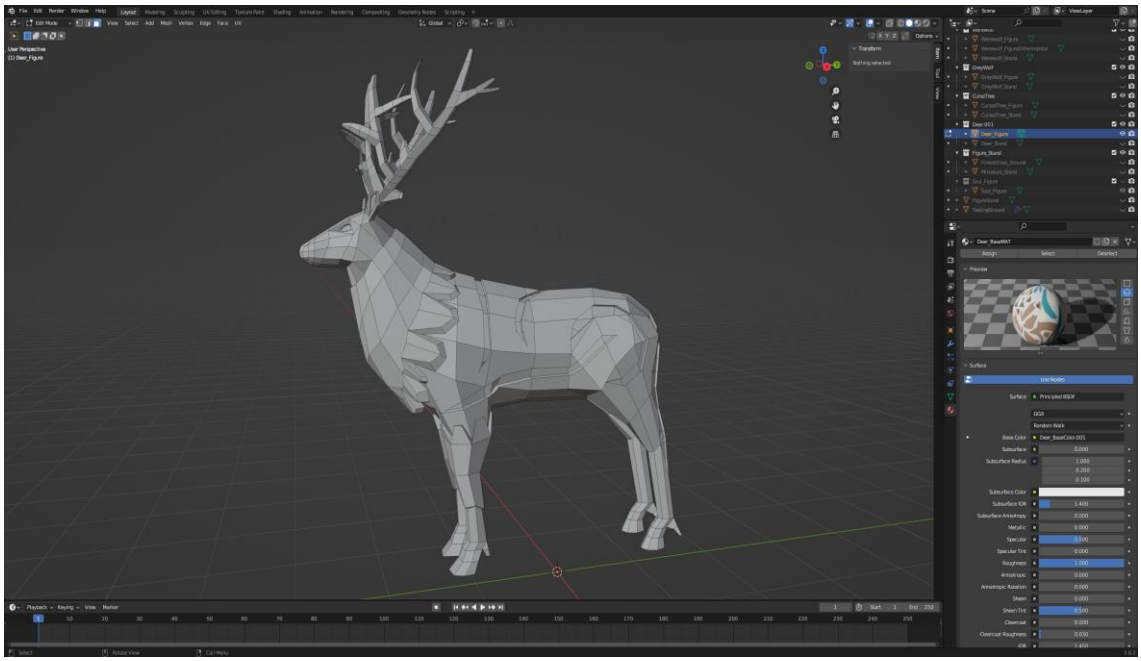


Slika 4. Leonardo AI sučelje

3. Razrada teme

U ovom poglavlju prolazi se kroz proces razvoja igre. Prvo je objašnjeno na koji način su kreirani neki od korištenih 3D modela za razvoj igre. Zatim se kratko prolazi kroz kreiranje projekta i pojašnjenje izvođenja igre. Na kraju se obrađuju sve glavne klase koje se nalaze u projektu i njihove interakcije s akcijama u igri.

3.1. Kreiranje 3D modela



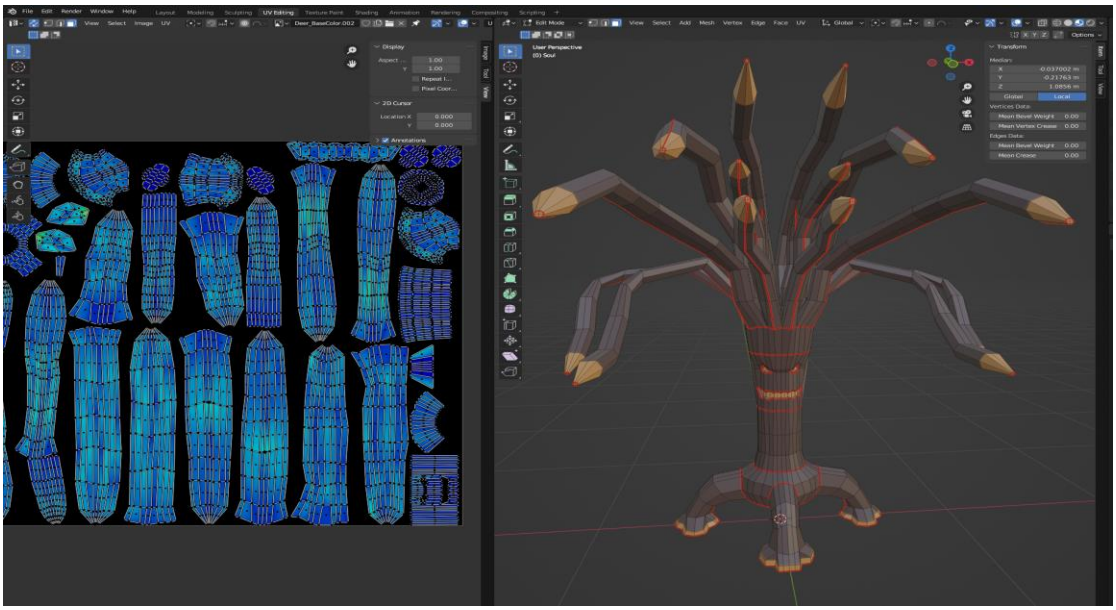
Slika 5. Primjer izrade 3D modela

Prvi korak bio je instalacija Blender alata koji je moguće preuzeti putem njihove službene stranice ili putem *Steam* platforme. Alat ima složeno sučelje koje je u potpunosti fleksibilno za uređivanje i potrebno ga je prilagoditi prema korisničkim potrebama. Iako postoje razne opcije i ikone putem kojih korisnik može doći do ciljanih rezultata, preporuča se rad sa kombinacijama tipka tipkovnice koji predstavljaju prečac za određene funkcije i tako ubrzava rad i olakšava veće alternacije 3D modela u kratkom vremenu. Proces samog modeliranja započinje odabirom načina obrađivanja 3D modela gdje je moguće krenuti sa osnovnim oblicima koji se skulpturiraraju putem danih i uređenih kistova koji svojim zadanim svojstvima utječu na deformaciju vrhova točaka na modelu. Takav način rada vrlo je koristan za modele sa velikim brojem vrhova ali nije toliko pozitivan utjecaj imao na izradu *Low-poly* modela korištenih u ovom projektu. Iz tog razloga odabran je ručni rad sa vrhovima, bridovima i licima

te takav način rada započinje dodavanjem osnovnog 3D modela kocke koji se tada uređuje putem pomicanja, skaliranja i rotacije postojećih i dodavanja novih vrhova.

Za rad su korištene razne slike referenca likova koji su dodani kao pozadinske slike iza modela i kada su korišteni zajedno u kombinaciji s *Wireframe* prikazom modela, omogućuju korisniku istovremenu vidljivost modela i slike, a time pomaže u postavljanju željene anatomije kreacija. Dodatno je korišten i *PureRef* alat za jednostavan prikaz kolekcije slike.

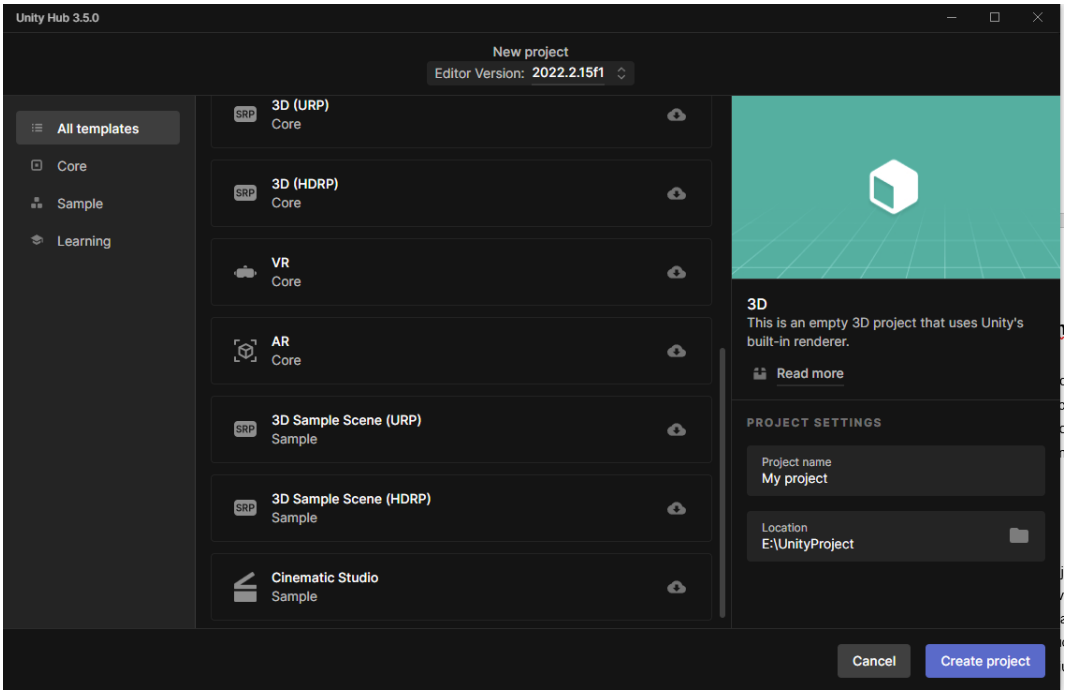
Nakon izrade kompletnih 3D modela započinje proces UV raspakiravanja kako bi se 2D teksture dodijelile na 3D modele. UV mapa je ravna reprezentacija površine 3D modela koja se koristi za jednostavno postavljanje tekstura procesom naziva UV raspakiravanje. U i V se odnose na horizontalnu i vertikalnu os 2D prostora [8]. Izvodi se na način da se prema označenim bridovi 3D površina raspakirava i transformira u 2D oblik koji predstavlja jedan otok u nizu te njegove točke na teksturi određuju način projiciranja teksture na model. Kako bi postiglo zadovoljavajuće raspakiravanje, na modelu se crvenim linijama opcijom *Mark Seam* označuju krajnji bridovi otoka te odabere opcija UV raspakiravanja. Zatim se svaki otok zasebno uređuje pomicanjem, rotiranjem i skaliranjem površine za željeni rezultat. Na kraju se pridodaje materijal koji sadrži površinska svojstva i omogućuje pravilno dodavanje teksture na model. Nakon uspješno provedenog procesa, model je spreman za renderiranje i izvoz u jedan od ponuđenih formata za daljnje korištenje unutar drugih alata i preglednika za 3D modele.



Slika 6. Primjer UV raspakiravanja 3D modela

3.2. Postavljanje projekta u Unity alatu

Proces razvoja projekta unutar *Unity* **alata** započinje pokretanjem *Unity Hub* aplikacije te odabirom verzije *Unity* alata koji će se koristi. Preporučeno je tijekom razvoja igre ostajati na određenoj verziji na kojoj je projekt započet jer nadogradnja ponekad potiče neželjeno ponašanje ako dolazi do velikih izmjena u određenim segmentima alata. Za brzu postavu projekta moguće je iskoristiti neke od danih predložaka koji automatski postavljaju osnovne dijelove projekta za određenu vrstu razvoja i pridodaje neke osnovne materijal za početak rada. Naposlijetku potrebno je odabrati lokaciju za spremanje projekta.



Slika 7. Sučelje za izradu projekta

Ključna odluka u kreiranju projekta također je u odabiru cjevovoda za renderiranje. On je odgovoran za izvođenje seta operacija uzimanja konteksta iz scene te njihov prikaz na ekranu. Ponuđena su 3 cjevovoda: *URP*, *HDRP* i *Built-in* a također postoji i SRP što je omogućuje vlastoručnu izradu cjevovoda. Za ovaj projekt odabran je HDRP, odnosno *High Definition Render Pipeline*, koji se koristi za izradu igara *Triple-A* kvalitete i koristi fizički baziranu svijetlost i materijale te ostvaruje naprednu grafiku visoke kvalitete za grafički zahtijevanije platforme. Ovaj cjevovod omogućuje korištenje najnovijih *Unity* tehnologija za vizualno poboljšanje ali također najviše utječe na performanse igre [9].

3.3. Izvođenje i interakcija s igrom

Kada se igra pokreće, igrač ulazi u glavni izbornik igre u kojem je prikazan tekst naslova igre i dva interaktivna gumba. Prvi gumb odvest će igrača na početak igre dok će drugi gumb ugasiti igru.

Slika 8. Početni izbornik igre

Igrač zatim pokreće igru i kamera prelazi u pogled igračeve ruke. Sada se odigrava prvi potez zvan pripremni potez, u kojem se u ruku dohvaća šest karti iz špila duša. Igrač odigrava željeni broj karata, iako se preporuča postaviti ih sve na prvom potezu jer će se karte ionako vratiti u špil prije početka sljedećeg poteza. Kada je izvučen maksimalan broj karata za taj potez, igraču se dozvoljava selekcija karte iz ruke. Ako se kursorom miša prolazi iznad neke karte, ona će se skalirati i istaknuti sve dokle god igrač ne odabere neku drugu kartu ili pomakne kursor dalje od ruke karata. Odabirom karte kamera prelazi u pogled na polja za karte, na koja u ovom slučaju igrač samo postavlja karte duša i priziva figure na odabrano polje igraće ploče. Kada svoj potez želi završiti, potrebno je kliknuti na gumb u donjem desnom kutu ekrana a zatim na red dolaze neprijateljske figure.

Slika 9. Prikaz igračevog pripravnog poteza

Nakon završetka poteza na red dolaze neprijatelji ali s obzirom da je to prvi njihov potez, oni ga također izvode kao pripremni potez. U tom potezu sve neprijateljske karte u borbi pozivaju svoje figure na nasumično odabrana polja desne polovice ploče i automatski završava njihov potez. Igrač je ponovno na redu ali ovaj puta i svaki puta nadalje vuče pet karata iz špil svih njegovih karti, i dvije karte iz špila duša. Razlog tome je što karte duša primarno služe kao figure za žrtvovanje zbog nepostojanja pripadajućih karta za napade i nema brojke zahtijeva za žrtvovanje kada se one prizivaju. Prvo se preostale neiskorištene karte iz prijašnjeg poteza vraćaju u pripadajući špil, zatim se špil promiješa i igrač dobiva novu mješovitu ruku karata. Više nema pripremni poteza i sada figure mogu krenuti u borbu.

Slika 10. Prikaz igračevog normalnog poteza

Ako igrač odabere jednu od karti figure koje zahtijevaju žrtvovanje, što se vidi po broju unutar oblika kapljice crvene boje u lijevo gornjem kutu karte, morat će odabrati određeni broj figura koje su već prizvane kao karte za žrtvovanje koje će se vratiti natrag u pripadajući špil. Nakon žrtvovanja, igrač više ne može poništiti svoju akciju i odabire mjesto gdje će prizvati novu kartu a zatim i pripadajuću figuru.

Slika 11. Žrtvovanje figura

Nadalje ako je odabrana karta za napad, igrač mora prvo postaviti kartu na polje a zatim i odabrati kompatibilnu figuru na koju može pripojiti kartu za napad. Kada je takva karta pripojena, igrač more iskoristiti figuru i njena svojstva kako bi napao neprijatelja u dometu sa jednim od pripojenih napada odabrane figure.

Slika 12. Povezivanje karte napada s kartom figure

Nakon što su postavljene karte figura i pripadajućih napada, igrač može baratati pozicijom kamere pomicanjem kotača miša i tako se pomaknuti na pogled igraće ploče. U ovom pogledu ima pregled svih figura na ploči a odabirom bilo kojeg polja, ono će se označiti bojom koja predstavlja mogućnost polja za neko akciju. Ako igra kursor pomakne do jedne od figura neprijateljskog tima i zatim pritisne lijevi klik miša, prikazat će se elementi korisničkog sučelja za informacije o odabranoj figuri. Ako je odabrana figura na igračevoj strani, pojavljuju se interaktivnih gumbovi za napad i pomicanje figure. Kada se odabere neko prazno polje, informacije o figuri nestaju do ponovnog odabira figure.

Slika 13. Odabir figure na ploči

U slučaju pritiska gumba za pomicanje, sve informacije nestaju a polja na koja se igrač može pomaknuti označuju se plavom bojom. Tada se bira nove polje na koje će se figura postaviti, ili je moguće desnim klikom miša zaustaviti svoju akciju. Figura se na taj način može pomicati po ploči, ali samo jednom za određenu distancu po potezu. Alternativno, ako se pritisne gumb za napad, igraču se prikazuje izbornik svih trenutno dostupnih napada koji su pripojeni na odabranu figuru. Navigaciju među kartama vrši tako da je ponuđena karta za odabir uvijek u sredini, a interaktivnim gumbovima može postaviti sljedeću ili prijašnju kartu na centralno mjesto.

