Izrada 2D računalne igre temeljene na misijama praćenja u Unity-ju

Katoliković, Ivan

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Ivan Katoliković

IZRADA 2D RAČUNALNE IGRE TEMELJENE NA MISIJAMA PRAĆENJA U UNITY-U

DIPLOMSKI RAD

Varaždin, 2023.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Ivan Katoliković

Matični broj: 48047/15–R

Studij: Informacijsko i programsko inženjerstvo

IZRADA 2D RAČUNALNE IGRE TEMELJENE NA MISIJAMA PRAĆENJA U UNITY-U

DIPLOMSKI RAD

Mentor:

Doc. dr. sc. Mario Konecki

Varaždin, travanj 2023.

Ivan Katoliković

Izjava o izvornosti

Izjavljujem da je moj završni/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 potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

Opseg diplomskog rada je kreiranje igre temeljene na misijama praćenja. Praktične je prirode i uključuje razvoj kompleksnih međusobno povezanih sustava korištenjem raznih alata koje pruža Unity. Uključuje kreiranje mrežnih modela kroz kôd, kreiranje jednostavnih umjetnih inteligencija kako bi računalo igralo protiv igrača i korištenja ScriptableObject-a za stvaranje naprednih funkcionalnosti i modularnosti, načina korištenja raznih atributa kako bi se prilagodio inspektor za prilagodio za napredne mogućnosti stvaranja resursa. Dotiče se aksijalnih koordinata za mrežu heksagona i načina kako se njima služiti za navigaciju po mreži. Ukratko će biti objašnjeno koje su glavne karakteristike i razlike između Unity klasa i njihovih kuka, te načini očuvanja podataka između scena.

Ključne riječi: Unity; razvoj; misije praćenja; C#; csharp; video igra; roguelike;

Sadržaj

1. Uvod

Tijekom studiranja koristio sam Unity kao rješenje za izrade projekta, ali samo jedan od tih projekata je bio video igra. Imao sam prilike raditi na dijelu male video igre s kolegama, no nisam ulazio u dubinu, već samo površinski. Od tuda dolazi moja motivacija za ovom temom. Unity mi je daleko najdraži alat koji koristim (osim možda vječitog WinRar-a), ali ga ne koristim za njegovu primarnu svrhu – izradu video igara.

Misije praćenja su jedan od najkontroverznijih pojmova u svijetu video igara. U gotovo svim slučajevima izazivaju negativne reakcije. Premisa najčešće ide nekako ovako:

1. Prihvati misiju otpratiti pratilaca od točke A do točke B.
2. Na putu će doći do obračuna gdje se igrač mora pobrinuti o dobrobiti pratilaca.
3. Igrač i pratilac uspješno stižu do cilja i time misija završava.

U praksi to često izgleda ovako:

1. Prihvati misiju otpratiti pratilaca od točke A do točke B.
2. Pratilac će hodati ili prebrzo ili presporo i pričati nekakve besmislene monologe.
3. Kada dođe do sukoba, neće oklijevati izgubiti svoj život, štoviše u tome će se truditi.
4. Kada konačno igrač i pratilac budu blizu točke B, pratilac će zapeti za nekakav kamenčić i tu staje misija bez da ju je moguće završiti.

Na temelju tih iskustava, htjedoh napraviti igru koja se temelji baš na ovim negativnim stavkama.

2. Opis korištenih alata

Korišteni alati i verzije:

1. Unity – 2021.3.22f1
2. Visual Studio Code – 1.77.1
3. GIMP – 2.10.30
4. Leonardo.ai

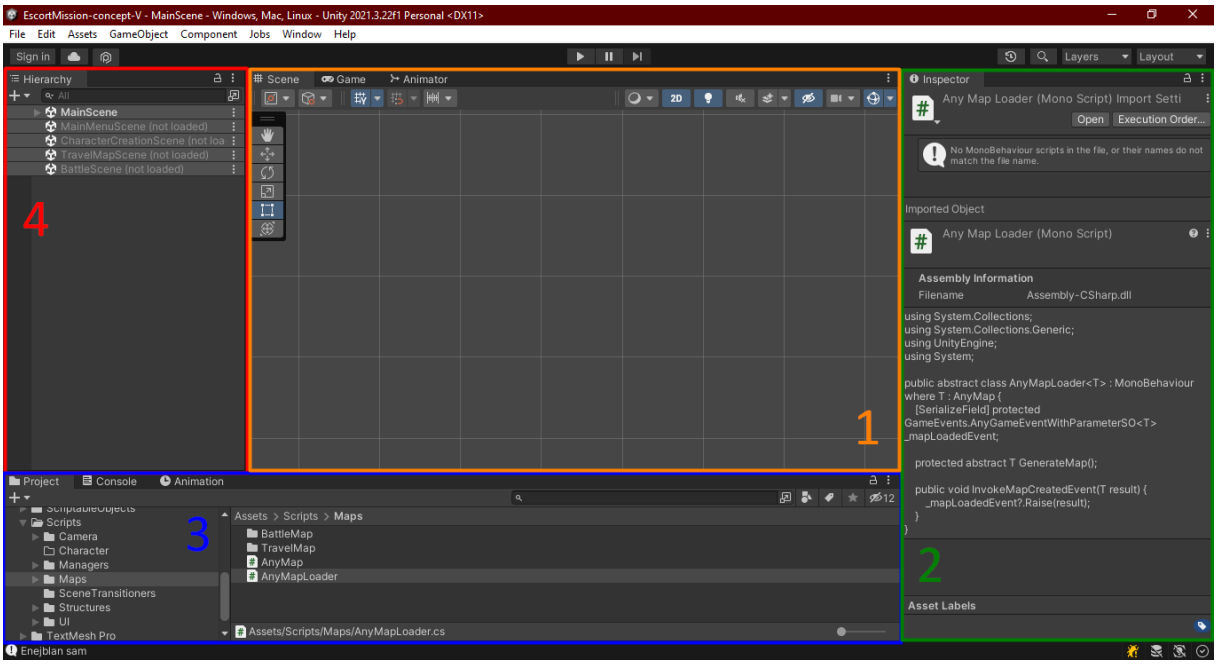
Za izradu video igre, korištena je razvojna platforma Unity. Uz Unity se klasično koristi Microsoft Visual Studio razvojno okruženje za pisanje koda skripti, no ono nije korišteno za izradu ovog rada zbog neočekivanog problema u radu s Unity-em. Umjesto njega, korišten je Microsoft Visual Studio Code. Za izradu i uređivanje grafičkih resursa igre, korišteni su GIMP i Leonardo.ai.

2.1. Razvojna platforma Unity

Unity je snažan alat za razvoj i, u trenutku pisanja, ima besplatnu verziju koja je dostupna za sve osobe i organizacije koji imaju prihode ili fondove prikupljene u posljednjih dvanaest mjeseci manje od 100 tisuća američkih dolara. [1]

Temeljen je na C++ jeziku, a za kodiranje programeri primarno koriste C#, iako je moguće koristiti i JavaScript (UnityScript) ili Boo. Pisani kôd se izvodi na Mono-u (Microsoft .NET Framework). Radi se o 2D/3D pogonskom sustavu koji pruža mogućnost za razvoj sustava i/ili aplikacijskih scena za 2D, 2.5D i 3D. Primarno se koristi za izradu video igara, ali je veoma koristan za izradu bilo čega čemu je potreban rad u prostoru. Također dolazi s ogromnom trgovinom resursima gdje se može pronaći mnoštvo tekstura, zvukova ili alata. [2]

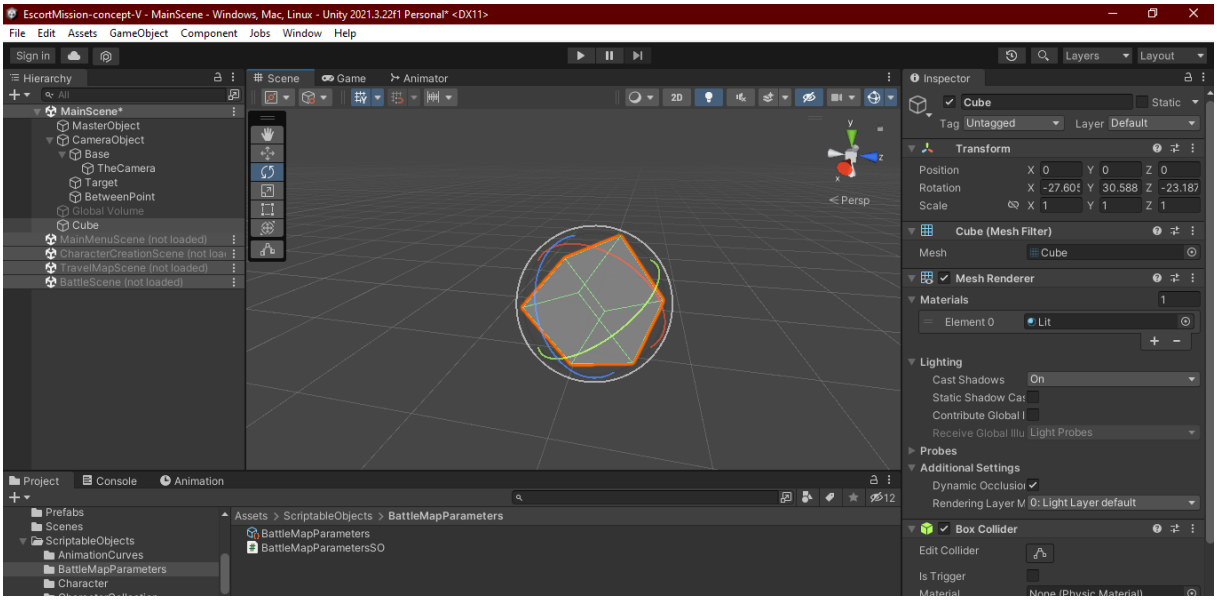
Unity uređivač ima nekoliko glavnih prozora koji se mogu organizirati po želji korisnika. Ovo je raspored koji osobno koristim (Slika 1).



Slika 1. Unity uređivač

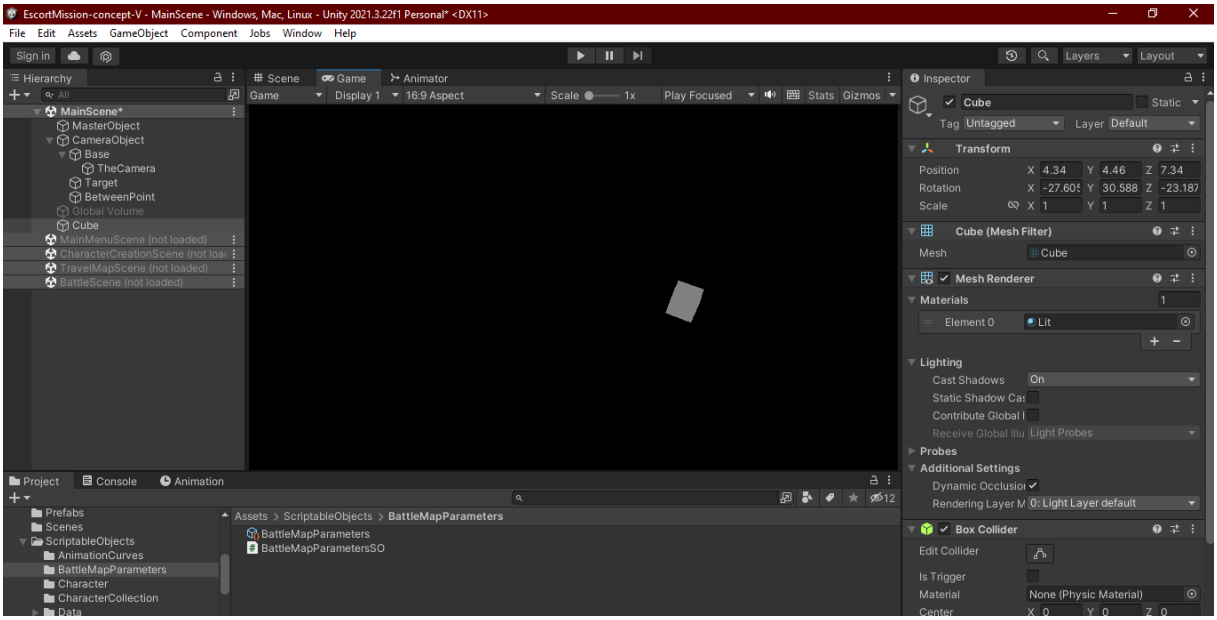
2.1.1. Scena i igra prozori

Prozor scene služi za uređivanje rasporeda objekata što u prostoru, a što za korisničko sučelje. Na žalost u radu nemam nekih „vidljivih“ objekata igre jer se uglavnom objekti kreiraju tek kada se igra pokrene. Za primjer (Slika 2) sam u scenu dodao novi 3D objekt igre (kocku) koji sam malo rotirao kako bi se lakše vidjela.



Slika 2. Primjer kocke

Prozor igre prikazuje kako scena izgleda iz perspektive kamere koja je uključena (ako takva postoji) i ovdje možemo vidjeti kako tak kocka izgleda (Slika 3) kroz oči kamere.

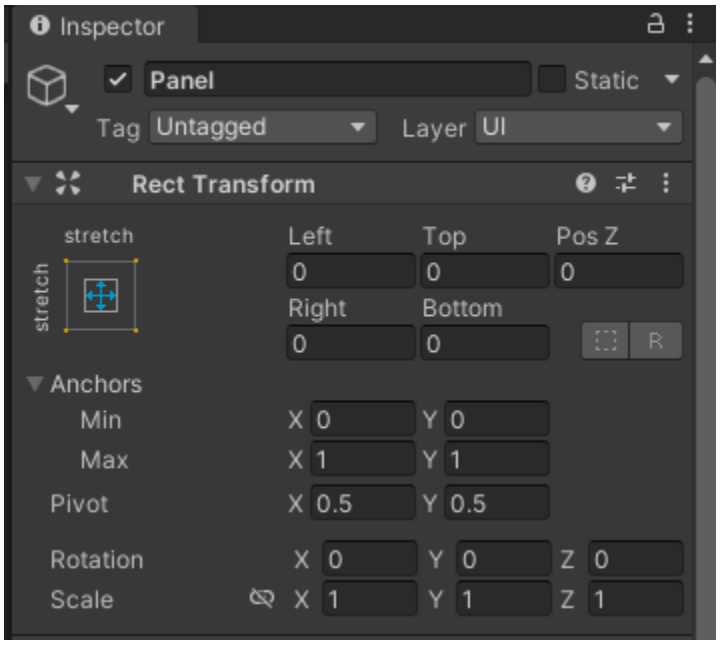


Slika 3. Kocka kroz kameru

Zato što koristim 2D univerzalni cjevovod za renderiranje, kocka nije osvijetljena kako treba, pa ne vidimo njene karakteristike (izgleda kao kvadrat).

2.1.2. Inspektor

Inspektor služi za uređivanje karakteristika objekata igre. Objekt igre sam po sebi nema nikakve karakteristike. Karakteristike i funkcionalnosti mu daju komponente koje se na njega dodaju. Komponente mogu biti skripte, rendereri, mrežni modeli, kamere itd. Jedina komponenta koju svaki objekt igre mora imati je komponenta Transform, to jest komponenta transformacije koja definira objektovu poziciju u svijetu (ili u odnosu na roditelja ako ga ima), rotaciju i mjerilo. Objekti igre koji služe za korisničko sučelje, umjesto Transform komponente, imaju RectTransform komponentu (Slika 4) koja se može gledati kao 2D Transform komponenta i ima neke dodatne mogućnosti kao što su sidrenje i zakretanje.

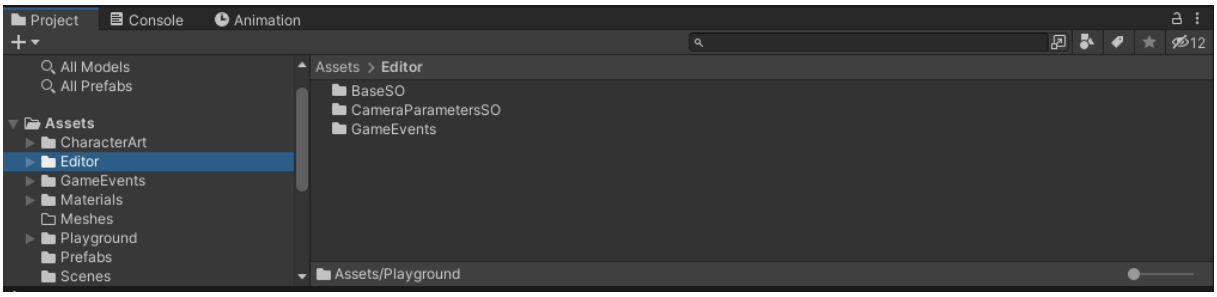


Slika 4. RectTransform komponenta

Postoje mnoge komponente kao što su kamera, sustavi događaja, sudarači raznih oblika, mrežni modeli, platna, bacači zraka itd. Također, možda najbitnija vrsta komponenti su skripte koje proširuju MonoBehaviour klasu. Detaljnije o tome kasnije, ali bitno je napomenut da ako skripta ne proširuje MonoBehaviour klasu, nije ju moguće „zakvačiti“ na objekt igre. Skripte koje ne proširuju MonoBehaviour klasu i dalje mogu biti referencirane u MonoBehaviour klasama – ne mora svaka skripta proširivati MonoBehaviour klasu. To je pogotovo evidentno jer se MonoBehaviour klase ne mogu instancirati pomoću „new“ ključne riječi koju koristi C#.

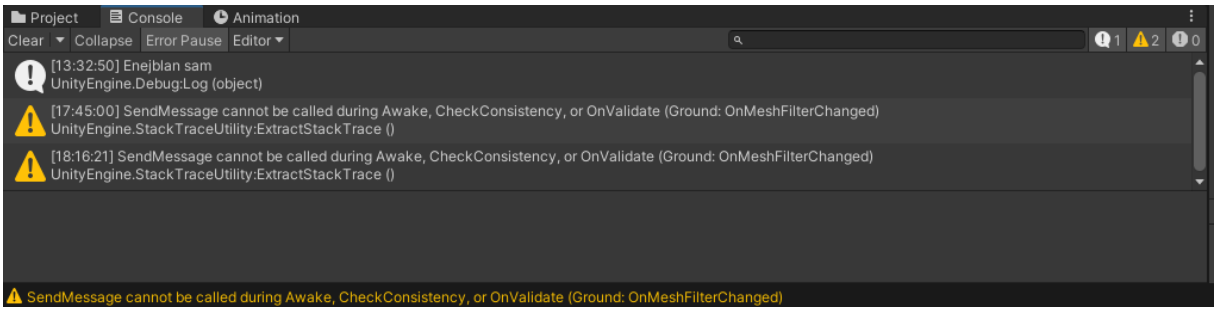
2.1.3. Prozor projekta i konzole

U prozoru projekta se može vidjeti i uređivati struktura datoteka, ali i kreirati skripte, skriptne objekte, resurse itd. U lijevom dijelu prozora se može vidjeti struktura kao stablo, a u desnom kao mapa (Slika 5).