Slika 14. Odabir karte za napad

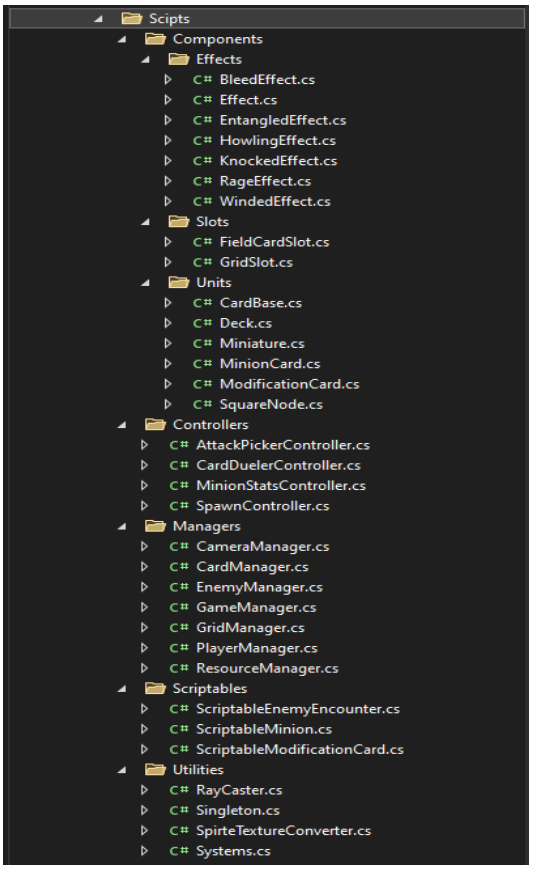
Odabirom željene karte putem interaktivnog gumba, igraču se sakriva izbornik i sva polja u dometu napada označavaju se žutom bojom. Ako se napad želi zaustaviti u ovom dijelu, korisnik može pritisnuti desni klik miša kako bi se vratio na početak prije odabira napada. Kada igrač odabere neprijateljsku figuru u zadanom dometu, prikazuju se elementi korisničkog sučelja za duel karata koje sudjeluju u trenutnoj borbi. Svaka karta napada ima zadane kocke koje se moraju izvrtjeti kako bi se kalkulirala šteta ili efekt napada. Također se prikazuju i informacije statusa figure koja se trenutno napada te se označavaju polja ispod figura.

Slika 15. Duel figura

Kada igrač završi svoj potez, neprijatelji će se pomaknuti prema najbližim figurama igračevog tima i pokušati napasti jednom od konfiguriranih karta napada nasumičnu figuru u dometu. Njihove karte napada se za razliku od igrača ne troše već se postavlja vrijeme prije ponovnog korištenja. Svaki početak potez također se primjenjuju svi pogođeni efekti neke figure. Tako će se do kraja igre izmjenjivati potezi sve do kad jedan od timova ne izgubi.

3.4. Arhitektura projekta

Još u početcima nastanka *Unity* alata nastalo je pitanje dobre strukture koda. Za vrijeme početaka razvoja alata često za aplikacije koristila *Model-View-Controller* struktura ali je problem nastao kada se takva struktura htjela iskoristiti i za izradu igara. Snaga *Unity* alata je u njegovoj strukturi komponenata, lakoći korištenja uređivača i ponovljivom iskorištenju relacija između predloška objekta i komponenata što se ne uklapa u MVC strukturu. Još od početnih vremena do danas, konkretna struktura nikad nije kompletno definirana a među zajednicom krenulo je korištenje *Manager* i *Controller* klasa u kombinaciji s *Helper* klasama. U mnogim projektima svrhe tih klasa su varijabilne i važno je napomenuti kako nema određenih pravila o organizaciji koda jer ona ovise o osobnim preferencijama, veličini projekta, odlukama razvojnog tima i sličnim aspektima. Ono što je ključno jest kako je potrebno za bolju organizaciju potrebno u početku definirati pravila koja će se konzistentno održavati tijekom čitavog razvoja.



Slika 16. Raspored klasa u projektu

Menadžeri su klase koji su stalne za vrijeme izvođenja igre, i odgovorni su za komunikaciju sa drugim skriptama. *Singletoni* pomažu toj klasi kako bi osigurali jednu instancu koja će biti dostupna svim ostalim klasama da dohvate njegove varijable i metode. U općenitom smislu, menadžeri su odgovorni za upravljanje logikom više entiteta [10]. Kontroleri se obično fokusiraju na upravljanje jednim entitetom i vrše komuniciraju sa pomoćnim klasa i menadžerima za opće zahtjeve. Ovakav način strukture definiran je početkom izrade ovog projekta kako bi se dostigla razina organizacije koda koja pomaže održavati čist i strukturiran kod za lakšu razumljivost koda, lakše praćenje i rješavanje grešaka tijekom razvoja, održivost te olakšano dodavanje novih značajka.

3.4.1 Singleton uzorak dizajna

Singleton je uzorak dizanja često korišten unutar *Unity* okoline, pružajući funkciju kontejnera za održavanje vrijednosti koje su globalno dostupne kroz cijelu scenu. Vrlo su korisni i kada je potreban prebačaj nekih ključnih podataka iz jedne scene u drugu bez potrebe za spremanjem i ponovnim učitavanjem podataka u pozadini. Iako korištenje u malim projektima neće uzrokovati nikakve negativne posljedice, kod većih projekata dobra je praksa održavati minimalnu količinu *singleton* klasa od kojih svaki ima svoju specifičnu odgovornost kako ne bi došlo do otežanog održavanja i nepotrebnog zadržavanja podataka u memoriji. Glavna vrlina uzorka jest njegovo jedinstveno postojanje kroz instancu tijekom igre koje je prema arhitekturi moguće iskoristiti za menadžerske klase [11].

```
public abstract class Singleton<T> : MonoBehaviour where T : MonoBehaviour{
    public static T Instance { get; set; }
    protected virtual void Awake() => Instance = this as T;
    protected virtual void OnApplicationQuit(){
        Instance = null;
        Destroy(gameObject);
    }
}
```

Ova generička klasa *Singleton<T>* omogućava stvaranje *singleton* instanci za nadolazeće komponente koje nasljeđuju *MonoBehaviour* klasu. Statičko polje instance omogućuje pristup jedinstvenoj instanci a metoda *Awake* postavlja referencu na trenutnu komponentu kod inicijalizacije skripte. Kada se izvršava gašenja igre, pozvat će se **m**etoda *OnApplicationQuit* koja će instancu postaviti na *null* vrijednost i uništiti pripadajući objekt, osiguravajući na taj način oslobađanje resursa nakon izlaza iz aplikacije.

3.4.2 MonoBehaviour klasa

MonoBehaviour je osnovna klasa za sve skripte koje su u interakciji sa objektima igre. Nasljeđivanje ove klase omogućuje pristup raznim ugrađenim funkcijama i eventima koji su od velike važnosti za definiranje konkretnog ponašanja nekog objekta. Svaki objekt u igri može sadržavati jednu ili više povezanih *MonoBehaviour* komponenti napisanih u C# programskom jeziku koje definiraju dio ponašanja i reakcije na različite događaje unutar igre. Veliki značaj u *MonoBehaviour* komponentama obuhvaća set metode koje se automatski pozivaju tijekom životnog ciklusa jedne komponente kod izvođenja igre i one omogućuju programeru potpunu kontrolu ponašanja objekta u igri kroz njegovo postojanje [12].

Osnovne metode za rad:

- *Awake()* – Metoda *Awake* poziva se prilikom aktiviranja ili kreiranja objekta i obično se koristi za postavljanje početnih vrijednosti komponente i za inicijalizaciju komponenti
- *Start()* – Metoda *Start* poziva se neposredno prije prve *Update* metode, odnosno prije prvog okvira u kojem će objekt biti vidljiv igraču na ekranu.
- *Update()* – Metoda *Update* poziva se svaki okvir tijekom izvođenja igre i često služi za implementaciju logike nekog objekta koji zahtjeva ponavljajuće izvođenje kao naprimjer za provjeru unosa tipke na tipkovnica ili mišu, fizičko ponašanje objekta ili provjere eksterne ili interne promjene nekog stanja. Iako metoda sama po sebi nije inherentno opasna za performanse i memoriju, nepravilno korištenje može potencijalno uzrokovati probleme ako se ne upravlja ispravno. Time može uzrokovati nepotrebne pozive na metode, ne oslobađanje resursa, česte alokacije i dealokacije bez nepravilnog oslobađanja koje utječu na fragmentaciju memorije, povećanu upotrebu memorije zbog neučinkovitih(skupih i memorijskih zahtjevnih) upita i izračuna te mnoge druge probleme. Stoga je preporučeno pažljivo baratati metodom i uvijek provjeravati performanse i memoriju tijekom izvođenja. Također ponašanje ove metode zavisi o performansama igre, odnosno o trenutnom broju okvira koji se pojavljuje u sekundi što može utjecati na metode i varijable koje zahtijevaju stalnu vremensku konzistentnost kao što je metoda za odbrojavanje vremenskog intervala.
- *FixedUpdate()* – Metoda *FixedUpdate* poziva se određeni broj puta u sekundi što obično iznosi 50 puta i koristi se za operacije koje ovise o kontinuiranom vremenu, kao što je to slučaj kod fizičkog ponašanja objekta.

- *LateUpdate()* – Metoda LateUpdate poziva se nakon svih *Update* metoda i korisna je u trenutcima kada se želi osigurati ažuriranje ostalih objekata prije trenutnog objekta, i često se koristi za izvođenje animacija nad kostima na čija transformacijska svojstva u isto vrijeme utječe i neka skripta.
- *OnEnable()* – Metoda *OnEnable* poziva se pri aktiviranju objekta i koristi se za pokretanje posebnih metoda i inicijalizaciju vrijednosti pri svakom aktiviranju trenutnog objekta
- *OnDisable()* – Metoda *OnEnable* poziva se pri deaktiviranju objekta i koristi se za pokretanje posebnih metoda i inicijalizaciju vrijednosti pri svakom deaktiviranju trenutnog objekta, ili za čišćenje resursa i spremanje stanja kod deaktivacije
- *OnDestroy()* – Metoda *OnDestroy* poziva se neposredno prije uništenja objekta i korisna je za čišćenje resursa i spremanje stanja te izvođenje konačnih operacija prije samog uništenja objekta

Kao što je prije navedeno, svaka *Unity* skripta mora biti izvedena iz *MonoBehaviour* klase ako se želi objektima igre dodati kao komponenta, a ako se ona stvara unutar *Unity* uređivača već će biti izvedena iz *MonoBehaviour* klase i ne treba o tome brinuti. Kako se objekti tijekom izvođenja dinamički stvaraju, omogućuju, onemogućuju i uništavaju, za razliku od razvoja aplikacija ovdje se ne preporuča korištenje konstruktora jer to otežava i ograničava upravljanje životnim ciklusom objekata i komponenta u igri. Isto tako često je potrebno pristupiti drugim komponentama na istom ili drugim objektima a takvo referenciranje nije dostupno u konstruktorima jer oni nisu svjesni konteksta čitave igre. Umjesto konstruktora, preporuča se korištenje specifičnih metoda za precizno upravljanje redoslijedom inicijalizacije i rada kao što su *Awake*, *Start*, *OnEnable* i slično.

3.4.3 ScriptableObject klasa

ScriptableObject posebna je vrsta klase koja se nasljeđuje i omogućava programeru stvaranje prilagođenih podatka koji se mogu spremiti kao resursne datoteke i dijeliti između različitih objekata u *Unity* projektu. Za razliku od *MonoBehaviour* klase ne može se dodati kao komponenta nekom objektu ali se zato učestalo koristi za stvaranje i upravljanje podacima koji nisu povezani s određenim objektima ili komponentama. Korištenjem *ScriptableObject* klasa može se osigurati bolja organizacija podataka tijekom razvoja projekta. Svi kreirani podaci mogu se dohvatiti iz bilo koje klase, na bilo kojem objektu u bilo kojoj sceni [13].

Glavne značajke koje pridonosi su:

- Povezanost sa različitim skriptama – može se koristiti kao način za pohranu i dijeljenje podataka između različitih skripti i komponenti, bez potrebe za dupliciranjem podataka
- Razdvajanje konfiguracije od koda – omogućava izradu postavki koji se mogu učitati u igru i prilikom promjene postavki nije potrebna nikakva modifikacija koda
- Serijalizacija i uređivanje – kao resursi lako se daju urediti unutar *Unity* uređivača putem prilagođenih editor skripti i pomoću atributa što u konačnici omogućava stvaranje vlastitog korisničkog sučelja za upravljanje podacima.
- Ponovna upotrebljivost i pouzdanost – mogu se ponovno upotrijebiti za druge projekte i čuvaju se kao materijali unutar projekta, što znači da se podaci čuvaju između pokretanja igre i ne ovise o životnom ciklusu objekata ili cijele scene
- Prilagodljivost – lako ih je uređivati tijekom izvođenja te za razliku od *MonoBehaviour* klasa, izmijenjeni podaci ostaju promijenjeni i nakon gašenja igre

3.4.4 Karte

Igra se bazira na korištenju karata kako bi se upravljalo jedinicama u borbi, i tako postoji dvije vrste: karte minijatura i karte modifikacijskih napada. Kako obje vrsta karata imaju neka zajednička svojstva, kreirana je apstraktna klasa *CardBase* koju će konkretna vrsta karte nasljeđivati a time će povećati upotrebljivost koda ako se odluči kasnije dodavanje novih vrsta karata te je moguće ponovno naslijediti osnovna svojstva i pridodati vlastita.

```
  public abstract class CardBase : MonoBehaviour
  {
    public bool isPlaced = false;
    public bool isScaled = false;
    [SerializeField] private int _cardId = -1;
    [SerializeField] private string _cardName;
    [SerializeField] private CardType cardType;
    [SerializeField] private Vector3 _lastNormalScale;

    public virtual void SetCardText(){}
    public void SetBaseProperties(int id, string name, CardType type)
    {
        _cardId = id;
        _cardName = name;
        cardType = type;
    }
    public int GetCardId(){ return _cardId; }
    public string GetCardName(){ return _cardName;}
    public CardType GetCardType(){ return cardType;}
    public void ScaleCard()
    {
        const float _scaleOffset = 0.05f;
        if (GameManager.Instance.gameState == GameState.PlayerTurn &&
!scaled)
        {
            _lastNormalScale = transform.localScale;
            transform.localScale = new Vector3(1.2f, 1.2f, 1.2f);
            transform.position = new Vector3(transform.position.x,
transform.position.y + _scaleOffset, transform.position.z);
            scaled = true;
        }
    }
    public void NormalizeScale()
```

```
    {
        const float _scaleOffset = 0.05f;
        transform.localScale = _lastNormalScale;
        transform.position = new Vector3(transform.position.x,
transform.position.y - _scaleOffset, transform.position.z);
        scaled = false;
    }
}
```

Kada se postavlja tekst karte poziva se **m**etoda *SetCardText* koja koristi ključnu riječ *virtual* za označavanje metode u apstraktnoj klasi što znači da ta metoda mora biti nadjačana u izvedenim klasama, omogućavajući prilagodbu ponašanja metode u klasi koja ju nasljeđuje. Ona se bavi postavljanjem 3D teksta na kartu koji će označavati neke atribute koje karta ima. Iako sve karte sadržavaju tekst, svaka vrsta će imati različit broj objekata teksta na različitim mjestima, te će apstraktna klasa i virtualna metoda osigura da svaka naslijeđena klasa mora implementirati ovu metodu ali ima potpunu slobodu načina implementacije. Metoda *ScaleCard* dohvaća transformacijsku komponentu objekta i postavlja veličinu objekta na skaliranu veličinu veću za 0.2 od trenutne veličine objekta i zapisuje u Vector3 varijablu svoju normalnu veličinu kako bi se mogla vratiti u prijašnje stanje. Zatim se pozicija objekta na y osi povećava za određeni pomak kako bi u konačnici igrač na ekranu vidio povećani i približeni prikaz potencijalne karte za odabir. Kod potrebe za povratkom skalirane karte na početnu veličinu koristi se metoda *NormalizeScale* koja dohvaća transformacijsku komponentu objekta i postavlja veličinu objekta na zadnju zapisanu Vector3 varijablu koja sadrži koordinate x, y i z u koordinatnom sustavu. Zatim se postavlja pozicija objekta na poziciju prije pomaka objekta na y osi putem skaliranja. Provedena metoda sudjeluje u vizualnom prikazu smanjivanja karte na normalnu veličinu nakon kruženja miša nad sljedećom potencijalnom kartom za odigravanje. Ostale metode koriste se za dohvaćanje vrijednosti kako bi se spriječila nedozvoljena promjena stanja varijabli ili dohvaćanje nepotrebnih varijabla izvan klase.

Atribut *[SerializeField]* koristi se za označavanje privatnih varijabli kako bi se omogućio njihov prikaz unutar *Unity* uređivača. Ovim načinom postiže se uređivanje vrijednosti varijabli izvan koda čime je olakšano postavljanje i uređenje tih varijabli bez potrebe da varijable budu javne [14].