Slika 5. Prozor projekta/konzole

Konzola služi za ispis upozorenja, grešaka i zapisa. Neophodan je alat za razvoj (pogotovo ako imate poteškoća s programom za ispravljanje pogrešaka (eng. *debugger*)). Ispod (Slika 6) možemo vidjeti (odozgo prema dolje) zapis, koji je ispisala jedna skripta koja služi kao repozitorij nekih podataka i mora uvijek biti omogućena za ispravan rad, i dva upozorenja.



Slika 6. Prozor konzole

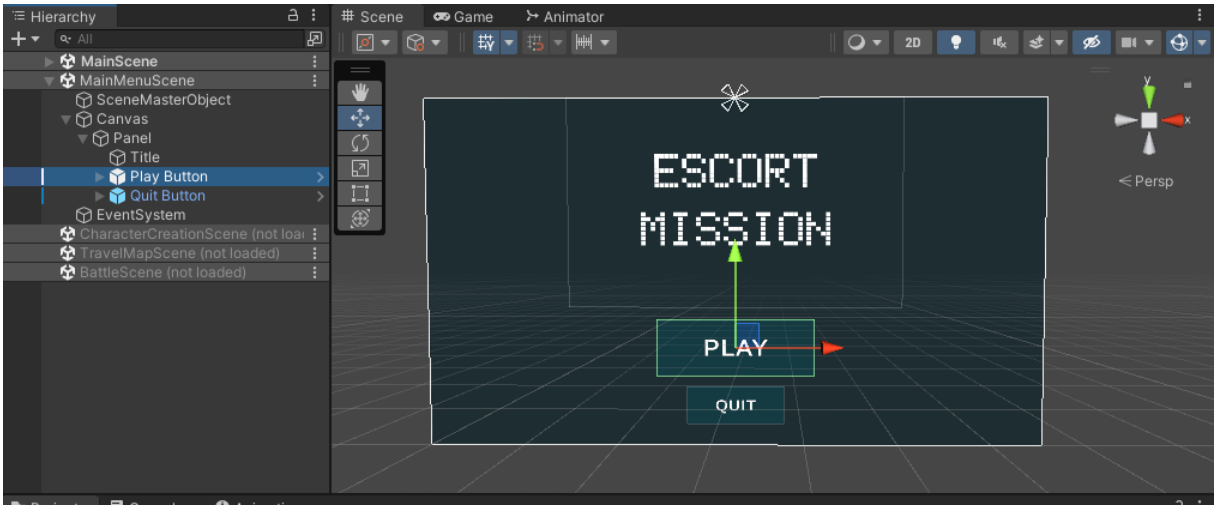
2.1.4. Prozor hijerarhije

Prozor hijerarhije služi za kreiranje scena (mogu se kreirati i u prozoru projekta) i objekata igre. Na primjeru (Slika 7) promatramo scenu glavnog izbornika. Nije važno kako se objekti igre zovu jer njihov naziv ne određuje kako će se oni ponašati. Njihovo ponašanje i sposobnosti određuju komponente koje imaju. Objekti igre koje ova scena ima:

1. SceneMasterObject
    - Objekt pomoću svojih komponenti upravlja scenom. U ovom slučaju jednostavne scene, samo tranziciju na slijedeću scenu.
2. Canvas

- Platno služi za korisničko sučelje i njegove elemente. Ono ima jedan objekt igre kao svoje dijete – Panel. Taj objekt, služi samo kao pozadina. Ono što mu daje konkretnije značenje su njegova djeca:
    1. Title – ono ima „TextMeshPro – Text (UI)“ komponentu koja služi za renderiranje mrežnog modela teksta naslova igre.
    2. Gumb Play – kada se klikne, započinje igra. U principu, pri kliku zove metodu glavnog objekta igre koji onda prelazi na slijedeću scenu. Na sebi ima komponentu Button koja poziva metodu jedne od skripata na glavnom objektu scene.
    3. Gumb Quit – kada se klikne, igra se gasi. Zove metodu skripte koju ju ima kao komponentu i ona gasi igru.

3. EventSystem
    - Ako se u sceni žele koristi događaji (npr. klik na gumb ili dozivanja kroz skripte), a često su korišteni u ovom radu, potrebno je imati sustav događaja kako bi događaji radili. Također, EventSystem objekt sam po sebi nema nikakve mogućnosti (osim komponente Transform), već mu funkcionalnosti daju komponente.

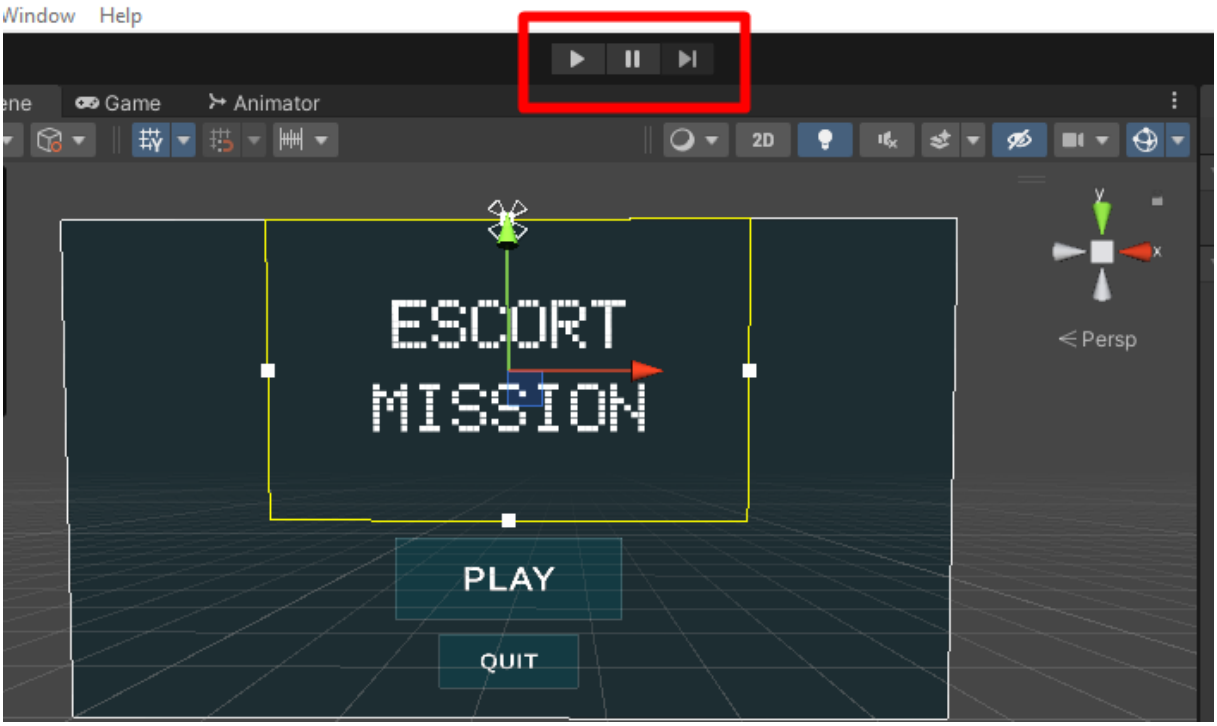


Slika 7.Prozor hijerarhije scena

2.1.5. Pokretanje igre u uređivaču

U uređivaču, izvođenjem igre se kontrolira trima označenim gumbima (Slika 8). Prvi gumb pokreće igru, a kada je pokrenuta, ponovnim klikom zaustavlja igru. Drugi gumb pauzira

izvođenje, a radi i u pokrenutom načinu i u zaustavljenom načinu. Treći gumb radi samo kada je igra pokrenuta i napreduje igru za jedan kadar (eng. *frame*).



Slika 8. Gumbi kontrole izvođenja

2.1.6.MonoBehaviour i ScriptableObject

MonoBehaviour i ScriptableObject su klase izvedene iz klase Object i predstavljaju važne klase za rad u Unity-u.

2.1.6.1. MonoBehaviour

Kada se bilo koja klasa kreira kroz Unity uređivač, ona automatski proširuje MonoBehaviour klasu. Naravno, ako to ne želimo skriptu možemo urediti tako da ju ne proširuje ili da proširuje neku drugu klasu.

Ako skriptu želimo zakvačiti na objekt igre, onda ta skripta mora proširivati MonoBehaviour. MonoBehaviour pruža radni okvir (eng. *framework*) koji omogućuje kvačenje na objekte igre i kuke (eng. *hooks*) na korisne događaje. Također omogućuje upravljanje korutinama (eng. *corutines*) koje omogućuju pisanje asinkronog kôda. [3]

2.1.6.2. Korisni događaji MonoBehaviour-a

Kada kreiramo skriptu kroz Unity uređivač, uz to što klasa proširuje MonoBehaviour, automatski su kreirane i dvije funkcije – Start i Update. Osim njih postoje i druge funkcije koje se mogu ručno dodati.

Najbitniji, najkorišteniji događaji poredani po životnom ciklusu klase:

1. Awake
    - Funkcija Awake se poziva kada je instanca skripte učitana, to jest kada je objekt igre inicijaliziran kada se učita scena ili kada je, do sada, neaktivan objekt igre aktiviran ili kreiran. [4]
    - Primarni način korištenja je za inicijalizaciju varijabli i stanja prije nego je aplikacija pokrenuta. [4]
    - Može se dogoditi samo jednom u životnom ciklusu skripte. [4]
    - Unity poziva Awake metodu na objektima igre na nedeterministički način, stoga može doći do problema ako se u ovoj metodi pokušaju referencirati objekti koji se instanciraju u Awake metodi drugih objekata igre. [4]
2. OnEnableoo
    - Svaki objekt igre i svaka komponenta mogu biti omogućene (eng. *enabled*) i onemogućene (eng. *disabled*).
    - Poziva se svaki put kada je objekt omogućen [5] i neposredno nakon Awake metode. [6]
    - Veoma čest način korištenja ove metode je pretplaćivanje na događaje.
3. Start
    - Poziva se u kadru kada je skripta prvi puta omogućena i to prije bilo koje Update metode. [7]
    - U ovoj metodi je moguće referencirati varijable drugih komponenti objekata igre koje su instancirane u Awake metodi. [7]
4. FixedUpdate
    - Poziva se u svakom fiksnom kadru fiksne stope i ima frekvenciju sustava fizike. [8]
    - Služi za rad s fizikom, točnije omogućuje ispravne kalkulacije fizike neovisno o broju kadrova po sekundi. Kada bi se kalkulacije fizike

izvodile kalkulirale u Update metodi, ovisno o broju kadrova po sekundi, došlo bi do devijacija radi kojih fizika ne bi radila kako treba. [8]

5. Update
    - Poziva se svaki kadar. [9]
    - Vrlo čest način korištenja je, barem do nedavno, za upravljanje korisničkim unosom podataka. Na primjer, svaki kadar skripta bi gledala je li korisnik stisnuo određeni botun i prema tome dalje izvršavala rad. Unity ima novi sustav za korisnički unos – Input System gdje se skripte pretplaćuju na njegove događaje i postupaju u skladu s time. Time se ne mora svaki kadar provjeravati je li uvjet unosa ispunjen. [10]
6. LateUpdate
    - Poziva se nakon svih Update funkcija. [11]
    - Radi se o korisnoj funkciji za sve akcije koje se moraju dogoditi nakon što je nešto odrađeno u Update ostalim Update funkcijama (ili FixedUpdate) funkcijama. Na primjer, ako kamera mora pratiti objekt koji se pomiče u nekoj od Update funkcija, praćenje tog objekta (njegove pozicije) je dobro implementirati u LateUpdate metodi jer će onda pratiti njegovu ažuriranu poziciju. U suprotnom bi moglo doći do neuglađenog praćenja, to jest kretnji kamere. [11]
7. OnDisable
    - Funkcija se poziva kada je objekt onemogućen što uključuje i kada je objekt uništen. Također, poziva se i kada je završeno ažuriranje skripti nakon što je kompilacija završena. U tom slučaju se potom poziva OnEnable funkcija nakon što je skripta učitana. [12]
    - Dobar primjer korištenja je poništavanje pretplate na događaje i čišćenje koda.
8. OnDestroy
    - Funkcija se poziva kada scena ili igra završi, to jest kada je objekt uništen. [13] Iako dokumentacija ne specificira, to uključuje i pozivom funkcije Destroy(objekt koji se želi uništiti).
    - Neće biti pozvana ako objekt prije nije bio aktivan. [13]

- Napomena je da nije pouzdano ni preporučeno koristiti ovu funkciju za aplikacije na mobilnim platformama. [13]

9. OnValidate
    - Ova funkcija će biti pozvana samo u uređivaču i to kada su svojstva izmijenjena, uključujući kada je objekt deserijaliziran. [6]

2.1.6.3. ScriptableObject

Skriptabilni objekti se koriste kada želimo stvoriti objekte koji žive izvan objekata igre. Klasa skripte mora proširiti ScriptableObject klasu kako bi mogla biti skriptabilni objekt. Nadalje, postoji nekoliko načina za stvaranje instanci [14]:

1. Korištenjem funkcije CreateInstance
2. Korištenjem atributa CreateAssetMenuAttribute
3. Zvanjem AssetDatabase.CreateAsset
4. Pomoću ScriptImporter-a

Za razliku od MonoBehaviour-a ScriptableObject-i ne postoje unutar Unity-evog životnog ciklusa objekta, ali i dalje imaju funkcije koje se pozivaju kod određenih događaja. Glavne razlike kod tih funkcija su to da ne postoje funkcije Start, Update, FixedUpdate i LateUpdate jer su one karakteristične za Unity-ev životni ciklus objekta.

OnEnable se poziva kada je ScriptableObject klasa učitana u memoriju. OnDisable se poziva kada ScriptableObject izađe iz okvira, Awake se poziva kada je ScriptableObject pokrenut i OnDestroy kada će ScriptableObject biti uništen. OnValidate se poziva kada je skripta učitana ili kada je vrijednost svojstva izmijenjena. [14]

ScriptableObject je jako koristan alat za stvaranje događaja, pohranu podataka i uštedivanja memorije. Bitno je napomenuti kako se ne može direktno koristiti za sustav pohrane i učitavanja.

2.2. Visual Studio Code

Visual Studio Code je lagan i moćan uređivač izvornog koda i dostupan je za Windows, macOS i Linux operacijske sustave. Dolazi s ugrađenom podrškom za JavaScript, TypeScript i Node.js. Također, ima bogat ekosustav proširenja za druge jezike i okruženja. [15] Jedno od tih okruženja je C#, u kojem sam pisao kôd rada. Koristio sam i ekstenziju *„Debugger for Unity“*, to jest programski ispravljač pogrešaka za Unity koji je dostupan na linku:

https://marketplace.visualstudio.com/items?itemName=deitry.unity-debug, ali je diskontinuiran i evidentno postoji problemi u njegovom radu kada je u pitanju novija verzija Unity-a poput ove korištene u radu. Unatoč tome, pronašao sam nekoliko mitigacijskih metoda kojima je proces pronalaženja grešaka bio moguć, ali ih neću opisivati jer vrlo vjerojatno više neće raditi u skorije vrijeme.

Ukratko, preporučujem korištenje Microsoft Visual Studio alata. Ranije spomenuti problem na koji sam naišao u radu s Unity-em, zbog kojega uopće koristim Visual Studio Code, je davno ispravljen.

2.3. GIMP

GIMP je akronim za *„GNU Image Manipulation Program“* što bi na hrvatskom značilo GNU program procesiranja slika. Radi se o besplatno distribuiranom programu za zadatke tipa retuširanje fotografija, kompoziciranja i autorizaciju slika. Može se koristiti kao jednostavan program za crtanje, ali i kao stručan program za retuširanje fotografija, sustav za mrežnu serijsku obradu, renderer slika za masovnu proizvodnju, itd. [16]

GIMP sam koristio za crtanje mapa i uređivanje slika. Unity sam po sebi nema mogućnost uređivanja slika, te je nekakav alata ovakve prirode veoma bitan dio arsenala kada je u pitanju razvoj video igara.

2.4. Leonardo.ai

Leonardo.ai je alat koji koristi umjetnu inteligenciju za stvaranje slika. Njime se mogu kreirati sredstva igre, to jest slike oklopa, zgrada, okoliša itd. [17] Trenutno je u ranom pristupu i još nije javno dostupan.

Za rani pristup se može prijaviti na poveznici: https://leonardo.ai/?ref=FutureTools.io.

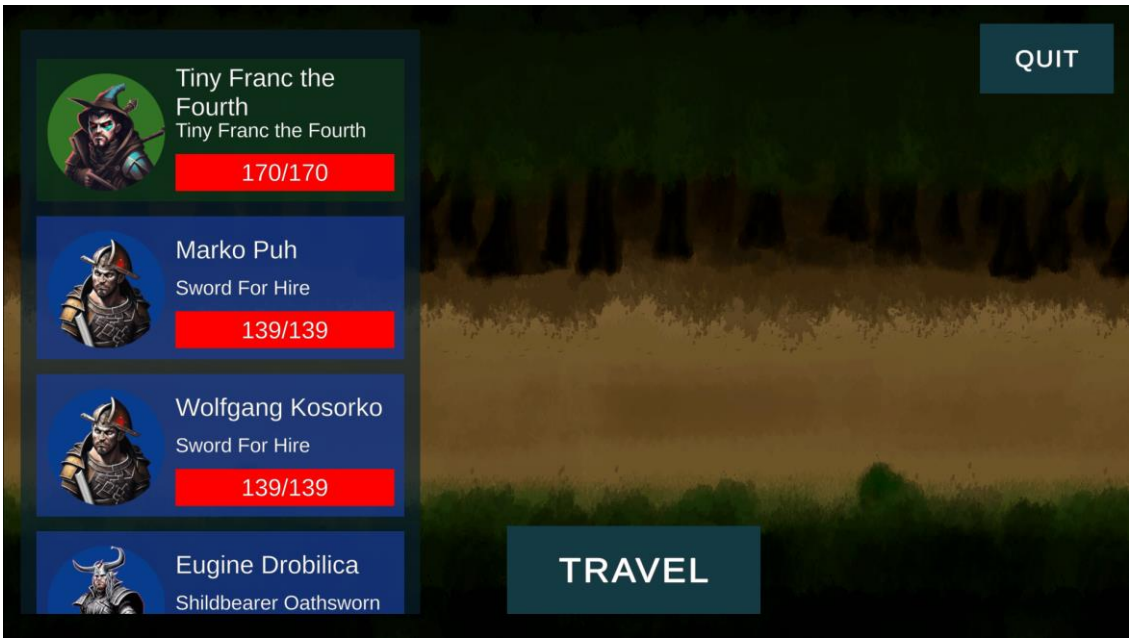
Leonardo.ai sam koristio za generiranje slika karaktera. Generirane slike sam dalje uredio u GIMP-u.