3.4.4.1 Karte figura

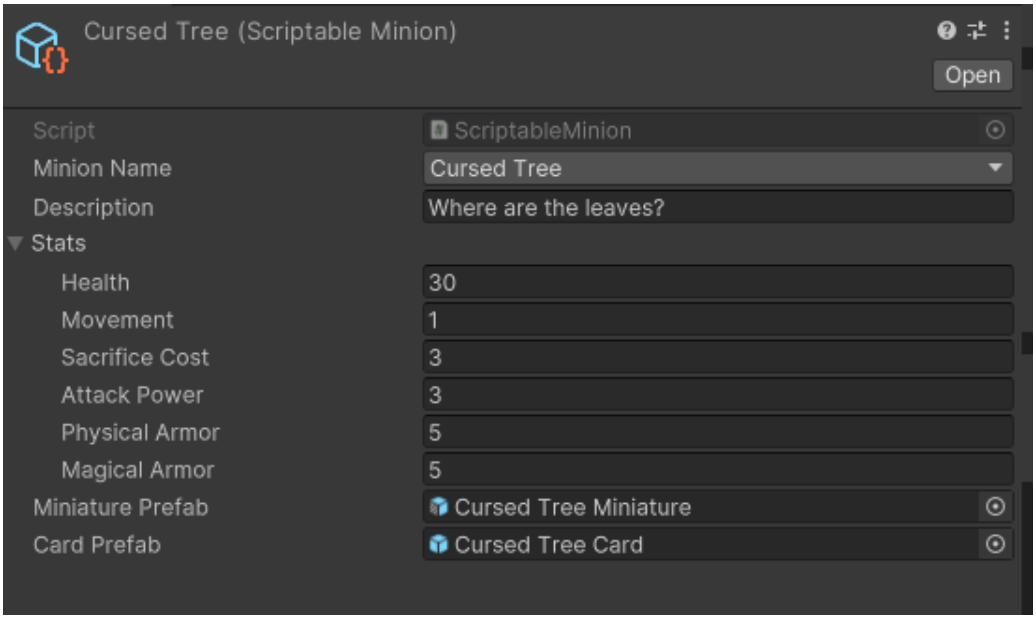
Prva vrsta karte u igri su karte figura koje igrač može odigrati na slobodno polje za karte i stvoriti figuru na odabranom slobodnom mjestu igraćoj ploči. Svaka figura ima zajednička svojstva različitih početnih i trenutnih vrijednosti koja reprezentiraju jačinu karte i broj karti za žrtvovanje kako bi se figura prizvala u igru.

```
[CreateAssetMenu(fileName = "New minion card", menuName =
"Scriptables/Minion")]
public class ScriptableMinion : ScriptableObject
{
    public MinionName minionName;
    public string description;
    public Stats stats;
    public GameObject miniaturePrefab;
    public GameObject cardPrefab;
}

[Serializable]
public struct Stats
{
    public int Health;
    public int Movement;
    public int SacrificeCost;
    public int AttackPower;
    public int PhysicalArmor;
    public int MagicalArmor;
}
```

Najprije je kreirana klasa *ScriptableMinion* koja nasljeđuje prije spomenutu *ScriptableObject* klasu i služi kao kontejner za podatke čije će kreirane resurse svaka odgovarajuća klasa karte dohvatiti i na temelju njih inicijalizirati dio svojeg početnog stanje. Atribut koji se pojavljuje iznad naziva klase označavanja kreiranje posebnog izbornika za stvaranje novog skriptabilnog objekta a u nastavku postavlja zadano ime i putanju opcije u izborniku. Struktura sadrži atribut *[Serializeable]* koja klasu označuje kao serijalizirajuću i omogućuje njeno uređivanje i prikaz u *Unity Inspector* prozoru. Zatim slijedi ime, opis karte koji će se prikazati pomoću 3D objekta teksta na karti, referenca na objekt figure koja se stvara odigravanjem karte, 3D model karte i borbene osobine.

Struktura *Stats* sadrži organizirani niz osobina karte:

- *Health* – trenutni životni bodovi figure
- *Movement* – domet koraka figure za koji se jednom po potezu pomiče
- *SacrificeCost* - broj potrebnih figura za žrtvovanje
- *AttackPower* – broj koji se pridodaje iznosu napada kao dodatna šteta
- *PhysicalArmor* **i** *MagicalArmor* – broj koji se odbija od ukupnog izračuna štete koji figura prima u trenutku



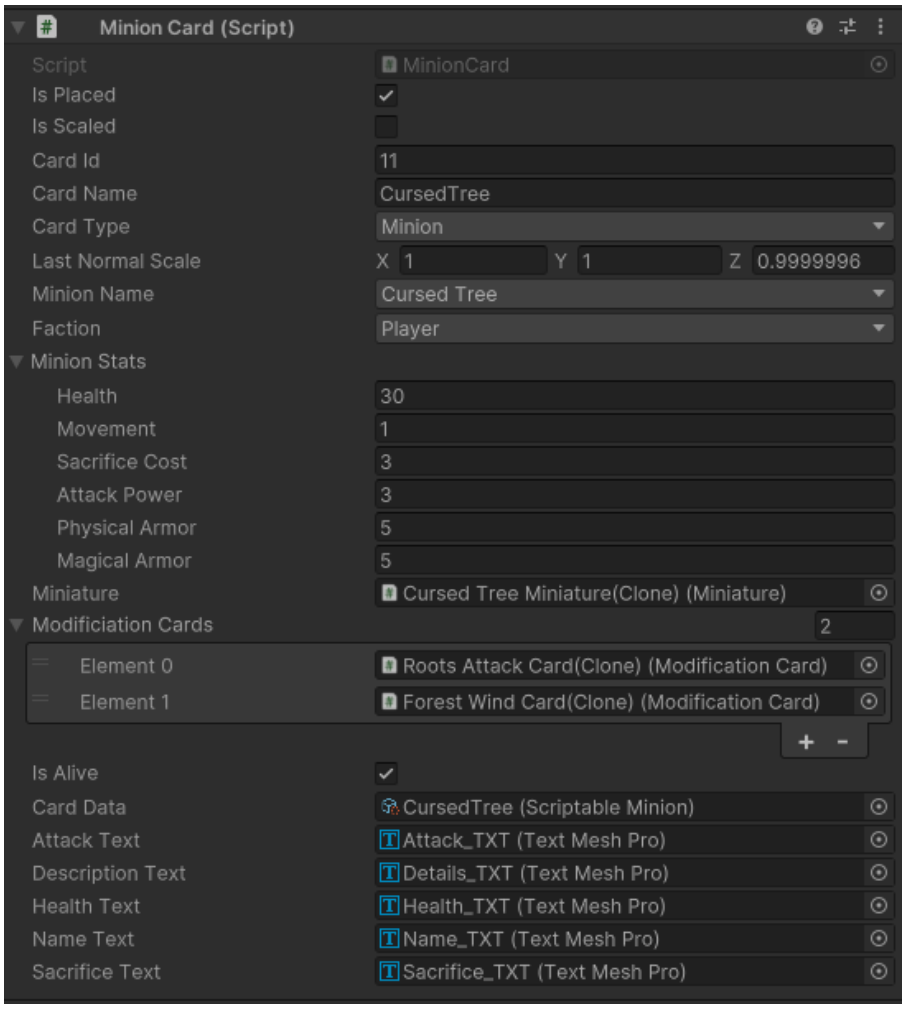
Slika 17. ScriptableMinion skriptabilni objekt

Slijedi implementacija pripadajuće klase *MinionCard* koja nasljeđuje *CardBase* klasu i sva njezina svojstva. Kako bi prepoznali kojoj minijaturi karta pripada, korištena je enumeracija *MinionName* sa svim imenima postojanih figura u igri i isto tako napravljena je enumeracija *Faction* koja raspoznaje igračev i neprijateljski tim figura.

```
public class MinionCard : CardBase{
      public MinionName minionName;
      public Faction faction;
      public MinionStats minionStats;
      public Miniature miniature;
      public List<ModificationCard> modificiationCards;
      [SerializeField] private bool _isAlive = true;
      [SerializeField] private ScriptableMinion _cardData;
            …
      override
      public void SetCardText(){}
      public void SetupProperties(int cardId, ScriptableMinion cardData,
      Faction cardFaction){}
      public void ManageHealth(int amount){}
      public void ResetAttackDamage(){}
      public bool IsMinionAlive(){}
      public int GetMinionMaximumHealth() {}
      public GameObject GetMiniaturePrefab(){}
}
```

Kako *ScriptableMinion* sadrži predefinirana inicijalna svojstva karte, koristi se metoda *SetupProperties* koja se poziva kod kreiranja karte i postavlja referencu unutar klase na skriptabilni objekt i postiže povezanost i dostupnost pripadajućih podataka unutar cijelog životnog ciklusa karte. Metoda *SetCardText* i *SetBaseProperties* poziva se kako bi se karta u potpunosti popunila potrebnim podacima i prikazala neka svojstva kao 3D tekst na karti. S obzirom da karta svoje glavna *Stats* svojstva mijenja tokom igre dok skriptabilni objekt dijeli svoje podatke između više istih karata i pamti promijene nakon završetka igre, potrebni podaci kopirani su vlastitu *Stats* strukturu. Ona se može proizvoljno mijenjati te ako postoji potreba za vračanjem inicijalnih podataka može se realizirati dohvatom reference skriptabilnog objekta.

Kada igračeva figura prima iznos štete, poziva se metoda *ManageHealth* koja oduzima vrijednost štete od trenutnih životnih bodova karte i provjerava jesu li bodovi pali na ili ispod nule u kojem slučaju se postavlja varijabla *isAlive* na lažnu vrijednost, označavajući smrt igračeve figure i pripadajuće karte. U nekom trenutku pozvat će se metoda *IsMinionAlive* i odstraniti će kartu i figuru iz igre. Ostale metode služe za dohvat i vraćanje inicijalnih vrijednosti privatnih varijabli koje kontrolu pristupa vrše putem metoda umjesto direktnog pristupa.

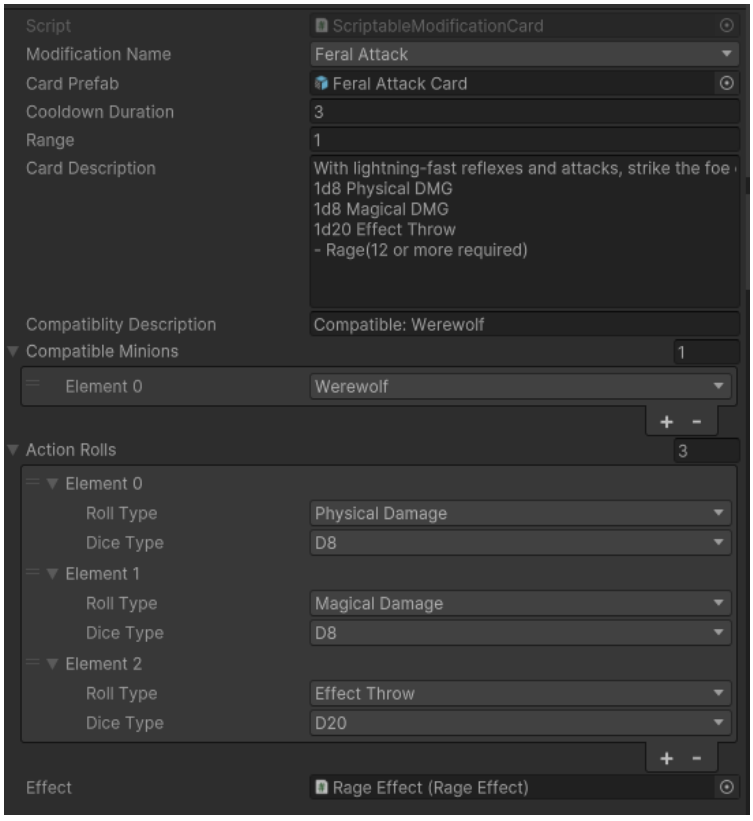


Slika 18. MinionCard komponenta

3.4.4.2 Karte za napad

Druga vrsta karte u igri su karte za napad, povezive na dostupne kompatibilne karte figura. Svaka karta inicijalizira svoja svojstva putem *ScriptableModificationCard* objekta i unutar igre postoji šest različitih definiranih objekata, odnosno postoji šest različitih *Modification* karata za odigravanje.

```
[CreateAssetMenu(fileName = "New modification card", menuName =
"Scriptables/ModificationCard")]
public class ScriptableModificationCard : ScriptableObject{
    public ModificationName modificationName;
    public GameObject cardPrefab;
    public int cooldownDuration;
    public int range;
    public string cardDescription;
    public string compatibilityDescription;
    public List<MinionName> compatibleMinions;
    public List<RollStats> actionRolls;
}
```



Slika 19. Primjer ScriptableModificationCard objekta

Klasa *ModificationCard* nasljeđuje apstraktnu klasu *CardBase* i sadrži metode za postavljanje i dohvaćanje svojstva karte figure te postavljanje vrijednosti 3D teksta i bazičnih podataka svake karte.

```
public class ModificationCard : CardBase{
    [SerializeField] private int _cooldownleft = 0;
    [SerializeField] private ScriptableModificationCard _cardData;
    [SerializeField] private Effect _cardEffect;
    [SerializeField] private TextMeshPro _nameText;
    [SerializeField] private TextMeshPro _descriptionText;


    override
    public void SetCardText(){}
    public void SetupProperties(int cardId,
ScriptableModificationCard cardData){ }
    public void SetCooldown() { }
    public void ReduceCooldown(){ }
    public void ApplyAftermathEffect(CardDuelerController
cardDueler){}
    public bool IsOnCooldown(){ }
    public ModificationName GetModificationName(){}
    public List<RollStats> GetActionRolls() { }
    public string GetAttackDescription(){ }
    public int GetRange() { }
}
```

Klasa sadrži reference na 3D tekst koji su postavljeni kao djeca roditelja objekta na koji je ova klasa dodana kao komponenta, sadrži referencu na objekt efekta koja se aktivira nakon napada ovisno o dovoljnom broju rezultata kocke, te referencu na skriptabilni objekt koji inicijalizira potrebna svojstva karte. Postoji i posebna metoda *ApplyAftermathEffect* koja se poziva unutar kasnije obrađenog kontrolera i ako je dobiven dovoljan broj na kocki stvara objekt sa komponentom specifičnog efekta. Objekt se postavlja kao dijete na odgovarajuću figuru i zapisuje u njegovu listu trenutno pripojenih efekata koji se izvršavaju na početku svakog poteza. Ako je karta korištena od strane neprijateljske figure, ona se neće obrisati iz igre kao što je slučaj kod igračevog korištenja, već će putem metode *SetCooldown* postaviti vrijeme prije ponovnog korištenja izraženog u broju poteza. *ReduceCooldown* metoda će svaki potez taj broj smanjiti do kad ne dođe do nule, a provjera stanja izvan klase radi pozivom *IsOnCooldown* metode.

3.4.5 Efekti

Svaka karta za napada nakon duela može ostaviti određeni efekt na nekoj figuri ovisno o potrebnom broju dobivenom na kocki. Efekt je definiran ukupnom vrijednosti trajanja u potezima, trenutnim brojem preostalih poteza, štetom koju nanosi figuri, zahtijevanom broju za postavljanje efekta, imenom i vrijednosti za određivanje hoće li se efekt postaviti na napadačku ili braniteljsku figuru u duelu.

```
public abstract class Effect : MonoBehaviour
{
    public int turnsDuration;
    public int turnsLeft;
    public int damage;
    public int rollRequired;
    public bool targetSelf;
    public string effectName;
    public virtual void Apply(MinionCard minion){ }
    public bool IsStillActive() {
        if(turnsLeft > 0) {
            return true;
        }
        else{
            return false;
        }
    }
    public void ShowPopupText(Miniature miniature){
        string infoText;
        if(damage  > 0)
        {
            infoText = "-" + damage + " damage" + "(" + effectName +
")\n" + "(" + turnsLeft + " turns left" + ")";
        }
        else{
            infoText = "(" + effectName + ")\n" + "(" + turnsLeft + "
turns left" + ")";
        }
        GameManager.Instance.ToggleMinionPopup(infoText,miniature);
    }
}
```

Klasa *Effect* je apstraktna klasa koju nasljeđuje šest različitih klasa zasebnog efekta i nadjačava njenu metodu *Apply*. Svaki put kada igrač ili neprijatelji dođe na potez, poziva se metoda *IsStillActive* koja provjerava hoće li efekt ostati na figuri do sljedeće provjere i ako je iznos preostalih poteza veći od jedan vratiti istinu vrijednost ili u drugom slučaju vratiti laž. Ovisno o odgovoru metode efekt će ostati i aktivirati se ili će biti otklonjen sa figure.

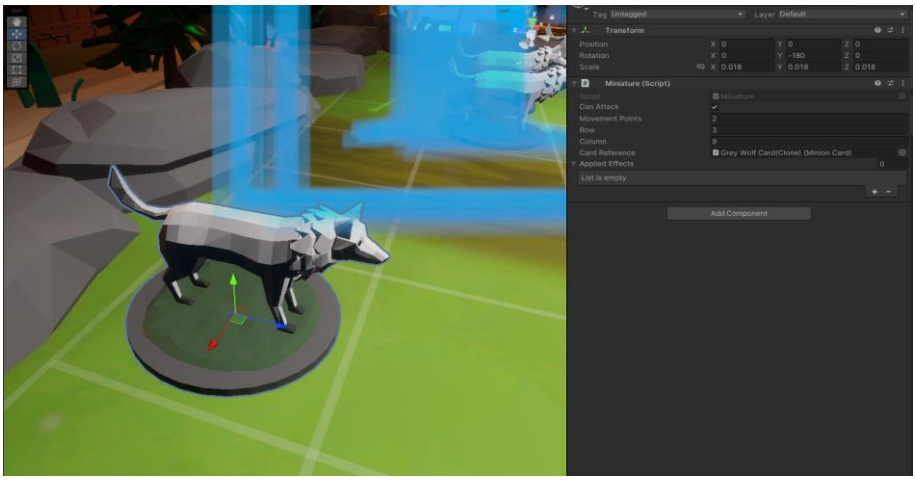
```
override
public void Apply(MinionCard minion)
{
    if (turnsLeft == 0) { return; }

    minion.ManageHealth(-damage);

    turnsLeft--;
    ShowPopupText(minion.miniature);
}
```

Ovdje je za primjer prikazana nadjačana metoda klase BleedEffect koja poziva metodu *ManageHealth* proslijeđene karte figure i oduzima joj životne bodove za vrijednost štete, a zatim smanjuje broj preostalih poteza za jedan i poziva metodu *ShowPopupText* kako bi se postavio i prikazao tekst na ekranu. Tekst prikazuje ime efekta, vrijednost štete koju nanosi i preostali broj poteza za koji će efekt biti aktivan na figuri.