3. Escort Mission

Tema *Escort Mission*-a (ili doslovno na hrvatskom – Misija Praćenja) je putovanje i preživljavanje. Kod pokretanja sesije prvo je potrebno sastaviti družinu (Slika 9), igraču je na raspolaganju 20 bodova koje može (ali ne mora) potrošiti na svoje karaktere. Postoje 3 klase karaktera s kojima igrač može igrati, a da su raspoloživi, to jest njihova dobrobit nije direktno bitna za igru.

Slika 9. Sastavljanje družine

Nakon što igrač potroši svoje bodove, igra prelazi na scenu putovanja (Slika 10) gdje se igrač prvi put susreće s glavnim likom igre. Tiny Franc the Fourth je pratilac (eng. *escortee*) i njegova dobrobit je najbitnija stvar jer kada on (tragično) nastrada, a nastradati će, igra je gotova. Ovakav sustav igru karakterizira kao *„roguelike“* žanr. Ovdje je moguće pregledati članove družine.

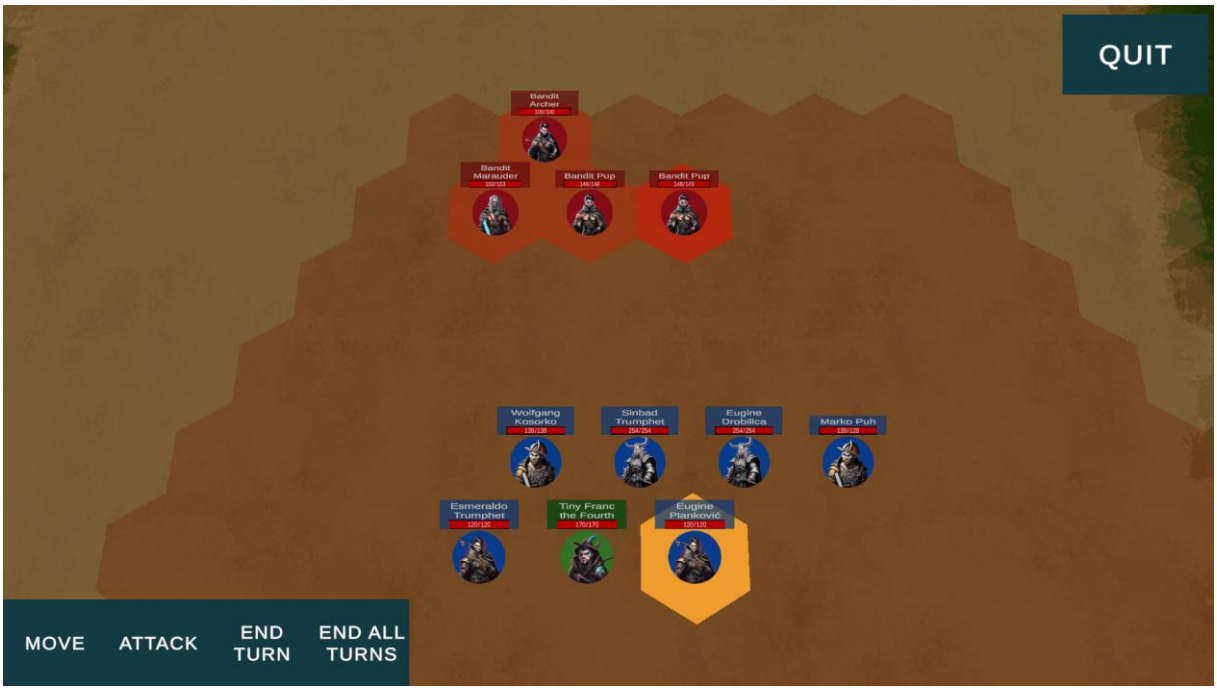


Slika 10. Scena putovanja

Na putovanju, družinu će ubrzo napasti jedna od dvije frakcije; banditi ili dezerteri. Tada igra prelazi na scenu bitke (Slika 12) koja se izvodi na heksagonalnoj mreži/mapi. Prije nego bitka započne, igrač ima priliku rasporediti svoje karaktere na svojem dijelu mape (Slika 11).

Slika 11. Faza razmicanja karaktera

Kada je igrač gotov s raspodjelom karaktera, započinje borba. Igrač uvijek ima prvi potez. Može igrati sa svim svojim karaktere uključujući Tiny Francom. Svaki karakter se može pomicati na drugi heksagon dok ne potroši sve bodove kretnje za taj potez ili dok ne završi svoj potez. Neovisno o tome je li se karakter pomaknu ili ne može probati napasti neprijatelja koji mu je u dometu (odabirom opcije ATTACK (Slika 12) oko karaktera se pojavi crvena zona koja predstavlja domet njegovih napada). Kada karakter izvrši napad njegov potez je automatski gotov i mora čekati drugi krug. Ako igrač želi odustati od napada, može kliknuti gumb MOVE (Slika 12) ili ponovo na njegovog označenog karaktera kao bi poništio odabir.



Slika 12. Scena bitke

Kada igrač završi svoj potez, bilo da da je odigrao sa svim svojim karakterima ili je stisnuo gumb za završetak poteza, započinje „međupotez“. On služi kao jak alat, ali i kao potencijalno velik problem. Tada pratilac ponovo ima priliku za akciju. Može se micati i napadati, ali njime sada upravlja umjetna inteligencija i igrač ne može na nju utjecati direktno, već mora smisliti strategiju unaprijed kojom bi pratioca držao pod kontrolom, to jest van opasnosti. S obzirom na to da pratilac sada ima ponovo priliku za napad, može biti veoma moćan alat kojim se lakše dolazi do pobjede u tekućoj bitci. Stoga ga nije pametno previše držati izvan opasnosti.

Slijedi potez neprijatelja. Ako neprijatelji uspe životne bodove pratioca spustiti na nulu, igra završava.

Potezi se rotiraju sve dok jedna strana nije pobijedila – igrač je uspio poraziti sve neprijatelje ili su neprijatelji porazili pratioca.

Ako igrač pobjedi u bitci, igra se vraća na scenu putovanja (Slika 13) gdje družina putuje do slijedećeg susreta s neprijateljima koji će tada biti jači (ili zbog većeg broja neprijatelja ili zbog jačih neprijatelja) i ciklus života sesije ide u krug. Također, možemo vidjeti i situaciju nakon bitke (Slika 13) gdje Marko Puh više nije s nama, a Wolfgang Kosorko je zadobio ozljede. Sve ozljede su trajne, stoga svaki rizik može imati velike utjecaje na rezultate budućih bitaka.

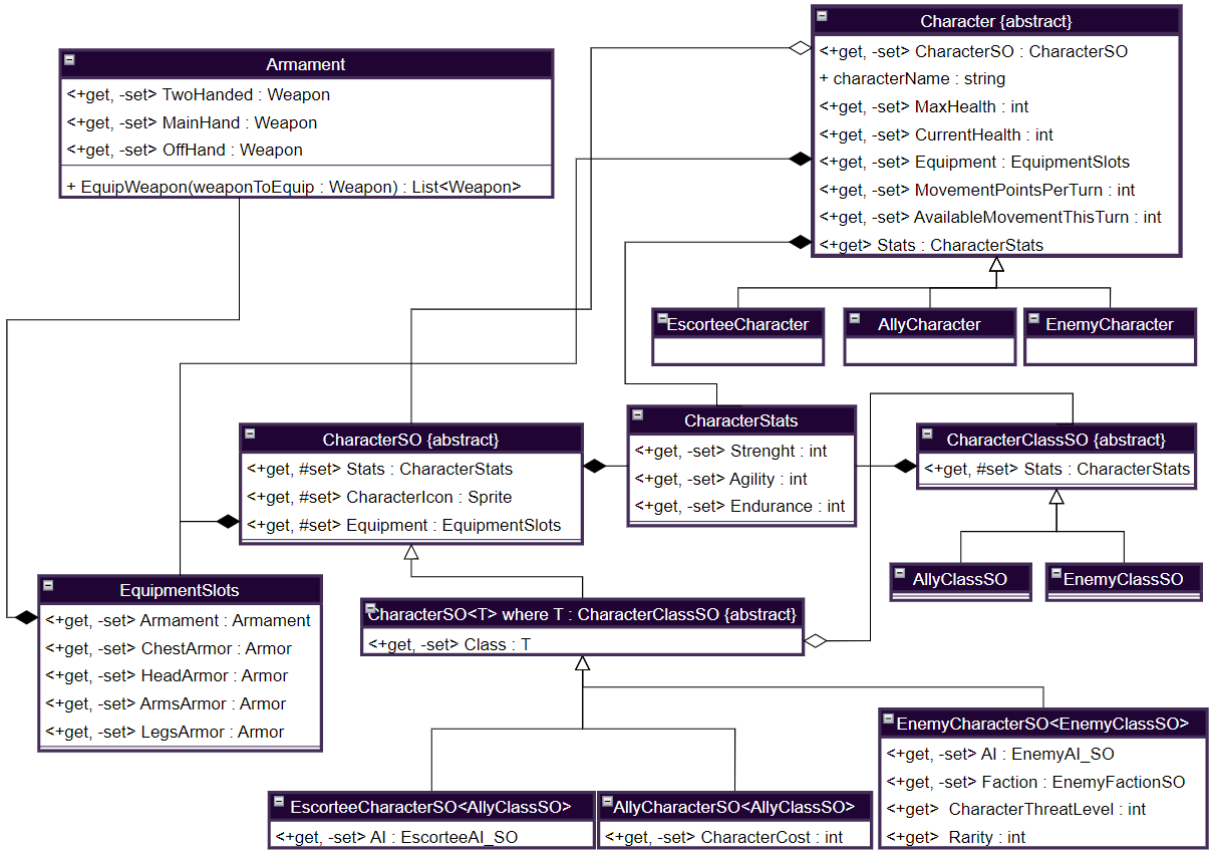
Slika 13. Situacija nakon pobjede u prvoj bitci