3.4.6 Figure

Kada se aktivira karta figure, igrač na polje treba prizvati pripadajuću figuru na jedno od slobodnih polja ploče. Prizvanom figurom može se upravljati i izvršavati akcije a kada se stvore neprijateljske karte na početku igre, također se ubrzo nakon prizovu pripadajuće figure tih karata na igraćoj plohi čijom će logikom upravljati *EnemyManager* instanca klase. Svaka figura jednom se po potezu može pomicati na igraćoj ploči za određeni broj mjesta i jednom može napasti neprijateljsku figuru iskoristivši pritom određenu kartu za napad. Figura svoju akciju tada ne može izvoditi do početka sljedećeg poteza i ovo pravilo vrijedni podjednako za igračeve i neprijateljske figure. *Miniature* klas definirana je vrijednosti tvrdnje o mogućnosti napasti ovaj potez, distancom pomaka na igraćoj plohi, redom i stupcem u kojima se nalazi, referencom pripadajuće karte i trenutno prikvačenih efekata na figuri

```
public class Miniature : MonoBehaviour{
    public bool canAttack;
    public int movementPoints;
    public int row;
    public int column;
    public MinionCard cardReference;
    public List<Effect> appliedEffects = new List<Effect>();
    public void ApplyEffects(){}
    public void SetMiniatureGridLocation(GridSlot slot){}
    public void RestoreAction(){}
}
```



Slika 20. Prikaz figure

3.4.7 Špil karata

Postoji tri vrste špila unutar igre: igračev špil, špil igračevih duša i neprijateljski špil. Svi špilovi inicijaliziraju se na početku igre i služe kao roditeljski objekti pripojenih karata. Igrač vuče karte iz špila duša i svojeg špila ostalih karata a broj karata iz špila ovisi o vrsti poteza. . Na početku svakog igračevog poteza osim prvog, preostale neiskorištene karte iz prijašnjeg poteza vraćaju se u pripadajući špil i mogu se ponovno pojaviti u igračevoj ruku na nekom od njegovih sljedećih poteza.

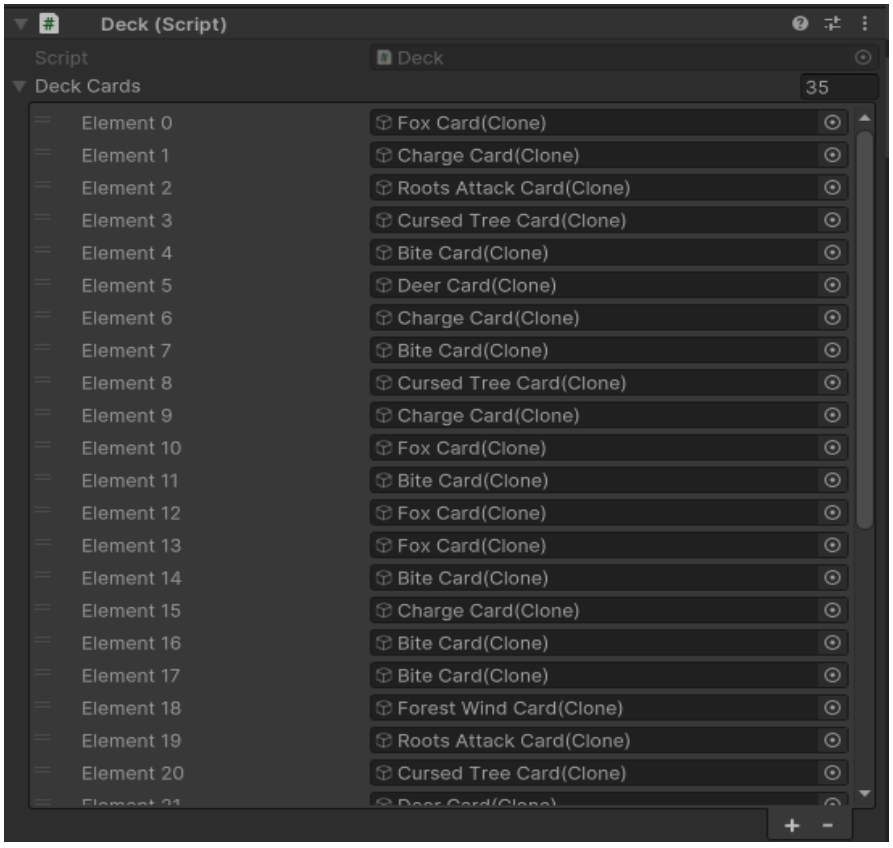
```
public class Deck : MonoBehaviour
{
    public List<GameObject> deckCards = new List<GameObject>();
    private const float _deckCardOffset = 0.0007f;

    public void LoadDeck(){}
    public void ShuffleDeck(){}
    public GameObject GetNextCard(){}
    public Vector3 GetNextTopDeckPosition(){}
    public Quaternion GetDeckCardRotation(){}
}
```

Svaki puta kada je potrebno učitati sve karte u špilu, poziva se metoda *LoadDeck* koja prolazi kroz svu djecu pripadajućeg objekta ove klase te se učitavaju u listu .Svaki puta kada je potrebno dohvatiti karte iz špila poziva se metoda *ShuffleDeck* koja učitava sve karte u listu a zatim se nad listom koristi varijacija Fisher-Yates algoritma za nasumično miješanje elemenata liste. Algoritam prolazi kroz svaku kartu špila i zamjenjuje ju sa drugom nasumičnom kartom. Ovim pristupom miješanja svaka karta ima jednaku šansu završiti na bilo kojoj poziciji u špilu, čineći algoritam pravednim načinom za miješanje špilova.

```
public void ShuffleDeck(){
        LoadDeck();
        for (int i = 0; i < deckCards.Count; i++){
            GameObject temp = deckCards[i];
            int randomIndex = UnityEngine.Random.Range(i,
deckCards.Count);
            Vector3 firstCardPosition = temp.transform.position;
            Vector3 secondCardPosition =
deckCards[randomIndex].transform.position;

            deckCards[i].transform.position = secondCardPosition;
```

```
            deckCards[randomIndex].transform.position =
firstCardPosition;
            deckCards[i] = deckCards[randomIndex];
            deckCards[randomIndex] = temp;
        }
        for (int i = 0; i < deckCards.Count; i++) {
            deckCards[i].transform.SetSiblingIndex(i);
        }
    }
```

Nadalje metoda *GetNextCard* izvodi se nakon *ShuffleDeck* metode i vraća zadnji element liste karata ako lista nije prazna te briše predanu kartu iz liste špila kako bi se karta mogla u potpunosti predati u igračevu ruku. Slijedi metoda *GetNextTopDeckPosition* koja za zadanu distancu između dviju karti špila vrši množenje varijable distance sa brojem djece i računa sljedeću poziciju karte na vrhu putem globalne točke y osi u prostoru koordinatnog sustava scene igre te vraća izračunatu točku. Na kraju se nalazi metoda *GetDeckCardRotation* koja vraća rotaciju špila kako bi karte koje se vraćaju u špil imale ispravnu rotaciju kao i špil.



Slika 21. Prikaz podatka špila

3.4.8 Polje za karte

Igraču je dostupno dvanaest slobodnih polja koje on može koristit kako bi na njih postavio i aktivirao karte iz svoje ruke. Gornjih šest polja ima kompatibilnost sa *Minion* kartama, dok donjih šest je kompatibilno sa *Modification* kartama. Svako polje ima referencu na kartu koja je postavljena na to polje kako bi se u bilo kojem trenutku moglo pristupiti karti za daljnje akcije i provjeriti postoji li već neka karta na polju na koje igrač želi odigrati novu kartu. Kako bi igraču bilo vizualno jasnije koje polje je dostupno za neku odabranu akciju, dodana je referenca na objekt *FieldSpotHighlight* koji se kreira na početku i putem metode aktivira u trenutku kada je potrebno. Objekt je oblika koji pokriva cijelo polje i sadrži transparentan materijal za čije se boje dinamički mijenjaju ovisno o tome je li polje dostupno ili nedostupno za igračevu akciju.

Slika 22. Označavanje polja za karte

```
public class FieldCardSlot : MonoBehaviour{
    public CardType compatibleCardType;
    public CardBase slotedCard;
    [SerializeField] private GameObject _fieldHightlightPrefab;
    [SerializeField] private GameObject _fieldHighlighter;
    [SerializeField] private Vector3 _highlighterScale;
    [SerializeField] private Color _noncompatibleColor = new
Color(0.26f, 0.78f, 0.38f);
    [SerializeField] private Color _compatibleColor = new
Color(0.478f, 0.811f, 0.949f, 0.5f);
```

```
    void Start(){
        SetHighlighterScale();
        SpawnHighlighter();
        _fieldHighlighter.SetActive(false);
    }
    private void SpawnHighlighter(){
        _fieldHighlighter = Instantiate(_fieldHightlightPrefab,
transform.position, transform.rotation, transform);
        _fieldHighlighter.transform.localScale = _highlighterScale;
    }
    private void SetHighlightMaterial(Color newColor){
        Material material =
_fieldHighlighter.GetComponent<Renderer>().material;
        material.color = newColor;
    }
    public void SetHighlighterScale(){
        _highlighterScale = new Vector3(0.045f, 0.067f, 0.004f);
    }
    public void RemoveHighlight(){
        _fieldHighlighter.SetActive(false);
    }
    public void Highlight(bool isCompatible){
        _fieldHighlighter.SetActive(true);
        if (isCompatible) {
            SetHighlightMaterial(_compatibleColor);
        }
        if (!isCompatible){
            SetHighlightMaterial(_noncompatibleColor);
        }
    }
    public bool CheckForOpenSlot(){
        if (slotedCard != null) { return false; }
        return true;
    }
}
```

Putem metode Start poziva se metoda *SetHighlightScale* i *SpawnHighlighter* kako bi se stvorio i postavio objekt za vizualno označavanje polja. Kada je objekt u stanju označenosti poziva metoda *Highlight* koja postavlja jednu od boja označavanja putem *SetHighlightMaterial* metode ovisno o kompatibilnosti sa odabranom akcijom koja se izvršava.

3.4.9 Polje na igraćoj ploči

Borba između igračevih i neprijateljskih figura izvodi se na igraćoj ploči koja se generira putem *GridManager* instance klase. Realizira se na način da na svakoj poziciji A x B ploče stvara se objekt sa pripojenom komponentom *GridSlot* klase na koji se kasnije može postaviti figura. Svako polje sadrži svojstva poput broja reda i stupca na kojem se polje nalazi, tim figure, tvrdnju o popunjenosti polja, strukture i objekte za upravljanje objektom za označavanje. Ovdje se nalazi i referenca na objekt *SquareNode* klase koja sudjeluje u algoritmu za pronalaženje optimalnog puta od početnog do ciljanog čvora koji će se obraditi kasnije u radu.

```
public class GridSlot : MonoBehaviour{
    public Color playerColor;
       …
    public SquareNode node;
    [SerializeField] private bool _isEmptySlot;
    [SerializeField] private GridSlotProperties _properties;
    [SerializeField] private Faction _slotedFaction;
    [SerializeField] private Vector3 _highlighterScale;
    [SerializeField] private GameObject _spotHightlightPrefab;
    [SerializeField] private GameObject _spotHighlighter;
    [SerializeField] private Miniature _slotedMiniature;
    private void Awake(){}
    private void Start(){}
    private void SpawnHighlighter(){}
    public void Highlight(HighlightMode mode){}
    public void RemoveHighlight(HighlightMode currentMode){}
    public void SetRangeHighlight(HighlightMode currentMode){}
    private void SetHighlightMaterial(Color newColor){}
    public void SetHighlighterScale(){}
    public void SetCellProperties(float slotSize, int row, int col){}
    public bool SetMiniatureToSlot(Miniature miniature, Faction
faction){}
    public void RemoveMiniatureInSlot(){}
    public void ClearGridSlot() {}
    public bool IsEmptySlot() {}
    public Miniature GetSlotedMiniature(){}
    public Faction GetMiniatureFaction() { }
    public GridSlotProperties GetCellProperties(){}
    public Faction SetHighlightMode(){}
}
```

Slično kao kod igračevih polja za karte, klasa upravlja objektom za označavanje mjesta kako bi igrač imao vizualni prikaz svojstva polja za odabranu akciju. Definirana je enumeracija koja predstavlja različite načine označavanja i koristi se kao ulazni parametar u metodama unutar klase odgovornima za upravljanje objektom oznake i definiraju kakav će biti vizualni prikaz objekta. Kada se objekt stvara pozivaju se *Awake* metoda i postavlja polje na prazno a zatim metoda Start koja stvara objekt za označavanje. Osnovna svojstva reda, stupca i veličine polja predaju se objektu putem *SetScellProperties* i zove se metode *SetHighligther* scale koja će osigurati ispravno veličinu objekta za označavanje jednaku veličini cijelog polja. Kada igrač ili neprijatelj postavlja figuru na polje, mora ispitati slobodnost polja putem *IsEmptySlot* metode i postaviti figuru pozivom *SetMiniatureToSlot* metode ili *RemoveMiniatureInSlot* ako je u pitanju brisanje figure sa polja.

Slika 23. Polje igrače ploče

U klasi su postavljene boje za označavanje prema kojima će se objekt istaknuti i vizualno igraču pojasniti svojstvo polja za vrijeme neke akcije. Kada igrač postavi kursor iznad jednog polja, poziva se metoda *Highlight* koja odlučuje kojom bojom je potrebno sada označiti objekt a kada više nije potrebno označavanje metodom *RemoveHighlight* označavanje se briše. Za vrijeme prikazivanja dometa neke akcije poziva se *SetRangeHighlight* metoda i time će automatski slobodna polja u dometu odmah biti označena a ako igrač pomiče kursor prema njima, biti će označena nekom drugom bojom. Ostale metode koriste se u svrhu postavljanja, dohvaćanja i provjere svojstva ove klase.

```
public void Highlight(HighlightMode mode){
        _spotHighlighter.SetActive(true);
        if (mode == HighlightMode.Passive){
            if (_isEmptySlot){
                SetHighlightMaterial(emptyColor);}
            else{
                if (_slotedFaction == Faction.Player){
                    SetHighlightMaterial(playerColor);
                }
                if (_slotedFaction == Faction.Enemy){
                    SetHighlightMaterial(enemyColor);}
            }
        }
        if (mode == HighlightMode.Attack){
            if (_slotedFaction == Faction.Enemy){
                SetHighlightMaterial(enemyColor);}
            else{
                SetHighlightMaterial(emptyColor);}
        if (mode == HighlightMode.Movement){
            if (_isEmptySlot){
                SetHighlightMaterial(emptyColor);}
            else{
                SetHighlightMaterial(enemyColor);}
        }
        if (mode == HighlightMode.DuelTarget){
            if (!_isEmptySlot){
                SetHighlightMaterial(duelTargetColor);
            }
        }
    }
```

3.4.10 Kontroleri

3.4.10.1 Kontroler za odabir karte napada

Kada igrač odabere figuru i pokreče akciju napada, na ekranu se pojavljuje izbornik svih karta napada trenutno povezanih na kartu pripadajuće figure. Igrač zatim koristi interaktivne gumbove za navigaciju kroz karte za napad, pregledava opis i odabir kartu koju želi iskoristiti. Nakon odabira, izbornik se zatvara i označavaju se polja na ploči koja su u dometu napada te igrač mora odabrati neku neprijateljsku figuru u dometu kako bi izvršio odabrani napad.

```
public class AttackPickerController : MonoBehaviour{
    [SerializeField] private GameObject _middleCard;
    [SerializeField] private GameObject _rightCard;
    [SerializeField] private GameObject _leftCard;
    [SerializeField] private GameObject _middleChoiceUI;
    [SerializeField] private GameObject _rightChoiceUI;
    [SerializeField] private GameObject _leftChoiceUI;
      …
    private void OnEnable(){}
    private void OnDisable(){}
    public void EnterModificationPickMode(){}
    public void EnterAttackMode(){}
    private void SetMiddleChoice(int id){}
    private void SetLeftChoice(int id){}
    private void SetRightChoice(int id){}
    public void SwapNextCard(){}
    public void SwapPreviousCard(){}
    public void ReturnCardsToOriginPoints(){}
    private void ResetAllChoices(){}
    private void HideUI() {}
}
```

Kontroler je referenciran i aktiviran metodom unutar *CardManager* klasa kada je to potrebno prema postavljenom trenutnom stanju unutar klase. Najprije se inicijalizira struktura koja predstavlja svojstva koja kontroler koristi kako bi izvršio operacije i ta svojstva se zapisuju unutar *CardManager* klase a zatim pozivom singleton instance dohvaćaju unutar kontrolera tijekom početka rada. Kontroler sadrži reference na objekte korisničkog sučelja koji se prikazuju igraču na ekranu i objekte koji predstavljaju točke na ekranu gdje će se pojaviti izbor trenutno odabrane, lijeve i desne karte napada figure. Metode *OnEnable* i *OnDisable* koriste

se za aktivaciju i deaktivaciju elemenata korisničkog sučelja, zaključivanje i otključavanje kontrole kamere. Metoda *EnterModificationPickMode* prva je metoda koja se poziva i dohvaća konfigurirana svojstva a zatim postavlja prve karte za pregled na odgovarajuća mjesta. Metoda *EnterAttackMode* koristi se nakon odabira karte kako bi se ona zapisala u svojstva i proslijedila drugom kontroleru za izvod napada te označuje sva polja igraće ploče u dometu karte drugom bojom i omogućuje igraču izbor neprijateljske figure koju će napasti.

Slika 24. Izgled ekrana za odabir napada

Slika 25. Označavanje polja odabirom napada

Prilikom pritiska interaktivnog gumba za sljedeću kartu u izborniku, poziva se metoda *SwapNextCard* koja radi na sličan princip kao i *SwapPreviousCard* kojom se odabire prijašnja karta. Dohvaćaju se odgovarajuće vrijednosti i dohvaća se zadnji indeks prikazane karte, odnosno desne karte na prikazu a zatim uspoređuje postojanost takvog indeksa karte na izboru. Do situacije nepostojeće karte se dolazi kada je prikazana zadnja desna karta i igrač ju želi odabrati, tada se karta pomiče na sredinu i desno je prikazana prazna karta što znači da je igrač trenutno odabrao zadnju kartu. Ako trenutna karta u sredini nije zadnja karta, karte se pomiču za jedno mjesto ulijevo kako bi nova desna karta mogla biti prikazana te se trenutna svojstva ažuriraju.

```
public void SwapNextCard(){
        AttackPickerProperties properties =
CardManager.Instance.GetAttackPickerProperties();
        List<ModificationCard> mods =
properties.selectedMinion.modificiationCards;
        if (properties.lastCardIndex == mods.Count + 1)
            return;
        properties.lastCardIndex++;
        ResetAllChoices();
        SetMiddleChoice(properties.lastCardIndex - 1);
        SetLeftChoice(properties.lastCardIndex - 2);

        if (properties.lastCardIndex < mods.Count){
            SetRightChoice(properties.lastCardIndex);
        }
        else{
            _rightChoiceUI.SetActive(false);
        }
   CardManager.Instance.UpdateAttackCardPickerProperties(properties);
 }
```

ReturnsCardsToOriginPoints metoda poziva se na kraju kako bi se karte koje trenutno sudjeluju u odabiru vratile na svoja izvorna mjesta prije otvaranja ovog izbornika. Ostale metode koriste se u svrhu postavljanja i otklanjanja karti sa odgovarajućeg mjesta, te skrivanje nekih elementa korisničkog sučelja u određenom vremenu.

3.4.10.2 Kontroler borbe između karata

Prilikom odabira karte kojom figura napada i figure na koju će karta napada utjecati, iz *CardManager* instance klase poziva se objekt sa komponentom *CardDuelController*. Kontroler je odgovoran za tok borbe u kojem se čitaju svojstva napadačke karte i prema njima izvrte brojevi zadanih kocki. Ovisno o napadačkim svojstvima figure koja napada, braniteljskim svojstvima figure koja se napada, brojevima na kocki, tipu štete i vrsti efekta, kontroler će se pobrinuti za kalkulacije te nakon izvoda borbe izvesti sve potrebne operacije nad odgovarajućim figurama.

```
public class CardDuelerController : MonoBehaviour{
    public MinionCard attackerCard;
    public MinionCard defenderCard;
    public ModificationCard modificationCard;

    [SerializeField] private int _totalRollNumber;
    [SerializeField] private int _nextRollId;
    [SerializeField] private bool _nextRollCycle;
    [SerializeField] private GridSlot attackerSlot;
    [SerializeField] private GridSlot defenderSlot;
    [SerializeField] private Dictionary<DiceType, GameObject>
_dicePrefabs = new Dictionary<DiceType, GameObject>();
    [SerializeField] private Dictionary<RollType, int> _diceRollOutputs;
    [SerializeField] private GameObject _diceRollUI;
        ...
    void Awake(){}
    void OnEnable(){}
    void Update() { }
    public void StartCardDuel(){}
    public void ResolveCardDuel(bool isDefenderAlive) {}
    public void RestoreCardTransformAfterDuel(bool isDefenderAlive){}
    public Dictionary<RollType, int> GetDiceRollOutputs() { }
    private void PrepareBattle(){}
    private void CalculateOutcome(RollStats roll, int rollNumber) { }
    private void SetDiceDictionary() { }
    private IEnumerator RollTheDice(List<RollStats> rolls, bool
moreRolls) {}
    private int GetDiceNumber(DiceType type){}
}
```

Kada se instanca skripte učitava poziva se metoda *Awake* i *SetDiceDictionary* kako bi se u rječnik kao vrijednosti postavile reference na pripadajući objekt korisničkog sučelja koji reprezentira jednu vrstu kocke koja se prikaže igraču i kao ključ dodaje se pripadajuća vrijednost iz enumeracije svih vrsta kocke. Kocke koje u igri postoje su d4,d6,d8,d10 i d20 čiji brojevi predstavljaju broj stranica kocke, odnosno u implementaciji bit će korišteni kao maksimalni inkluzivni broj u nasumičnom odabiru broja u rangu od 1 do maksimalnog broja. Igraču se na ekranu pojavljuje sučelje borbe u kojem su prikazane sve potrebne informacije o borbi. Izvrti se svaka potrebna kocka i napada se izvršava te su označene figure koje sudjeluju u napadu. Nakon svakog tipa štete ili efekta, prikazan je finalni izračun i klizač životnih bodova obrambene figure se spušta za određeni iznos.

Slika 25. Korisničko sučelje tijekom izvođenja napada

CardManager klasa poziva ovaj kontroler nakon zatvaranja izbornika odabira karte za napada tako što ga aktivira u hijerarhiji scene i poziva se metoda *OnEnable* koja inicijalizira neka interna svojstva na početne vrijednosti i aktivira odgovarajuće elemente korisničkog sučelja. Zatim se iz kontrolera poziva početna metoda *StartCardDuel* i ona dohvaća konfigurirana svojstva iz *CardManager* klase te prema njima postavlja preostale vrijednosti.

```
private IEnumerator RollTheDice(List<RollStats> rolls, bool moreRolls){
    yield return new WaitForSeconds(1f);
    foreach (RollStats roll in rolls){
        int diceNumber = GetDiceNumber(roll.diceType);
        int rollNumber = 0;
        for (int i = 0; i <= 30; i++){
            rollNumber = UnityEngine.Random.Range(1, diceNumber + 1);
            _diceText.text = rollNumber.ToString();
            if (i <= 20)
            { yield return new WaitForSeconds(0.025f);}
            if (i > 20 && i <= 23){yield return new WaitForSeconds(0.05f);}
            if (i > 23 && i <= 26){yield return new WaitForSeconds(0.1f);}
            if (i > 27){yield return new WaitForSeconds(0.2f);}
        }
        _diceRollOutputs.Add(roll.rollType, rollNumber);
        CalculateOutcome(roll, rollNumber);
        Tween tween = _diceText.transform.DOPunchScale(new Vector3(2.5f,
2.5f, 1f), 0.7f, 2, 1f);
        yield return tween.WaitForCompletion();
        yield return new WaitForSeconds(0.7f);}
    Tween tweeen = _damagePopupText.transform.DOPunchScale(new
Vector3(0.002f, 0.002f, 1f), 0.2f, 2, 1f);
    yield return tweeen.WaitForCompletion();
    if (rolls[0].rollType != RollType.EffectThrow){
        defenderCard.ManageHealth(-_totalRollNumber);

GameManager.Instance.ShowMinionCurrentHealthUI(defenderCard.minionStats,
defenderCard.miniature.movementPoints,
defenderCard.GetMinionMaximumHealth());}
    _totalRollNumber = 0;
    if (!moreRolls){
        ResolveCardDuel(defenderCard.IsMinionAlive());
        GameManager.Instance.CloseMinionCurrentHealthUI();
        gameObject.SetActive(false);}
    _nextRollCycle = moreRolls;
}
```

Korutina se izvodi unutar *Update* metode sve dokle god postoji još kocka za izvrtjeti. Ona će pomoći u sekvencijalnom izvođenju vrtnje kocke, omogućavajući igraču da određeno vrijeme vidi ishod svake kocke. Isto tako kontrolira globalne varijable iz klase koje *Update* metoda provjerava u svakom trenutku i ograničava da se sljedeća korutina izvodi tek nakon što trenutna završi. Za svaki ishod kocke izvrti se nasumičnih 30 brojeva koji se prikazuju igraču na ekranu, usporavajući prikaz svakog broja nakon nekoliko iteracija sve do zadnjeg koji se koristi za ishod. Broj se prosljeđuje *CalculateOutcome* metodi kako bi se izračunao pravi iznos ovisno o vrsti vrtnje kocke, napadačkim svojstvima jedne figure i obrambenim svojstvima druge. Ako postoji više kocka koje se bacaju za jednu vrstu vrtnje proces se ponavlja do kraja i ako sljedeća vrtnja ne postoji, poziva se metoda *ResolveDuel* koja će prema ishodu borbe izvršit određene operacije nad figurama i ostalim objektima koji su sudjelovali u duelu, postaviti interne vrijednosti na početne, vratiti karte na početna mjesta prije borbe i završiti duel.

3.4.10.3 Kontroler za korisničko sučelje statusa figure

MinionStatsController klasa upravlja prikazom informacija figure igraču kako bi saznao sve potrebne informacije o figuri svojeg ili neprijateljskog tima kada je potrebno.

```
public class MinionStatsController : MonoBehaviour
{
    [SerializeField] private Slider healthSlider;
    [SerializeField] private TextMeshProUGUI healthText;
    [SerializeField] private TextMeshProUGUI magicalArmorText;
    [SerializeField] private TextMeshProUGUI physicalArmorText;
    [SerializeField] private TextMeshProUGUI attackText;
    [SerializeField] private TextMeshProUGUI movementText;
    public void RefreshHealthBar(int currentHealth, int maxHealth){}
    public void ShowMinionStats(MinionStats stats, int miniatureMovement,
int maxHealth){}
}
```

Kontroler sadrži samo dvije metode od koji se uvijek izvan klase poziva *ShowMinionStats* metodu i popunjava tekst korisničkog sučelja proslijeđenim informacija u parametru metode. Ona u izvođenju poziva drugu metodu *RefreshHealth* koja upravlja prikazom omjera trenutnih i maksimalnih životnih bodova figure putem klizača koji se pojavljuje na ekranu.

3.4.10.4 Kontroler za pojavu figura i karata

SpawnController klasa odgovorna je za stvaranje i uništavanje figura i karata. Kontroleru se može pristupiti putem *GameManager* instance klase i poziva se u raznim trenutcima. Jedino što kontroler interno prati jest koji će biti sljedeći indeks nove kreirane karte i on se povećava za jedan nakon kreiranja objekta karte.

```
public class SpawnController : MonoBehaviour
{
    [SerializeField] private int _nextCardId = 0;
    [SerializeField] private Vector3 _minitarueScale = new Vector3(0.018f,
0.018f, 0.018f);
    public Miniature PointSpawnMiniature(MinionCard card, int row, int
column){}
    public Miniature RandomPointSpawnMiniature(MinionCard card){}
    public MinionCard SpawnMinionCard(ScriptableMinion minion, Deck deck,
Faction faction){}
    public ModificationCard
SpawnModificationCard(ScriptableModificationCard modification, Deck deck){}
    public void SpawnDeck(Dictionary<MinionName, int> minionCards,
Dictionary<MinionName, int> soulCards, Dictionary<ModificationName, int>
modificationCards) { }
    public void SpawnEnemies(ScriptableEnemyEncounter encounter) { }
    public void DespawnModificationCard(ModificationCard card) { }
    public void DespawnMinion(MinionCard card) { }
    private Vector3 GetMiniatureRotation(int col){}
}
```

Metoda *RandomSpawnMiniature* i *PointSpawnMiniature* rade na sličan princip ali se razlikuju u tome hoće li se figura stvoriti i postaviti na određeno polje x reda i y stupca, ili će se nasumično pokušati dodijeliti polje koje nije popunjeno figurom. Uzmemo li za primjer implementaciju metode *PointSpawnMiniature*, kroz parametre metode prosljeđuje se karta, broj stupca i broj reda a zatim se iz *GridManager* instance klase dohvaća pripadajuće polje ploče. Figuri se postavlja rotacija i pozicija u koordinatnom sustavu scene te se iz proslijeđene karte uzima referenca na objekt 3D modela figure i stvara se pomoću *Unity* metode *Instantiate*. Na kraju se objektima popune potrebne vrijednosti nakon stvaranja figure i koriste se *DOTween* metode za vizualne ulazne animacije.

```
public Miniature PointSpawnMiniature(MinionCard card, int row, int column)
    {
        Miniature miniature;
        GridSlot slot = GridManager.Instance.GetGridSlot(row, column);
        if (slot.IsEmptySlot()){
            Vector3 rotation = GetMiniatureRotation(column);
            Vector3 spawnPosition = new Vector3(slot.transform.position.x,
slot.transform.position.y + 1f, slot.transform.position.z);
            GameObject miniatureObject =
Instantiate(card.GetMiniaturePrefab(), spawnPosition,
Quaternion.Euler(rotation.x, rotation.y, rotation.z), transform);
            miniature = miniatureObject.GetComponent<Miniature>();
            miniature.cardReference = card;
            miniature.RestoreAction();
            slot.SetMiniatureToSlot(miniature, card.faction);
            miniatureObject.transform.localScale = _minitarueScale;
            miniatureObject.transform.DOMove(slot.transform.position,
0.6f).OnComplete(() => {
                miniatureObject.transform.DOShakeRotation(1f, 10f, 10, 90f,
true, ShakeRandomnessMode.Harmonic);
            });
            return miniature;
        }
        return null;
    }
```

Kada se stvaraju karte, pozivaju se metode *SpawnMinionCard* i *SpawnModification* za kreiranje instance objekta određene karte na temelju proslijeđenog skriptablnog objekta a zatim zovu unutarnju metodu karte i prosljeđuje podatke kroz parametre kako bi se karta u potpunosti postavila. *SpawnDeck* metoda kao parametre prihvaća liste svih karti pripadajućih špilova i prolazi kroz svaku kartu u pojedinom špilu te poziva odgovarajuću metodu za stvaranje objekta karte. *SpawnEnemies* metoda prihvaća skriptabilni objekt u kojem su definirani podaci koji određuju koje figure će biti na neprijateljskoj strani, koliko će njih biti za svaku vrstu i koje će napade figura moći koristiti. Za *Modification**Card*** i *Minion**Card*** karte postoje i metode koje upravljaju pravilnim uništenjem njihovih objekata. Posljednja metoda *GetMiniatureRotation* pomaže kod određivanja početne rotacije figure, ovisno o odabranoj polovici na igraćoj ploči na koju se ona postavlja.

3.4.11 Menadžeri

3.4.11.1 Menadžer za karte

CardManager klasa predstavlja središnju komponentu za upravljanje interakcijom između karata i figura unutar igre.

```
public class CardManager : Singleton<CardManager>{
    public AttackPickerController attackPickerController;
    public CardDuelerController cardDuelerController;
    public bool canScaleCards = true;
    [SerializeField] private AttackPickerProperties
_attackPickerProperties;
    [SerializeField] private CardDuelProperties _cardDuelProperties;
    [SerializeField] private Miniature _selectedMiniature;
    ...
    [SerializeField] private List<FieldCardSlot> _tributeSlots = new
List<FieldCardSlot>();
    [SerializeField] private List<FieldCardSlot> _fieldSlots = new
List<FieldCardSlot>();
    public void ActivateCard() { }
    public bool SummonMinion() { }
    public void HighlightHoveredCard() { }
    public void ReturnHoveredCard() { }
    public void PrepareCardHovering() { }
    public void DetectFieldSlotForTributeMinion() { }
    IEnumerator TributeSelectedMinions() { }
    public void CancelMinionTributing() { }
    public void DetectFieldSlotForCardActivation() { }
    public bool TryPlaceCardOnField() { }
    public void TrySelectTributeMinion() { }
    public bool TrySelectHoveredCard() { }
    private void TryCleanLastHitFieldSlot() { }
    public void ToggleSelectedCard() { }
    public void ToggleAttackPickMode(bool toggleOn) { }
    private void CreateAttackCardPickerProperties(MinionCard minion) { }
    public void UpdateAttackCardPickerProperties(AttackPickerProperties
properties) { }
    public void ClearAttackPickerProperties() { }
    public void CreateCardDuelProperties(MinionCard attacker, MinionCard
defender, ModificationCard modificationCard) { }
    public CardDuelProperties GetCurrentCardDuelProperties() { }
    public void ClearCardDuelProperties() { }
```

```
    private void SaveModificationCardOriginPoints() { }
    public void EnterMovementChoiceMode() { }
    public void EnterCardDuelMode() { }
    public void TransfromCardToMousePoint() { }
    public void OpenMinionUI(Miniature miniature, Faction faction) { }
    public void CloseMinionActionsUI() { }
    public bool IsMinionTributeRequired() { }
    public bool IsMiniatureSelected() { }
    public bool IsInCardDuelMode() { }
    public CardBase GetLastHitCard() { }
    public CardType GetSelectedCardType() { }
    public Miniature GetSelectedMiniature() { }
    public List<FieldCardSlot> GetFieldSlots() { }
    public AttackPickerProperties GetAttackPickerProperties() { }
    public void SetModPickerLastCardIndex(int index) { }
}
```