3.1. Implementacija i struktura hijerarhije klasa

U Escort Mission-u postoje dvije glavne hijerarhije klasa koje tvore kompleksnije sustave. To su hijerarhija karaktera i hijerarhija opreme.

3.1.1. Hijerarhija karaktera

U Escort Mission-u postoje 3 vrste karaktera; saveznici, neprijatelji i pratioci. U jednoj družini može biti više saveznika, ali samo jedan pratilac. Slijedi dijagram klasa koji opisuje strukturu karaktera (Slika 14).



Slika 14.UML dijagram klasa za karaktere

Iz dijagrama se može vidjeti kako postoje dvije klase karaktera; Character i CharacterSO. Postoji i treća koju ćemo spomenuti kasnije. Character je, nazovimo to, živi karakter. Konkretan primjer bi bio na iznad () – Wolfgang Kosorko. To je jedna individualna osoba. On je stvoren na temelju predloška CharacterSO, konkretnije AllyCharacterSO.

Napomena: sve klase kojima naziv završava na „SO“ su klase koje proširuju ScriptableObject. Odmah prema tome možemo vidjeti kako se radi o resursu.

3.1.1.1. CharacterStats

CharacterStats je jednostavna klasa i njen kôd je prikazan ispod (Slika 15).

Slika 15. CharacterStats polja i svojstva

Prva, najbitnija, stvar je da klasa ima atribut System.Serializable koji označava da se klasa može serijalizirati. To je bitno jer radi tog atribut Unity uređivač serijalizira ovu klasu, te se njene vrijednosti polja mogu vidjeti i uređivati u inspektoru uređivača.

Unatoč spomenutom atributu, Unity normalno serijalizira samo javna polja i automatski generirana svojstva s javnim nabavljačima i postavljačima. Ideja ove klase, a i mnogih drugih koje ćemo spomenuti je da im se vrijednost polja mogu uređivati kroz inspektor Unity uređivača, te da im se kasnije (tijekom izvođenja igre) vrijednosti ne modificiraju; tu je potrebna enkapsulacija.

Uz pomoć atributa SerializeField mogu se serijalizirati privatna polja. Tako je ovdje očuvana enkapsulacija jer se vrijednosti polja mogu dohvatiti preko njihovih svojstava s javnim nabavljačima i bez postavljača. Vrijednosti privatnih polja se mogu urešivati kroz editor jer Unity-eva serijalizacija koristi refleksiju. Dok je to veoma korisna stvar, bitno je napomenuti kako se u tom slučaju direktne izmjene vrijednosti nad serijaliziranim objektom zaobilazi bilo kakva logika postavljanja vrijednosti koja potencijalno postoji. To se donekle može mitigirati korištenjem ranije spomenute funkcije OnValidate.

Ideja je da se sve potrebne operacije nad vrijednostima (sumiranja, modifikacije) izvršavaju preko konstruktora (Slika 16). Prema tome ovdje je ovakav pristup adekvatan.

```
1 reference
public CharacterStats(params CharacterStats[] stats) {
    foreach (var stat in stats) {
        _strenght += stat._strenght;
        _agility += stat._agility;
        _endurance += stat._endurance;
    }

    _strenght = Mathf.Clamp(_strenght, 0, SkillCap);
    _agility = Mathf.Clamp(_agility, 0, SkillCap);
    _endurance = Mathf.Clamp(_endurance, 0, SkillCap);
}

1 reference
public CharacterStats(int strengthModifier, int agilityModifier, int enduranceModifier, CharacterStats baseStats) {
    _strenght = baseStats.Agility + strengthModifier;
    _agility = baseStats.Agility + agilityModifier;
    _endurance = baseStats.Endurance + enduranceModifier;

    _strenght = Mathf.Clamp(_strenght, 0, SkillCap);
    _agility = Mathf.Clamp(_agility, 0, SkillCap);
    _endurance = Mathf.Clamp(_endurance, 0, SkillCap);
}

0 references
public CharacterStats(int strength, int agility, int endurance) {
    _strenght = strength;
    _agility = agility;
    _endurance = endurance;
}
```

Slika 16. CharacterStats konstruktori

Atribut Range služi kako bi se u inspektoru, umjesto brojčanog polja, nacrtao klizač koji ima limitiranu maksimalnu i minimalnu vrijednost. Ispod možemo vidjeti kako ova klasa izgleda u inspektoru (Slika 17). Napomena: kako ova klasa ne proširuje MonoBehaviour ili ScriptableObject, ne možemo ju uređivati kroz inspektor već se ona mora nalaziti kao polje klase koja proširuje spomenute.

Slika 17. CharacterStats u inspektoru

3.1.1.2. CharacterClassSO

Radi se o apstraktnoj klasi koja proširuje ScriptableObject i ima CharacterStats polje. Predstavlja borbenu klasu karaktera i osnovna ideja je ta da karakteru daje osnovne statistike. Na primjer, karakter koji je borbene klase koja je specijalizirana za sukobe na bliskoj razini dobiva dodatne bonuse kada ima visoku snagu. Prema tome, takva borbena klasa bi trebala imati dodatne bodove usmjerene ka snazi. Predstavlja modularnost pri kreaciji karaktera (CharacterSO ScriptableObject-a). U budućnosti, ako će se dodavati nove funkcionalnosti igri, borbena klasa bi mogla definirati stablo vještina, specijalne osobine itd.

Ima dvije konkretne klase koje ju nasljeđuju; AllyClassSO i EnemyClassSO. Dizajn karaktera je osmišljen na način da se neprijatelji dizajniraju odvojeno od saveznika kako bi se stvorila nesimetričnost. Balans saveznika i neprijatelja nije željeno ponašanje, već saveznici moraju biti jači, to jest radi se o asimetričnom dizajnu.

3.1.1.3. CharacterSO

Radi se o apstraktnoj klasi koja bi, idealno, bila generička. Međutim tada ne bi bilo moguće imati serijalizirano polje u klasi Character koje predstavlja njegov CharacterSO jer tada inspektor ne može točno odrediti o kakvom tipu polju se radi, pa nije moguće dodati vrijednost kroz inspektor. Iz tog razloga CharacterSO je malo razdvojen na CharacterSO i generičku klasu CharacterSO<T> gdje je T CharacterClassSO i proširuje CharacterSO poljem _class (Slika 18).

```
public abstract class CharacterSO<T> : CharacterSO where T : CharacterClassSO {
    1 reference
    [SerializeField] private GameEvents.CharacterSOGameEventSO _characterCreated_Event;
    2 references
    private bool _hasAnnouncedItsCreation = false;
    2 references
    [SerializeField] private T _class;


    0 references
    public T Class => _class;

    4 references
    public override CharacterStats GetCharacterStats() {
        return new CharacterStats(_stats, _class.BaseStats);
    }
```

Slika 18. CharacterSO<T>

Na slici se vide i dodatna polja čija je svrha najavljivanje kreacije konkretne implementacije klase. Postoji ScriptableObject klasa CharacterSORepositorySO čija je svrha pohrana svih CharacterSO-a koje igra koristi. Stoga, kada se kreira CharacterSO, on najavljuje svoju kreaciju putem događaja kojim šalje svoju referencu i repozitorij ga pohranjuje. Taj repozitorij kasnije čitaju skripte kojima je to potrebno.

CharacterSO<T> pruža modifikaciju statistika karaktera, to jest CharacterStats. To radi na način da kreira novu instancu koja je suma statistika koju daje CharacterSO i CharacterClassSO (metoda GetCharacterStats).