U klasi postoji puno unutarnjih varijabli koje prate razne zadnje odabrane objekte tijekom igre kako bi ostale metode i klase u svoje vrijeme mogle izvesti operacije pomoću tih zapamćenih objekata. Kada se karta postavi na polje potrebna je aktivacija njenih namjera i poziva se *ActivateCard* metoda koja je odgovorna za aktivaciju funkcije karte i ovisno o karti izaziva različite akcije. Ona provjerava tip odabrane karte i ako je karta tipa *MinionCard*, igrač može pozvati kartu na prazno polje za karte i prelazi na pogled igrače ploče gdje odabire mjesto za pozivanje figure. Ako je odabrana karta tipa *ModificationCard*, ona se također postavlja na prazno polje i zahtijeva daljnju akciju od igrača da odabere kompatibilnu kartu na koju će se modifikacija pripojiti. *SummonMinion* metoda poziva novu figuru na polje ako je to moguće i pokušava postaviti figuru na zadano polje igraće ploče, pri čemu provjerava postoji li takvo polje sa danim koordinatama i jeli to polje slobodno. Ako ono jest slobodno, figura se stvara i polje se popunjava te se figura pridodaje listi igračevih figura. S obzirom da neprijateljske figure direktno koriste kontrolere za stvaranje svih figure u prvom potezu, one ovu metodu ne koriste.

```
    public bool SummonMinion(){
        GridSlot slot = GridManager.Instance.GetLastHitGridSlot();
        if (slot == null || !slot.IsEmptySlot()) { return false; }
        GridSlotProperties slotProperties = slot.GetCellProperties();
        int row = slotProperties.row;
        int column = slotProperties.column;
        MinionCard minionCard = _selectedCard.GetComponent<MinionCard>();
        minionCard.miniature =
GameManager.Instance.GetSpawnController().PointSpawnMiniature(minionCard,
row, column);
        _lastHitFieldSlot.slotedCard = _selectedCard;
        _lastHitFieldSlot = null;
        _lastCardSlot.SetActive(false);
        _lastCardSlot = null;
        PlayerManager.Instance.playerMinions.Add(minionCard);
        return true;
    }
```

HighlightHoveredCard metoda označuje kartu koja se trenutno nalazi ispod kursora miša i označava ju ako je to moguće. Također skalira kartu i pomiče ju po koordinatnoj osi kako bi odabrana karta bila više istaknuta igraču. Unutar klase postoji više metoda koje kao i ova koriste *RayCast* zrake za determiniranje postojanja traženog objekta ispod kursora miša te ga pronalaskom označuju. Objekt koji je trenutno označen ne može ostati u takvom stanju nakon što miš ne prepoznaje kolizijsko svojstvo elementa ispod sebe i zato se uvijek prati zadnji označeni objekt kojem će metoda maknuti označenost nakon što se prepozna novi objekt za označavanje ili uopće ne postoji traženi objekt ispod kursora. Metode sa slično implementacijom pozivaju se iz instance klase *PlayerManager* ovisno o konkretnom praćenom stanju unutar menadžer klase a izvode se kroz *Update* metodu što znači da se metoda izvodi svaki okvir igre.

```
    public void HighlightHoveredCard(){
    if (!canScaleCards) { return; }
    RaycastHit hit = RayCaster.CastRay();
    if (hit.collider != null){
        CardBase hitCard =
hit.collider.gameObject.GetComponentInChildren<CardBase>();
        if (hitCard != null){
            if (hitCard.isPlaced)
                return;
            if (_lastHitCard != null && _lastHitCard != hitCard) {
                if (_lastHitCard.isScaled == true){
                    _lastHitCard.NormalizeScale();}
```

```
            }
            _lastHitCard = hitCard;
            _lastHitCard.ScaleCard();
        }
        else{
            if (_lastHitCard != null && _lastHitCard.isScaled == true){
                _lastHitCard.NormalizeScale();
                _lastHitCard = null;}
        }
    }
    else{
        if (_lastHitCard != null && _lastHitCard.isScaled == true){
            _lastHitCard.NormalizeScale();
            _lastHitCard = null;}
    }
}
```

ReturnHoveredCard metoda vraća kartu na izvornu poziciju nakon što je ona označena i postavljena na polje ali nije prošla kroz cijeli proces postavljanja za taj tip karte. Kada se karta vrati ili kada je je selekcija karata u ruci isključena, potrebno je pozvati metodu *PrepareCardHovering* kako bi se ponovno postavila selekcija. *DetectFieldSlotForTributeMinion* metoda detektira polja na čijim se mjestima nalaze karte figure koje mogu poslužiti kao karte za žrtvovanje da se prizove nova karta figure. Pomoću *RayCast* zrake označava se i tako prikazuju igraču sva odabrana polja za žrtvovanje. Ako se proces žrtvovanja želi zaustaviti prije odabira finalne karte, može se pozvati *CancelMinionTributing* metoda. Kao i prijašnje spomenuta metoda za detekciju polja karata, *DetectFieldSlotForCardActivation* metoda radi na sličan princip i detektira polja karata koja se mogu koristiti kako bi se odabrana karta aktivirala i u potpunosti postavila. *TributeSelectedMinions* metoda izvršava proces žrtvovanja odabranih karta. Uklanja odabrane karte sa polja karata, dodavajući ih natrag u igračev špil karata te se karte mogu kasnije dobiti natrag u ruku ali sa istim svojstvima koje su imale prije žrtvovanja.. Odabranim kartama miče označenost i vraća trenutno odabranu kartu za odigravanje natrag *igraču* u ruku. Sljedećih troje metoda *TryPlaceCardOnField*, *TrySelectTributMinion* i TrySelectHoveredCard se također izvode na sličan princip. Prva metoda pokušava postaviti kartu na polje ako je to moguće, druga metoda pokušava odabrati karte figura za žrtvovanje u skladu s svojstvima karte koja se priziva, dok treća metoda pokušava odabrati kartu koja je trenutno označena ispod kursora miša i postavlja ju na mjesto koje se uvijek nalazi ispod kursora te igraču daje vizualnu reprezentaciju odabrane karte iz ruke.

```
    public bool TrySelectHoveredCard()
    {
        if (_lastHitCard == null) { return false; }
        canScaleCards = false;
        _selectedCard = _lastHitCard;
        _selectedCard.NormalizeScale();
        _lastCardSlot = _selectedCard.transform.parent.gameObject;
        _selectedCard.transform.eulerAngles = new Vector3(50f, 90f, 0f);
        _selectedCard.transform.position = new
Vector3(_selectedCard.transform.position.x - 0.05f,
_selectedCard.transform.position.y - 0.05f,
_selectedCard.transform.position.z);
        _selectedCard.transform.parent = Camera.main.transform;
        return true;
    }
```

TransformCardToMouse je jednostavna metoda koja svoje izvođenje ima unutar *Update* metode i ako postoji trenutno odabrana karta iz igračeve ruke, pratiti će i postavljati poziciju odabrane karte na jednaku poziciji kao i kursor miša, postavljajući tu poziciju u svakom okviru igre dokle se ne izvede daljnja akcija.

```
    public void TransfromCardToMousePoint(){
        if (_selectedCard != null && !_selectedCard.isPlaced){
            Vector3 position = new(Input.mousePosition.x,
Input.mousePosition.y,
Camera.main.WorldToScreenPoint(_selectedCard.transform.position).z);
            Vector3 worldPosition =
Camera.main.ScreenToWorldPoint(position);

            _selectedCard.transform.position = new Vector3(worldPosition.x,
worldPosition.y, worldPosition.z);
        }
```

ToggleAttackPickMode i *EnterCardDuel* metode koriste se za pozivanje i aktivaciju prije navedenih kontrolera koji tada izvode svoju svrhu te se ponovne deaktiviraju do sljedećeg poziva. *EnterMovementChoice* metoda ne poziva kontroler ali poziva metodu *GridManager* instance klase koja će označiti sva polja u dometu pomaka figure i omogućiti igraču odabir jednog od prikazanih polja za pomak svoje figure. Ostale metode koriste se za prikaz i sakrivanje elementa korisničkog sučelja igraču te za postavljanje, dohvaćanje i brisanje određenih varijabli unutar klase.

3.4.11.2. Menadžer za igraću ploču

Menadžer za igraću ploču predstavlja središnju komponentu za stvaranje i upravljanje igraćom pločom i pojedinim elementima ploče. Ploča se sastoji od x redova i y stupaca, veličine svakog polja, linija koje stvaraju rešetke i objekata za označavanje polja na ploči.

```
public class GridManager : Singleton<GridManager>{
    public Miniature movingMiniature;
    [SerializeField] private bool _drawGizmos;
    [SerializeField] private int _rowSize;
    [SerializeField] private int _columnSize;
    [SerializeField] private float _cellSize = 0.09f;
    [SerializeField] private HighlightMode _highlightMode;
    [SerializeField] private GameObject _linePrefab;
    [SerializeField] private GameObject _slotPrefab;
    [SerializeField] private Transform _gridSpawnPoint;
    [SerializeField] private GridSlot lastHitGridSlot;
    [SerializeField] private List<GridSlot> _slotList = new
List<GridSlot>();
    [SerializeField] private List<GridSlot> _slotsInRange = new
List<GridSlot>();
    public void GenerateGrid() { }
    private void DrawLine(Transform parent, Vector3 startPos, Vector3
endPos) { }
    public IEnumerator MiniatureMovement (GridSlot nextGridSlot){ }
    public void CheckForSelectedGridSlot() { }
    public void ClearGridHightlight() { }
    public void SetHighlightToLimitedRange(Miniature miniature, int range,
HighlightMode mode) { }
    public void RestoreHighlightToSelectRange() { }
    public void CacheSlotNeighbors() { }
    public GridSlot GetClosestPlayerMinion(GridSlot startingSlot) { }
    public List<GridSlot> GetSlotsInRange(Miniature miniature, int range){}
    public GridSlot GetGridSlot(int row, int col) { }
    public List<GridSlot> GetGridSlots() { }
    public GridSlot GetRandomSlotPoint(Faction faction) { }
    public Vector2 GetGridSize() { }
    public GridSlot GetLastHitGridSlot() { }
    public int GetGridRowSize() { }
    public int GetGridColumnSize() { }
}
```

Prva i glavna metoda ove klase je *GenerateGrid* koja je odgovorna za stvaranje mreže veličine zadanih parametara reda i stupca. Prvo se postavlja početna pozicija mreže temeljena na danoj točki u koordinatnom sustavu, parametrima reda i stupca te veličine polja. Stvori se roditeljski objekt za mrežu, jedno dijete za polja i drugo dijete za rešetke između polja na koja će se postavljati pripadajuća stvoreni objekti. Zatim se prolazi kroz svaki stupac reda i stvara se novi objekt sa komponentom *GridSlot* klase, inicijaliziraju se svojstva te komponente prema trenutnim koordinatama i objekt se tada dodaje listi svih polja u mreži. Nakon svih kreiranih polja zasebno se prolazi kroz svaki red i stupac te se izračunava pozicija na kojoj će se linija između svakog polja provući. Za svaku liniju poziva se metoda *DrawLine* koja će postaviti početnu i krajnju poziciju svake linije i u konačnici vizualno prikazati granice svakog polja ploče. Na kraju se poziva metoda *CacheSlotNeighbors* koja je nužna za provođenje algoritma za pronalaženje optimalnog puta za neprijateljske figure. Ona za svaki objekt polja poziva metodu *CacheNeighbors* iz komponente *SquareNode*.

```
public void GenerateGrid()
{
    Vector3 waypointPosition = _gridSpawnPoint.position;
    float startX = waypointPosition.x - (_rowSize - 1) * _cellSize * 0.5f;
    float startY = waypointPosition.y;
    float startZ = waypointPosition.z - (_columnSize - 1) * _cellSize *
0.5f;
    GameObject gridParent = new("Grid");
    GameObject gridSlots = new("Slots");
    GameObject gridLines = new("Lines");
    _highlightMode = HighlightMode.Passive;
    gridLines.transform.parent = gridParent.transform;
    gridSlots.transform.parent = gridParent.transform;
    for (int x = 0; x < _rowSize; x++){
        for (int z = 0; z < _columnSize; z++){
            Vector3 point = new(startX + x * _cellSize, startY, startZ + z
* _cellSize);
            GameObject _slot = Instantiate(_slotPrefab, point,
Quaternion.identity, gridSlots.transform);
            GridSlot slotScript = _slot.GetComponent<GridSlot>();
            slotScript.SetCellProperties(_cellSize, x, z);
            SquareNode node = _slot.AddComponent<SquareNode>();
            node.SetCoordinates(x, z);
            slotScript.node = node;
            _slotList.Add(_slot.GetComponent<GridSlot>());
        }
    }
```

```
    for (int z = 0; z <= _columnSize; z++){
        Vector3 cellStartPos = new Vector3(startX, startY, startZ + z *
_cellSize);
        Vector3 cellEndPos = new Vector3(startX + (_rowSize - 1) *
_cellSize, startY, startZ + z * _cellSize);
        Vector3 lineStartPos = new Vector3(cellStartPos.x - (_cellSize /
2), cellStartPos.y, cellStartPos.z - (_cellSize / 2));
        Vector3 lineEndPos = new Vector3(cellEndPos.x + (_cellSize / 2),
cellEndPos.y, cellEndPos.z - (_cellSize / 2));
        DrawLine(gridLines.transform, lineStartPos, lineEndPos);
    }
    for (int x = 0; x <= _rowSize; x++){
        Vector3 cellStartPos = new Vector3(startX + x * _cellSize, startY,
startZ);
        Vector3 cellEndPos = new Vector3(startX + x * _cellSize, startY,
startZ + (_columnSize - 1) * _cellSize);
        Vector3 lineStartPos = new Vector3(cellStartPos.x - (_cellSize /
2), cellStartPos.y, cellStartPos.z - (_cellSize / 2));
        Vector3 lineEndPos = new Vector3(cellEndPos.x - (_cellSize / 2),
cellEndPos.y, cellEndPos.z + (_cellSize / 2));
        DrawLine(gridLines.transform, lineStartPos, lineEndPos);
    }
    CacheSlotNeighbors();
}
```

CheckForSelectedGridSlot metoda poziva se unutar *Update* metode *PlayerManager* klase za svaki okvir i koristi *RayCast* zraku koja se baca iz pozicije kursora miša i provjerava postoji li ispod kursora kolizijski objekt sa komponentom *GridSlot* klase. Ako takav objekt postoji, označava ga ili ako ne postoji miče oznaku sa zadnjeg zapamćenog objekta označavanja. Na izvod ove metode također utječe i metoda *SetHighlightToLimitedRange* koja se poziva ako se označavanje ćelija limitira na određeni domet. Ako se kasnije **tijekom igre** pozove metoda *RestoreHighlightToSelectRange* označavanje će se vratiti na normalno stanje. Kada postoji posebna potreba za brisanjem zadnje ćelije pozvat će se metoda *ClearGridHighlight* koja će dohvatiti referencu na zadnju označenu ćeliju i isključiti oznaku. Kada se figura pomiče po ploči za određenu distancu na neko polje, pokreće se korutina *MiniatureMovement* sa proslijeđenim poljem putem parametra i započinje usporedba između trenutne i odredišne ćelije te ovisno o stupcu okreće figuru za 180 stupnjeva. Zatim se figura pomiče putem *DoTween* metode za skok, podaci se zapisuju u novo polje i brišu iz starog polja figure, a njoj se vrijednost za kretnju postavlja na nulu jer se ona taj potez više ne smije pomicati.

Kada je neprijateljska figura na potezu, poziva se metoda *GetClosestPlayerMinion* kako bi se od početnog polja za svaku ćeliju izračunala distanca između polja figure i svakog pojedinog polja na ploči. Ako je zadnja izračunata distanca veća od trenutnog polja, polje se zapisuje kao najbliže polje i nakraju se vrača najbliže polje igračeve figure koju neprijatelj može napasti. Ostale metode klase koriste se za dohvaćanje i postavljanje internih svojstva ploče.

```
public GridSlot GetClosestPlayerMinion(GridSlot startingSlot){
        GridSlot closestSlot = null;
        int distanceValue = -1;
        for (int row = 0; row < _rowSize; row++){
            for (int column = 0; column < _columnSize; column++){
                GridSlot currentSlot = GetGridSlot(row, column);
                if (currentSlot.IsEmptySlot()){
                    continue;
                }
                if (currentSlot.GetMiniatureFaction() != Faction.Player){
                    continue;
                }
                int rowDistance =
Mathf.Abs(startingSlot.GetCellProperties().row -
currentSlot.GetCellProperties().row);
                int columnDistance =
Mathf.Abs(startingSlot.GetCellProperties().column -
currentSlot.GetCellProperties().column);
                int curDistanceVal = rowDistance + columnDistance;
                if(distanceValue == -1){
                    distanceValue = curDistanceVal;
                    closestSlot = currentSlot;
                }
                else {
                    if (curDistanceVal < distanceValue){
                        distanceValue = curDistanceVal;
                        closestSlot = currentSlot;
                    }
                }
            }
        }
        return closestSlot;
    }
```

3.4.11.3. Menadžer za igrača

Klasa *PlayerManager* ima odgovornost upravljanja i koordinacije ponašanja igrača tokom izvađanja igre. Osnovna svrha jest pratiti i upravljati akcijama koje igrač može poduzeti tokom svojeg poteza uključujući interakciju s kartama, figuricama i slično.

```
public class PlayerManager : Singleton<PlayerManager>{
    public Deck playerDeck;
    public Deck soulDeck;
    public PlayerState playerState;
    public List<MinionCard> playerMinions = new List<MinionCard>();
    [SerializeField] private bool _isPlayerTurn = false;
    [SerializeField] private GameObject _slotPrefab;
    [SerializeField] private GameObject _minionActionsUI;
    [SerializeField] private Transform _deckDrawPoint;
    [SerializeField] private Transform _cardShowPoint;
    [SerializeField] private Transform _handPoint;
    [SerializeField] private List<GameObject> _playerHand = new
List<GameObject>();
    void Update() { }
    public void StartPlayerTurn(bool isPreparationTurn) { }
    public void EndPlayerTurn() { }
    private void HandlePlayerState() { }
    private void ApplyEffects() { }
    private void RestoreMinionActions() { }
    IEnumerator ClearPlayerHand() { }
    IEnumerator PreparePlayerHand(bool isPreparationTurn) { }
    IEnumerator DrawCardsFromDeck(int cardDraws, Deck decktoDraw) { }
    private void SendCardToDeck(GameObject card, Deck deck) { }
    private void HandlePlayerInput() { }
    private void HandleLeftMouseClick() { }
    private void HandleRightMouseClick() { }
    private void HandleMouseScroll(float scrollInput) { }
    private List<GameObject> GetNextHandSlots(int currentSlotSize) { }
}
```

Singleton instanca ove klase sadrži referencu na igračev špil i špil duša, interno stanje igrača, listu svih pripadajućih stvorenih figura, reference na razne objekte za rad, listu za karta u ruci i varijablu tvrdnje o igračev redu za potez koja dozvoljava provjeru internog stanja igrača i izvršavanje prikladnih operacija.

Kada dođe igračev potez, *GameManager* instanca klase poziva *StartPlayerTurn* metodu koja postavlja spomenutu tvrdnju na istinitost i dozvoljava izvršavanje unutar metode

Update a zatim prvo pokreće metodu *RestoreMinionActions*. Za svaku kartu figure dozvoljava se ponovno napadanje i pomicanje na ploči a tada se putem metode *ApplyEffects* na pripadajuće figure primijene svi postavljeni efekti. Ako su životni bodovi figure ispod nule, figura se dodaje u listu figura za destrukciju i kasnije se svaka pravilno obriše iz igre. Na kraju se poziva korutina *PreparePlayerHand* koja pričeka sekundu kako bi se igraču na ekranu na određeni vremenski period pojavio tekst preostalih efekata. Tada preko *CameraManager* instance klase postavlja pogled kamere na ruku a zatim zaključava promjenu pogleda i skaliranja karata.

Prije pravih poteza svaka strana ima pripremni potez. Kod neprijatelja taj potez se koristi za stvaranje neprijateljskih figura na ploči, a igrač dobiva šest karta duše koje može postaviti dok u slučaju običnog poteza izvlači pet karti iz igračevog špila i dvije karte iz špila duša. Kada igrač izvlači kartu iz špila poziva se korutina *DrawCardsFromDeck* i preko parametara prosljeđuje se broj karata za izvlačenje te špil iz kojeg se karte izvlače. S obzirom da igrač vuče karte iz dva špila, potrebno je pratiti koliki je trenutni broj karata u ruci i broj karta koje će biti nakon izvlačenja. Zatim se svaka potrebna karta dohvaća iz špila, sekvencijalno se pomiče na određene točke u prostoru i prikazuje izvlačenje karte iz špila u ruku. Metoda *GetNextHandSlot* poziva se kada karta dođe iznad igračeve ruke i kalkulira distancu svake karte od sredine. Tako je moguće održati pravilan razmak između karti koje se uvijek nalaze na donjoj polovici sredine ekrana. Na kraju se vrača lista objekata polja u ruci sa kolizijskom komponentom koja će pomoći odrediti pravu poziciju karte nakon što se ona označi i skalira. Vraćena lista uspoređuje se s listom trenutnih polja igračeve ruke i sve karte se pomiču na svoju novu lokaciju te postaju djeca objekata novih polja. Nova karta postavlja se na najzadnji objekt liste i animira se rotacija i pomicanje karte u igračevu ruku, a stara lista igračeve ruke se briše i popunjava se novim poljima.

```
IEnumerator DrawCardsFromDeck(int cardDraws, Deck decktoDraw){
    int currentSlots = _playerHand.Count;
    int handSize = currentSlots + cardDraws;
    for (int i = 0; i < handSize; i++){
        if (currentSlots < handSize){
            currentSlots++;
            Tween tween;
            GameObject nextCard = decktoDraw.GetNextCard();
            if (nextCard == null)
                break;
```

```
            tween = nextCard.transform.DOMove(_deckDrawPoint.position,
0.4f);
            yield return tween.WaitForCompletion();
            nextCard.transform.SetPositionAndRotation(new
Vector3(_cardShowPoint.transform.position.x - 1,
_cardShowPoint.transform.position.y, _cardShowPoint.transform.position.z +
5), _cardShowPoint.transform.rotation);
            tween =
nextCard.transform.DOJump(_cardShowPoint.transform.position, 0.1f, 1, 0.3f,
false);
            yield return tween.WaitForCompletion();
            yield return new WaitForSeconds(1);
            List<GameObject> newPlayerHand =
GetNextHandSlots(currentSlots);
            if (currentSlots != 1){
                for (int j = 0; j < currentSlots - 1; j++) {
                    Transform cardTransform =
_playerHand[j].transform.GetChild(0).gameObject.transform;

cardTransform.DOMove(newPlayerHand[j].transform.position, 0.1f);
                    cardTransform.parent = newPlayerHand[j].transform;
                }
            }
            nextCard.transform.DORotate(newPlayerHand[currentSlots -
1].transform.rotation.eulerAngles, 0.2f);
            tween = nextCard.transform.DOMove(newPlayerHand[currentSlots -
1].transform.position, 0.1f);
            yield return tween.WaitForCompletion();
            nextCard.transform.parent = newPlayerHand[currentSlots -
1].transform;
            foreach (GameObject slot in _playerHand){
                Destroy(slot);
            }
            _playerHand.Clear();
            foreach (GameObject slot in newPlayerHand) {
                _playerHand.Add(slot);
            }
        }
    }
}
```

Nakon što igrač izvuče karte**,** započinje njegov potez i putem metode *HandPlayerState* prati se stanje u kojem se nalazi igrač. Na početku poteza, postavljeno je *Strategizing* stanje i dozvoljava se izvođenje metode *HigihlightHoveredCard* koja će označavati karte iz ruke. Svoje akcije igrač potvrđuje lijevim klikom gumba i poziva se *HandleLeftMouseClick* metoda koja će na temelju unutarnjeg stanja odrediti operacije koje će se izvesti za igračevu akciju. Ako je stanje igrača još početno i igrač odabere kartu za odigravanje, provjerava se tip karte i ako je karta figure promijenit će stanje u *TributeMinion* gdje igrač mora odabrati karte za žrtvovanje putem metode *DetectFieldSLotForTributeMinion*. Nakon što igrač odabere sve potrebne karte za žrtvovanje, karte se vraćaju u pripadajući špil i prelazi se u stanje *CardPlacement* ili je odigrana karta duše koja ne zahtijeva žrtvovanje direktno se prelazi u spomenuto stanje. Igrač sada mora odabrati na koje mjesto će se postaviti karta. Za vrijeme tog stanja poziva se *CadManager* instanca klase i metode *TransformCardToMousePoint* kako bi odabranu kartu mogli pomicati kursorom miša. Zatim se zove *DetectFieldSlotForCardActivation* metoda koja će dozvoliti i označiti prazna i kompatibilna mjesta na koje se karta može postaviti.

Nakon postavljene karte pogled se prebacuje na igraču ploču, zaključava se promjena pogleda i stanje se mijenja na *SummonMinion*. Zbog pogleda na igraću ploču dozvoljeno je pozivanje metode *CheckForSelectedGridSlot* iz *GridManager* instance klase koje će pratiti i označiti trenutno selektirano polje na ploči. Na zadnje odabrano polje nakon lijevog klika miša pokušat će se postaviti nova figura ako to polje nije već zauzeto. Tada se stanje vraća na početno stanje *Strategizing* i kamera se postavlja na pogled igračeve ruke.

Kada je figura postavljena i pogled kamere je na ploči moguće je lijevim klikom odabrati sve stvorene figure te se otvara korisničko sučelje i prikazuju se podaci o figuri. Osim prikaza podataka svojstva figure i klizača sa trenutnim životnim bodovima, ako igrač odabere figuru svojeg tima, može putem interaktivnog gumba odabrati akciju napada ili pomaka. Za akciju napada iz *CardManager* instance klase pozvat će se prijašnje spomenuti kontroler za odabir karte napada i nakon odabira pojavit će se označena polja ploče u dometu napada od kojih igrač mora odabrati jedno sa neprijateljskom figurom a stanje će biti promijenjeno na *RangeSelection*. Kod pritiska lijevog gumba za odabir figure i ispravno odabranog polja na ploči poziva se drugi kontroler i karte ulaze u duel, nakon čega se stanje vrača na početno. Druga akcija koja se može izvršiti je postavljanje stanja na *MovementRangeSelection* kada se označavaju polja u dometu na ploči na koja igrač može pomaknuti odabranu figuru. Kada je pritisnut lijevi klik miša na odabrano prazno polje, iz *GridManager* instance klase poziva se korutina *MiniatureMovement* koja figuru pomiče i animira pomak nakon čega se stanje ponovno vraća na početno.

```
private void HandleLeftMouseClick(){
    if (playerState == PlayerState.Strategizing &&
CardManager.Instance.GetLastHitCard() != null){
        if (CardManager.Instance.TrySelectHoveredCard()){
            if (CardManager.Instance.GetSelectedCardType() ==
CardType.Minion) {
                if (CardManager.Instance.IsMinionTributeRequired()) {
       CameraManager.Instance.ChangeCurrentViewToNew(CameraView.FieldView);
                    playerState = PlayerState.TributeMinions;
                    CardManager.Instance.ToggleSelectedCard();
                    return;}
            }
       CameraManager.Instance.ChangeCurrentViewToNew(CameraView.FieldView);
            playerState = PlayerState.CardPlacement;}
        return;
    }
    if (playerState == PlayerState.Strategizing &&
GridManager.Instance.GetLastHitGridSlot() != null){
        GridSlot slot = GridManager.Instance.GetLastHitGridSlot();
        if (slot != null){
            if (slot.IsEmptySlot()) {
                CardManager.Instance.CloseMinionActionsUI();
                return;}
            if (!slot.IsEmptySlot()){

CardManager.Instance.OpenMinionUI(slot.GetSlotedMiniature(),
slot.GetMiniatureFaction());}}
        return;}
    if (playerState == PlayerState.SummonMinion){
        if (CardManager.Instance.SummonMinion()){
            playerState = PlayerState.Strategizing;
            CardManager.Instance.PrepareCardHovering();
            CameraManager.Instance.lockCamera = false;

CameraManager.Instance.ChangeCurrentViewToNew(CameraView.HandView);}
        return;}
    if (playerState == PlayerState.CardPlacement){
        if (CardManager.Instance.TryPlaceCardOnField()){
            CardManager.Instance.ActivateCard(); }
        return;
    } …
```

Kada igrač priziva novu kartu, žrtvuje karte, želi pomaknuti figuru, odabire kartu za napad ili neprijatelja u dometu, dozvoljeno je poništavanje akcije prije potvrde lijevog klika miša a zatim se poziva metoda *HandleRightClick*. Ona se može aktivirati desnim klikom miša u specifičnim stanjima i trenutno stanje pravilno vraća u početno stanje te izvršava potrebne operacije za vračanje. Ako korisnik pomiče kotačić miša u jednu ili drugu stranu kada mu je to dozvoljeno, poziva metodu *ChangeCameraViewInOrder* iz instance klase *CameraManager* kako bi sekvencijalno mijenjao trenutnu poziciju kamere. Na kraju svog poteza igrač pritisne gumb za kraj poteza i poziva *EndPlayerTurn* **metodu** koja će završiti potez igrača.

3.4.11.4. Menadžer za kameru

Klasa *CameraManager* ima zadaću upravljati pozicijom kamere u igri i pruža metode za promjenu pogleda glavne kamere. Glavna metoda klase je *ChangeCurrentViewToNew* koja kao parametar prima novi željeni pogled na koji se kamera postavlja. Klasa ima reference na transformacijske komponente pozicija kamere te ovisno o zadanom novom pogledu postavit će kameru na određenu točku. Kada igrač koristi klizač miša za pomak kamere, koristi se metoda *ChangeCurrentViewInOrder* kako bi se pogled pomicao na novi pogled koji je sljedbenik ili pratitelj trenutnog pogleda. Sve vrste pogleda određene su enumeracijom *CameraView*.

```
public class CameraManager : Singleton<CameraManager>{
    public bool lockCamera = false;
    [SerializeField] private CameraView cameraView = CameraView.HandView;
    [SerializeField] private Transform _showMainMenuWaypoint;
            …
    public void ChangeCurrentViewToNew(CameraView view){
        if (lockCamera){return;}
        switch (view){
            case CameraView.MainMenuView:

Camera.main.transform.DOMove(_showMainMenuWaypoint.transform.position,
0.5f).SetEase(Ease.InQuart);

Camera.main.transform.DORotate(_showMainMenuWaypoint.transform.rotation.eul
erAngles, 0.4f);
                cameraView = CameraView.MainMenuView;
                break;
            case CameraView.HandView:

Camera.main.transform.DOMove(_showCardsWaypoint.transform.position,
0.5f).SetEase(Ease.InQuart);

Camera.main.transform.DORotate(_showCardsWaypoint.transform.rotation.eulerA
ngles, 0.4f);
                cameraView = CameraView.HandView;
                break;
            case CameraView.FieldView:

Camera.main.transform.DOMove(_showFieldWaypoint.transform.position,
0.5f).SetEase(Ease.InQuart);

Camera.main.transform.DORotate(_showFieldWaypoint.transform.rotation.eulerA
ngles, 0.4f);
                cameraView = CameraView.FieldView;
```

```
                break;
            case CameraView.FrontArenaView:

Camera.main.transform.DOMove(_showFrontArenaWaypoint.transform.position,
0.5f).SetEase(Ease.InQuart);

Camera.main.transform.DORotate(_showFrontArenaWaypoint.transform.rotation.e
ulerAngles, 0.4f);
                cameraView = CameraView.FrontArenaView;
                break;
            case CameraView.SideArenaView:

Camera.main.transform.DOMove(_showSideArenaWaypoint.transform.position,
0.5f).SetEase(Ease.InQuart);

Camera.main.transform.DORotate(_showSideArenaWaypoint.transform.rotation.eu
lerAngles, 0.4f);
                cameraView = CameraView.SideArenaView;
                break;
            case CameraView.TopArenaView:

Camera.main.transform.DOMove(_showTopArenaWaypoint.transform.position,
0.5f).SetEase(Ease.InQuart);

Camera.main.transform.DORotate(_showTopArenaWaypoint.transform.rotation.eul
erAngles, 0.4f);
                cameraView = CameraView.TopArenaView;
                break;}
    }
    public void ChangeCurrentViewInOrder(bool toPreviousView) { }
    public CameraView GetCameraView() { }
}
[Serializable]
public enum CameraView{
    MainMenuView = 0,
    HandView = 1,
    FieldView = 2,
    FrontArenaView = 3,
    SideArenaView = 4,
    TopArenaView = 5
}
```

3.4.11.5. Menadžer za neprijatelje

Kada neprijatelji dođu na svoj potez, aktivira se klasa *EnemyManager* koja se koristi za upravljanje neprijateljskim jedinicama u igri. Služi za upravljanje ponašanjem neprijateljskih figura, njihovih akcija napada i kretanja prema igračevim figurama.

```
public class EnemyManager : Singleton<EnemyManager>
{
    public List<MinionCard> enemies = new List<MinionCard>();
    public Deck enemyDeck;
    [SerializeField] private Coroutine enemyCoroutine;

    public void StartEnemyTurn() { }
    public void TryForceStopEnemyTurn() { }
    private IEnumerator ExecuteEnemyActions() { }
    public IEnumerator MoveToClosestTarget(MinionCard minion) { }
    public void TryToAttackMinion(MinionCard minion) { }
    private MinionCard GetMinionToAttack(List<GridSlot> slots) { }
    private void RestoreMinionActions() { }
    private void ApplyEffects() { }
    private List<ModificationCard>
GetShuffledModifications(List<ModificationCard> modificationCards)/{ }
    public List<SquareNode> FindPath(SquareNode startNode, SquareNode
targetNode) { }
}
```

Instanca prati listu svih neprijateljskih karata trenutno u igri, špil u kojem se nalaze karte i referencu na korutinu koju stvara. Referenca korutine se pamti iz razloga što ako usred napadanja neprijateljske figure nestane zadnja igračeva figura, zaustaviti će njeno izvođenje jer ne postoji više figura koju neprijatelj može napasti. Kada su neprijatelji na potezu poziva se metoda *StartEnemyTurn* koja služi za sekvencijalno pozivanje ostalih metoda. Prvo se poziva metoda *RestoreMinionActions* koja će svakoj karti u svojoj listu povratiti mogućnost jednog napada, jedne akcije kretnje po ploči i za svaku kartu napada koju figura može koristiti smanjiti broj krugova do ponovnog korištenja ako je karta već korištena. Zatim se poziva *ApplyEffects* metoda koja će svakoj figuri u listi primijeniti pripojene efekte i provjeriti je li figura živa nakon primjene. Za sve figure koje je potrebno otkloniti iz igre poziva se kontroler i figura se briše. Na kraju se poziva korutina koja izvodi akcije neprijateljske figure počevši od pozivanja unutarnje korutine *MoveToClosestTarget*. Ona će prvo za svako polje ploče zapisati njegove susjede. jesu li susjedna polja prohodna i dohvatiti najbližu igračevu figuru putem *GridManager* instance klase. Zatim se provjera nalazi li se igračeva figura na jednom od susjednih polja od polja gdje se nalazi neprijateljska figura. Razlog je što nije potrebno figuru pomicati ako se već

nalazi pokraj neke igračeve figure. Ako je ipak potrebno pomicati figuru, koristi se metoda *FindPath* koja implementira A-star algoritam za pronalaženje optimalnog puta od jednog čvora do drugog. Za to se koristi klasa *SquareNode* čija referenca instance se zapisuje unutar klase *GridSlot* svakog polja ploče.