3.1.1.4. AllyCharacterSO

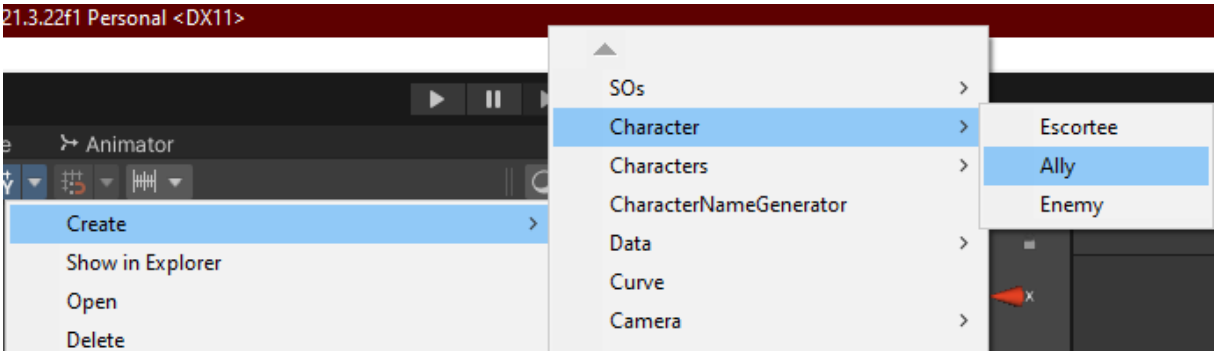
AllyCharacterSO je najjednostavnija specifikacija CharacterSO-a i ima samo dva dodatna polja (Slika 19); _characterCost, kojim je definirana cijena karaktera koja se oduzima od raspoloživih bodova na početku igre u fazi sastavljanja družine, i _characterNameGenerator, ScriptableObject koji sadrži kolekciju imena i prezimena.

```
using UnityEngine;

[CreateAssetMenu(menuName = "Character/Ally")]
6 references
public class AllyCharacterSO : CharacterSO<AllyClassSO> {
    1 reference
    [SerializeField] private int _characterCost;
    1 reference
    [SerializeField] private CharacterNameGeneratorSO _characterNameGenerator;
    5 references
    public int CharacterCost => _characterCost;
    1 reference
    public AllyCharacter CreateCharacterInstance() {
        return new AllyCharacter(this, _characterNameGenerator);
    }
}
```

Slika 19. AllyCharacterSO

Svaka tri specifikacije imaju metodu CreateCharacterInstance s različitim parametrima, ali svrha im je ista; kreirati instancu Character klase.

CreateAssetMenu je atribut kojim se kreira izbornik preko kojega je moguće (i najelegantine) kreirati instancu ScriptableObject-a kroz Unity uređivač (Slika 20). Desni klik u prozoru projekta > Create > putanja definirana atributom (Character/Ally).



Slika 20. Kreiranje ScriptableObject instanci kroz uređivač

3.1.1.5. EscorteeCharacterSO

Implementacija klase se može vidjeti ispod (Slika 21). S obzirom na to da pratioci imaju međupotez, potrebna im je nekakva direktiva kojom će se voditi i koja će im omogućiti izvršavanje akcija. To im pruža klasa EscorteeAI_SO o kojoj ćemo detaljnije kasnije.

```
[CreateAssetMenu(menuName = "Character/Escortee")]
5 references
public class EscorteeCharacerSO : CharacterSO<AllyClassSO> {
    1 reference
    [SerializeField] private EscorteeAI_SO _ai;
    1 reference
    public EscorteeAI_SO AI => _ai;
    2 references
    public EscorteeCharacter CreateCharacterInstance() {
        return new EscorteeCharacter(this);
    }
}
```

Slika 21. EscorteeCharacterSO

Ako će se igra nadograđivati funkcionalnostima, ovdje je moguće definirati novu vrstu borbene klase karaktera specijalno za pratioce koja bi mogla definirati nekakva dodatna ponašanja ili osobine. U tom slučaju samo će biti potrebno promijeniti generički tip i pospajati u inspektoru.

3.1.1.6. EnemyCharacterSO

Ispod je prikazana implementacija klase (Slika 22).

Slika 22. EnemyCharacterSO

Neprijatelji se grupiraju u frakcije. Kada tijekom igre dođe do okršaja s neprijateljima, neprijatelji će svi biti iz iste frakcije. Svaki neprijatelj ima određenu vrijednost koja predstavlja njegovu snagu u odnosu na druge neprijatelje (polje _characterThreatLevel) i vrijednost koja predstavlja koliko je njegova pojava rijetka unutar njegove frakcije. O tome više kasnije.

Igrač nikada ne upravlja neprijateljima, već su njihove akcije upravljane od strane računala. Kao i kod EscorteeCharacterSO klase, potrebno im je pružati način na koji će se ponašati i kojim će moći izvršavati akcije.

3.1.1.7. Character

Character je apstraktna klasa koja definira osnovne podatke o instanci karaktera (Slika 23).

Slika 23. Character, osnovni podaci

Klasa ima neka pomoćna svojstva za lakšu navigaciju svojstvima kao što su IsMelee i IsRanged. Implementiran je negativan modifikator za agilnost ako karakter ima pretešku opremu (Slika 24), kao i metoda za pretvaranje snage i izdržljivosti u životne bodove (Slika 25).

```
public int GetAgilityPenaltyFormEquipmentOverload() {
    int strengthMultiplier = 20;
    int enduranceMultiplier = 10;
    int characterMaxCarryWeight = (CharacterSO.GetCharacterStats().Strenght * strengthMultiplier) +
                                  (CharacterSO.GetCharacterStats().Endurance * enduranceMultiplier);

    int overload = Equipment.GetEquipmentLoad() - characterMaxCarryWeight;

    int agilityModifier = 0;
    while (overload > 0) {
        agilityModifier--;
        overload -= 30;
    }
    return agilityModifier;
}
```

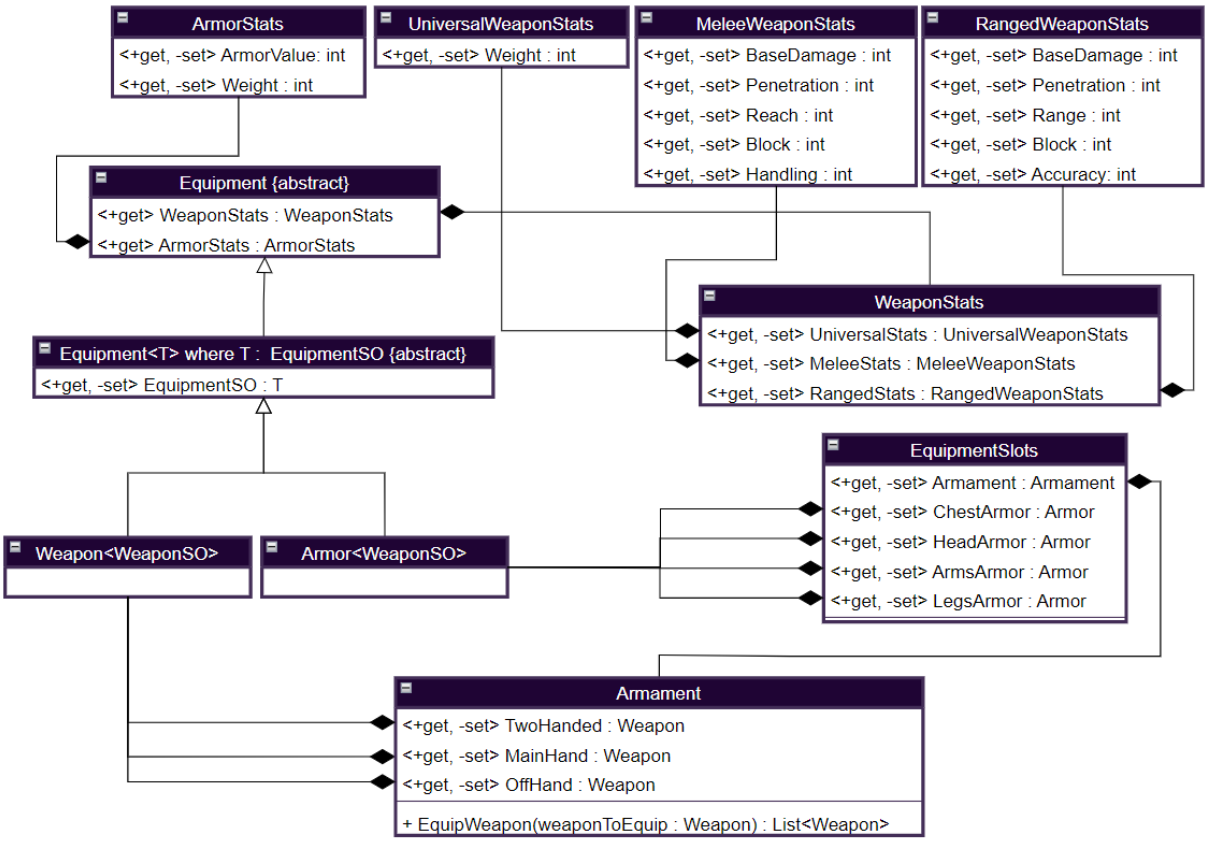
Slika 24. Character, negativni modifikator

```
private int CalculateMaxHealthFromCharacterStats() {
    return Stats.Endurance * 12 + Stats.Strenght * 5;
}
```

Slika 25. Character, računanje životnih bodoba

3.1.2.Sustav opreme

Sustav opreme ima je malo veći, pa mora biti razdijeljen na dva dijela. Ispod je dijagram instanci opreme; oružja i oklopa (Slika 26). Također, prikazani s i sustav opreme koji je bio prikazan ranije iz perspektive karaktera; i klase statistike oružja i oklopa.