Algoritam prvo inicijalizira dvije liste čvorova od kojih je prva za pretragu a druga za obrađene čvorove. Pretraga počinje sa početnim čvorom, što je prvi čvor koji se pretražuje. Ulazi se u glavnu petlju koja će se izvršavati sve dok lista za pretraživanje nije potpuno prazna, što je ključni dio A-star algoritma. U svakoj iteraciji petlje, algoritam odabire čvor iz liste pretraživanja, čiji se odabir vrši na temelju procjene ukupne cijene do ciljnog čvora. Procjena se sastoji od dvije komponente od koji prva je g, odnosno stvarna udaljenost od trenutnog čvora do početnog čvora. Druga komponenta je h, što je heuristička procjena udaljenosti od trenutnog čvora do ciljnog čvora. Algoritam odabire čvor koji ima najmanju ukupnu procijenjenu udaljenost f što je ustvari zbroj g i h komponente a zatim taj čvor zapisuje u listu procesiranih čvorova. Za svaki čvor u iteraciji provjerava se je li on jednak ciljnog čvoru i ako jest, to znači da je algoritam pronašao optimalan put. Kada je put pronađen, algoritam rekonstruira put od ciljnog čvora za do početnog tako da se prati lanac povezanih čvorova i cijeli put se pohranjuje a zatim se koristi metoda Reverse() kako bi algoritam mogao kao rezultat vratiti put od početka do ciljnog mjesta. Ako cilj nije pronađen, algoritam će prolaziti kroz sve susjedne čvorove trenutnog čvora u iteraciji i za svaki susjedni čvor koji već nije obrađen, algoritam provjerava udaljenost od početnog čvora. Ako je nova udaljenost manja od trenutne g vrijednosti susjednog čvora ili čvor nije u listi za pretragu, ažurira se g vrijednost susjednog čvora i postavlja se referenca za lanac povezanosti za putovanja unatrag te se čvor dodaje u listu za pretraživanje.

```
public List<SquareNode> FindPath(SquareNode startNode, SquareNode
targetNode){
    List<SquareNode> toSearch = new List<SquareNode>() { startNode };
    List<SquareNode> processed = new List<SquareNode>();
    while (toSearch.Any()){
        SquareNode current = toSearch[0];
        foreach (SquareNode t in toSearch){
            if (t.GetF() < current.GetF() || t.GetF() == current.GetF() &&
t.h < current.h)
                current = t;
        }

        processed.Add(current);
        toSearch.Remove(current);
```

```
        if (current == targetNode){
            SquareNode currentPathTile = targetNode;
            List<SquareNode> path = new List<SquareNode>();
            int count = 100;

            while (currentPathTile != startNode){
                path.Add(currentPathTile);
                currentPathTile = currentPathTile.Connection;
                count--;}
            path.Reverse();
            return path;
        }
        foreach (SquareNode neighbor in current.Neighbors.Where(t =>
!processed.Contains(t))){
            bool inSearch = toSearch.Contains(neighbor);

            if (neighbor == targetNode){
                neighbor.g = (current.g + current.GetDistance(neighbor));
                neighbor.SetConnection(current);
                toSearch.Add(neighbor);
                continue;
            }
            if (!neighbor.isWalkable) {
                continue;
            }
            var costToNeighbor = current.g + current.GetDistance(neighbor);
            if (!inSearch || costToNeighbor < neighbor.g){
                neighbor.g = costToNeighbor;
                neighbor.SetConnection(current);

                if (!inSearch){
                    neighbor.h = neighbor.GetDistance(targetNode);
                    toSearch.Add(neighbor);}
            }
        }
    }
    return null;
}
```

Kada je pronađen optimalan put do igračeve figure, neprijateljska figura će se pokušati pomaknuti pomoću korutine *MiniatureMovement* iz *GridManager* instance klase do čvora koji se nalazi jedno mjesto prije ciljnog čvora, odnosno se postavlja na mjesto pokraj igračeve figure. Ako figura nema dovoljnu vrijednost dometa do ciljnog mjesta, ona će se pomaknuti za broj čvorova na putu jednak broju distance koju figura može proći ovaj potez.

Nakon pozicioniranja figure na igraćoj ploči, prelazi se na napad i poziva se metoda *TryToAttackMinion*. Ona prvo poziva metodu *GetShuffledModification* kako bi popunila listu svih karti napada pomiješanih putem Fisher-Yates algoritma. Prolazi se kroz navedenu listu i traži prva karta napada koja nije na čekanju za broj poteza. Pronalaskom prve slobodne karte uzima se njen domet na ploči i poziva se metoda *GetMinionAttack* koja će proći kroz listu svih polja u dometu i nasumično odabrati jednu neprijateljsku figuru za napada, ako takva postoji. Svi nužni podaci postavljaju se u strukturu svojstva kontrolera i prosljeđuju se njegovoj metodi za početak duela. Kada su sve figure iskoristile akciju napada i pomaka, poziva se *EndTurn* metoda putem *GameManager* instance klase i završava se potez. U klasi još postoji metoda *TryForceStopEnemyTurn* odgovorna za zaustavljanje korutine izvan klase u slučaju igračeve pobjede, odnosno uništenjem svih neprijateljskih figura.

3.4.11.6. Menadžer za resurse

Klasa *ResourceManager* ima svrhu upravljanja i pristupanja različitim resursi. Ovdje se koristi za pristup svim skriptabilnim objektima unutar projekta koji se nalaze unutar *Resource* mape u projektu. Za objekte u navedenoj mapi koristi se dinamičko učitavanje tijekom izvođenja igre i podaci ne moraju biti unaprijed učitani u memoriju. *Unity* tada nu**d**i API za lak pristup tim objektima putem naziva puta do objekta ili imena objekta.

```
public class ResourceManager : Singleton<ResourceManager>{
    [SerializeField] private List<ScriptableMinion> minions;
    [SerializeField] private List<ScriptableEnemyEncounter>
enemyEncounters;
    [SerializeField] private List<ScriptableModificationCard>
modifications;
    private Dictionary<MinionName, ScriptableMinion> _minionDictionary;
    private Dictionary<EncounterType, ScriptableEnemyEncounter>
_enemyEncounterDictionary;
    private Dictionary<ModificationName, ScriptableModificationCard>
_modificationDictionary;
    protected override void Awake(){
        base.Awake();
        AssembleResources();}
    private void AssembleResources(){
        minions = Resources.LoadAll<ScriptableMinion>("Minions").ToList();
        enemyEncounters =
Resources.LoadAll<ScriptableEnemyEncounter>("Enemy Encounters").ToList();
        modifications =
Resources.LoadAll<ScriptableModificationCard>("Modifications").ToList();
        _enemyEncounterDictionary = enemyEncounters.ToDictionary(r =>
r.encounterType, r => r);
        _minionDictionary = minions.ToDictionary(r => r.minionName, r =>
r);
        _modificationDictionary = modifications.ToDictionary(r =>
r.modificationName, r => r);
    }
```

Kada se stvara instanca klase poziva se metoda *Awake* koja poziva nadjačava metodu klase *Singleton* i poziva metodu *AssembleResources*. Nakon poziva u zadane liste učitat će se svi potrebni skriptabilni objekti putem naziva mapa. Objekti su postavljeni u rječnike kao vrijednosti za ključem enumeracije pripadajućeg naziva figure za određeni objekt. Preostale metode *GetMinion*, *GetEnemyEncounter* i *GetModificatioCard* koriste se kako potrebni objekt dohvatio izvan klase.

3.4.11.7. Menadžer za igru

Klasa *GameManager* upravlja najvišom razinom toka igre te zajedničkim elementima između *PlayerManager* i *EnemyManager* instanca klase. Kada se stvori instanca klase poziva se metoda *Start* koja postavlja početno unutarnje stanje menadžera. Zatim **se** dohvaća **nužni** skriptabilni objekt koji definira koje će se neprijateljske figure kreirati i postavlja ikonu kursora miša na zadanu 2D teksturu. Nadalje prati svoje unutarnje stanje i mijenja svoje stanje putem metode *ChangeState* te ovisno o stanju dopušta izvršavanje određenih operacija.

```
public class GameManager : Singleton<GameManager>{
    public GameState gameState;
    [SerializeField] private bool _isFirstRound;
    [SerializeField] private TextMeshProUGUI _helpText;
    [SerializeField] private ScriptableEnemyEncounter _encounter;
    [SerializeField] private GameObject _turnEndUI;
           …
    [SerializeField] private Slider _clockSlider;
    [SerializeField] private TextMeshProUGUI _minionPopupUI;
    [SerializeField] private MinionStatsController _minionStatsController;
    [SerializeField] private Sprite cursorTexture;
    [SerializeField] private SpawnController _spawnController;
    void Start() { }
    public void ChangeState(GameState newState) { }
    public void EndTurn() { }
    private void HandleEncounterPreparation() { }
    public void EnterDemo() { }
    public void ExitGame() { }
    public void ReturnToMainMenu() { }
    public void ToggleMinionPopup(string infoText, Miniature miniature) { }
    public void ToggleTurnClock(bool toggleOn) { }
    public void ToggleHelpText() { }
    public void SetHelpText(string text) { }
    public void SetAlertText(string text) { }
    IEnumerator ShowAlertCoroutine(string text) { }
    IEnumerator SwitchToGameplay() { }
    public void ShowMinionCurrentHealthUI(MinionStats stats, int
miniatureMovement, int maxHp) { }
    public void CloseMinionCurrentHealthUI() { }
    public Slider GetTurnClock() { }
    public SpawnController GetSpawnController() { }}
```

Prvo stanje je *MainMenu* koje će kameru dovesti na određenu točku koordinatnog sustava za početni izbornik te uključiti pripadajuće elemente korisničkog sučelja. Sljedeće stanje je *EncounterPreparation* koje će postaviti igračev špil karata i pozvati *GridManager* instancu klase da postavi igraću ploču. Slijede stanja *PlayerTurn* i *EnemyTurn* koja će pozivati glavne metode za početak poteza timova putem odgovornih menadžera. Za početni potez neprijatelja umjesto njihovog klasičnog poteza, postaviti će se neprijateljske figure na ploču. Zadnje stanje je *EndBattle* koje će aktivirati elemente korisničkog sučelja za označavanje kraja igre. Krajem poteza tima poziva se metoda *EndTurn* koja naizmjenično postavlja stanje *PlayerTurn* i stanje *EnemyTurn*. Ostale metode koriste se za paljenje i postavljanje elemenata korisničkog sučelja.

```
public void ChangeState(GameState newState){
        gameState = newState;
        switch (newState){
            case GameState.MainMenu:
                CameraManager.Instance.lockCamera = false;
CameraManager.Instance.ChangeCurrentViewToNew(CameraView.MainMenuView);
                _mainMenuUI.SetActive(true);
                break;
            case GameState.EncounterPreparation:
                ToggleTurnClock(false);
                HandleEncounterPreparation();
                break;
            case GameState.PlayerTurn:
                PlayerManager.Instance.StartPlayerTurn(_isFirstRound);
                break;
            case GameState.EnemyTurn:
                ToggleTurnClock(false);
  CameraManager.Instance.ChangeCurrentViewToNew(CameraView.FrontArenaView);
                PlayerManager.Instance.playerState = PlayerState.Waiting;
                if (_isFirstRound){
                    _spawnController.SpawnEnemies(_encounter);
                    _isFirstRound = false;
                    EndTurn();}
                else{
                    EnemyManager.Instance.StartEnemyTurn();}
                break;
            case GameState.EndBattle:
                _creditsUI.SetActive(true);
                break;}}
```

4. Zaključak

Razvoj igara često nosi sa sobom niz izazova koje čine cijelu industriju vrlo zahtjevnom, pogotovo za one koji projekte rade samostalno. Razvijanje igre zahtjeva visoko znanje razumijevanja raznih disciplina kao što su glazba, animacije, umjetnički dizajn, programiranje i slično. Također je potrebno uložiti veliku količinu vremena i truda kako bi se stvorio kvalitetan produkt. No, iako je to težak i zahtjevan posao, nagrada dolazi u obliku zadovoljstva i kreativne ispunjenosti kada se vidi završni produkt i igrači uživaju u kreiranoj igri.

Razvoj strateške igre uloga u okruženju *Unity* i *Blender* alata pružio je duboki uvid u izazove i zadovoljstva kompletnog razvoja igre. Tijekom ovog putovanja naučio sam mnogo o programiranju, modeliranju i dizajnu igre. Za razliku od rada u timu, samostalan razvoj omogućio mi je potpunu kreativnu kontrolu nad igrom, ali i odgovornosti i izazove svih aspekata stvaranja igre. Neke ideje koje su nazirale isprva su se činile nemoguće za implementaciju, ali upornost i zalaganje pokazalo je kako ništa nije nemoguće ako se provede dovoljno vremena na učenje i razvijanje solucije problema.

Uspješnim završetkom projekta dokazano je kako su danas besplatno dostupne sve potrebne tehnologije i dovoljno je znanja da svaki individualac može krenuti u svijet razvoja igara te uz dovoljno truda postići kvalitetne rezultate. Nije potrebno kupovati gomile raznih visoko kvalitetnih paketa modela, pomoćnih programa i kodova već samostalnim radom i prelaženjem preko neuspjeha moguće je realizirati bilo kakvu ideju. Izrazito sam zadovoljan rezultatom rada i planiram razvijati ovaj projekt i dalje, te potencijalno jednog dana izraditi i objaviti kompletnu igru na web.

Naposljetku, zaključujem kako danas postoji puno znanja i alata za developere u svijetu kreiranja video igara. Današnja dostupnost besplatnih alata i resursa, uz upornost i predanost, omogućuje ostvarenje raznih ideja bez potreba za kupnjom velikog broja materijala. Moje iskustvo na ovom projektu potvrđuje da je moguće postići visokokvalitetne rezultate uz pomoć dostupnih resursa i vlastite kreativnosti. Ovaj projekt nije samo dokaz mojeg truda i rada, već i inspiracija za buduće poduhvate u razvijanju igra. Nastavljam s razvojem ovog projekta i gledam prema budućnosti u nadi da ću jednog dana ponosno objaviti svoju vlastitu igru.

Popis literature

[1] A Brief History of Role Playing Games - Ludogogy. Pristupano 2. Rujan, 2023. https://ludogogy.co.uk/a-brief-history-of-role-playing-games/

[2] What is a Role-Playing Game (RPG)? - Definition from Techopedia. Accessed 2. Rujan, 2023. https://www.techopedia.com/definition/27052/role-playing-game-rpg

[3] Origins of the Strategy RPG (1982-1995). Pristupano 2. rujan, 2023. https://www.superjumpmagazine.com/origins-of-the-strategy-rpg/

[4] What Is Unity? - A Top Game Engine Solution For Video Games - GameDev Academy. Pristupano 4. Rujan, 2023. https://gamedevacademy.org/what-is-unity/

[5] About — blender.org. Pristupano 4. Rujan, 2021. https://www.blender.org/about/

[6] DOTween - Get Started. Pristupano 4. Rujan, 2023. https://dotween.demigiant.com/getstarted.php

[7] How to Use Leonardo AI to Create AI Art. Pristupano 5. Rujan, 2023. https://www.makeuseof.com/how-to-use-leonardo-ai-create-art/

[8] What is UV Mapping & Unwrapping? Pristupano 5. Rujan, 2023. https://conceptartempire.com/uv-mapping-unwrapping/

[9] Unity: Understanding URP, HDRP, and Built-In Render Pipeline. Accessed Pristupano 5. Rujan, 2023. https://www.occasoftware.com/blog/unity-understanding-urp-hdrp-built-in

]10] Tip of the Day: Manager Classes & Singleton Pattern in Unity | by Mohamed Hijazi | Level Up Coding. Pristupano 8. Rujan, 2023. https://levelup.gitconnected.com/tip-of-the-day-manager-classes-singleton-pattern-in-unity-1bf3aafe9430

[11] Singletons In Unity - How To Implement Them The Right Way. Pristupano 8. Rujan, 2023. https://awesometuts.com/blog/singletons-unity/

[12] What is MonoBehaviour? - codinBlack. Pristupano 9. Rujan, 2023. https://www.codinblack.com/what-is-monobehaviour/

[13] Scriptable Objects in Unity (how and when to use them) - Game Dev Beginner. Pristupano 9. Rujan, 2023. https://gamedevbeginner.com/scriptable-objects-in-unity/

[14] What is “SerializeField” in Unity? Pristupano 10. Rujan, 2023. https://www.educative.io/answers/what-is-serializefield-in-unity

Popis slika

Slika 1. Unity sučelje 2
Slika 2. Blender sučelje 3
Slika 3. DoTween funkcija 5
Slika 4. Leonardo AI sučelje 6
Slika 5. Primjer izrade 3D modela 7
Slika 6. Primjer UV raspakiravanja 3D modela 8
Slika 7. Sučelje za izradu projekta 9
Slika 8. Početni izbornik igre 10
Slika 9. Prikaz igračevog pripravnog poteza 11
Slika 10. Prikaz igračevog normalnog poteza 11
Slika 11. Žrtvovanje figura 12
Slika 12. Povezivanje karte napada s kartom figure 12
Slika 13. Odabir figure na ploči 13
Slika 14. Odabir karte za napad 14
Slika 15. Duel figura 14
Slika 16. Raspored klasa u projektu 15
Slika 17. ScriptableMinion skriptabilni objekt 23
Slika 18. MinionCard komponenta 25
Slika 19. Primjer ScriptableModificationCard objekta 26
Slika 20. Prikaz figure 30
Slika 21. Prikaz podatka špila 32
Slika 22. Označavanje polja za karte 33
Slika 23. Polje igrače ploče 36
Slika 24. Izgled ekrana za odabir napada 39
Slika 25. Označavanje polja odabirom napada 39