Slika 26. Sustav opreme

Klase statistike oružja i oklopa su jednostavne klase čije se ponašanje i osobine mogu jasno vidjeti iz dijagrama, te neće biti specijalno prikazane, a što koja statistika predstavlja će biti objašnjeno kasnije u poglavlju kalkulacija. Imaju iste atribute koji omogućuju serijalizaciju privatnih polja i klizač u inspektoru kao što je objašnjeno kod klase CharacterStats.

3.1.2.1. EquipmentSlots

EquipmentSlots je klasa koja služi kao reprezentacija opreme kojom je karakter opremljen. Armament je reprezentacija naoružanja koje je malo kompleksnije iz razloga što postoje oružja za koja su potrebne dvije ruke, jedna ruka i, specijalno, nedominantna ruka. Oklop je jednostavniji jer postoje četiri mjesta za oklope; glava, ruke, noge i tijelo.

Na slici ispod (Slika 27) je prikazano svojstvo klase EquipmentSlots za nabavljanje oklopa tijela. Ostala mjesta za oklope nećemo prolaziti jer rade na istom principu.

```
[SerializeField] private Armor _chestArmor;
2 references
[SerializeField] private bool _hasChestArmor;
10 references
public Armor ChestArmor {
    get => _hasChestArmor ? _chestArmor : null;
    private set {
        _chestArmor = value;
        _hasChestArmor = _chestArmor != null;
    }
}
10 references
```

Slika 27. EquipmemtSlots, svojstvo ChestArmor

Kao i kod klase CharacterStats, koja je opisana ranije, korištena su privatna polja i svojstva s javnim nabavljačima. Međutim, ovdje postoje i privatni postavljači. To je tako radi serijalizacije. Ideja korištenja je da na klasama koje imaju EquipmentSlots polje postoje polja preko kojih se oružja i oklopi mogu opremiti u mjesta za opremu kroz inspektor. Ispod je prikazan takav primjer na ScriptableObject klasi (Slika 28). Ako se prisjetimo, ScriptableObject-i služe kao resursi i kako bi se moglo definirati koju opremu takav resurs ima bez da se napravi nekakva greška (npr. opremiti oružje za koje su potrebne dvije ruke u jednu ruku) potrebno je proći nekakvu danju obradu. Zbog serijalizacije, ne možemo koristiti postavljače, već ih zaobilazimo. Možemo se poslužiti funkcijom OnValidate koja se poziva kada je vrijednost objekta u inspektoru promijenjena.

```
#if UNITY_EDITOR
    3 references
    [SerializeField] protected WeaponSO _weaponToEquip;
    3 references
    [SerializeField] protected ArmorSO _armorToEquip;
#endif
    0 references
    void OnValidate() {
#if UNITY_EDITOR
        if (_weaponToEquip != null) {
            _equipment.EquipWeaponSO(_weaponToEquip);
            _weaponToEquip = null;
        }
        if (_armorToEquip != null) {
            _equipment.EquipArmorSO(_armorToEquip);
            _armorToEquip = null;
        }
        UnityEditor.EditorUtility.SetDirty(this);
#endif
```

Slika 28. EquipmentSlots, sustav opremanja opreme kroz inspektor

Ovdje je (Slika 28), prije svega, korištena predprocesorska direktiva kojom je označeno da će se obgrljene linije kompilirati samo ako se kôd kompilira u Unity uređivaču. Iako se OnValidate funkcija nikad ne poziva ako se kôd izvršava van uređivača, zato što se koriste polja koja neće uopće biti kompilirana potrebno je linije koje referenciraju polja koja se neće kompilirati također ovako označiti kako bi se kôd mogao kompilirati bez grešaka.

U OnValidate funkciji se provjerava ako je barem jedno od polja _weaponToEquip i _armorToEquip različito od NULL, nedefinirano. Ako je to istina, poziva se metoda koja opremljuje opremu koja je na tom polju i vrijednost tog polja postavlja na NULL. Na kraju se poziva metoda SetDirty koja daje inspektoru doznanja da se „stvarni“ objekt promijenio i kako ga je potrebno ponovo serijalizirati. Na taj način je riješen ovaj problem serijalizacije kod opremanja opreme.

Vratimo li se malo unatrag na klasu EquipmentSlots (Slika 26) dotaknuti ćemo se, u principu, istog problema. Kada bi se vrijednost polja postavila na NULL kroz kôd, serijalizacija bi stvorila novi objekt na njegovom mjestu koji ne bi imao prave, korisne podatke. Usporedba s NULL bi vratila laž iako je taj objekt tehnički NULL. Ovdje je korišten drugačiji pristup s pomoćnim boolean poljima kojima se prati je li objekt NULL.

Klasa Armament čini dio klase EquipmentSlots i predstavlja segment naoružanja, to jest EquipmentSlots ima polje tipa Armament. Funkcionalno ima istu svrhu kao i EuipmentSlots i tu je radi odvajanja ofenzivne i defenzivne opreme.

Armament i EquipmentSlots nakon opremanja opreme vraćaju uklonjenu opremu. EquipmentSlots ima jednostavnu metodu za opremanje dok Aramament ima malo kompleksniju prikazanu ispod (Slika 29).

```
public List<Weapon> EquipWeapon(Weapon weapon) {
    var unequipedWeapons = new List<Weapon>();
    if (weapon.EquipmentSO.Type.Wielding is TwoHandedWieldingSO) {
        unequipedWeapons = EquipTwoHanded(weapon);
    } else {
        Weapon unequipedWeapon;
        if (TwoHanded ≠ null) {
            unequipedWeapon = TwoHanded;
            UnequipTwoHanded();
            unequipedWeapons.Add(unequipedWeapon);
        }
        if (weapon.EquipmentSO.Type.Wielding is OffHandedWieldingSO) {
            if (OffHand ≠ null) {
                unequipedWeapon = OffHand;
                UnequipOffHand();
                unequipedWeapons.Add(unequipedWeapon);
            }
            OffHand = weapon;
        } else if (weapon.EquipmentSO.Type.Wielding is OneHandedWieldingSO) {
            if (MainHand ≠ null) {
                unequipedWeapon = MainHand;
                UnequipMainHand();
                unequipedWeapons.Add(unequipedWeapon);
            }
            MainHand = weapon;
        }
    }
    return unequipedWeapons;
}
```

Slika 29. Armament, opremljivanje

Metoda provjerava o kakvoj vrsti oružja se radi s pogleda na koliko i koja ruka je potrebna za njegovo korištenje i na temelju toga poziva pomoćne funkcije kojima oprema oružje i prikuplja ona koja su uklonjena.

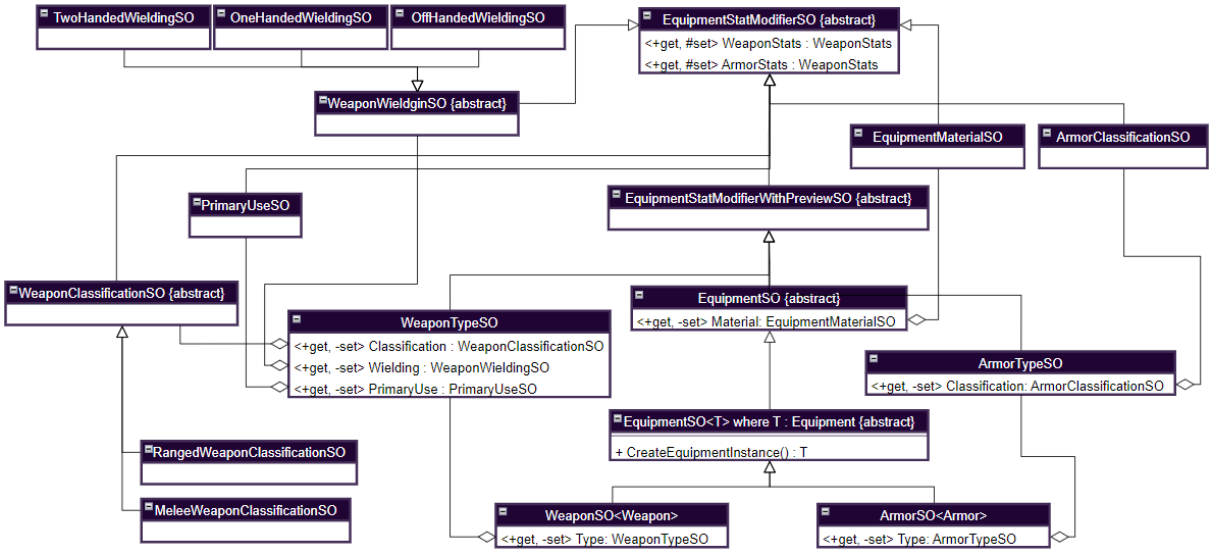
3.1.2.2. Equipment

Equipment je apstraktna klasa koja objedinjuje klasu Weapon i klasu Armor. Kako se ne bi morao specificirati tip opreme, Equipment jer razdvojen na Equipment i generički Equipment<T> gdje je T EquipmentSO. Kao i kod karaktera, ScriptableObject verzija služi kao resurs iz kojeg će biti kreirane instance opreme.

Postoje dvije konkretne klase koje proširuju Equipment<T>; Armor i Weapon.

3.1.2.3. EquipmentSO

Hijerarhija klasa ScriptableObject-a EquipmentSO je prikazana ispod (Slika 30).



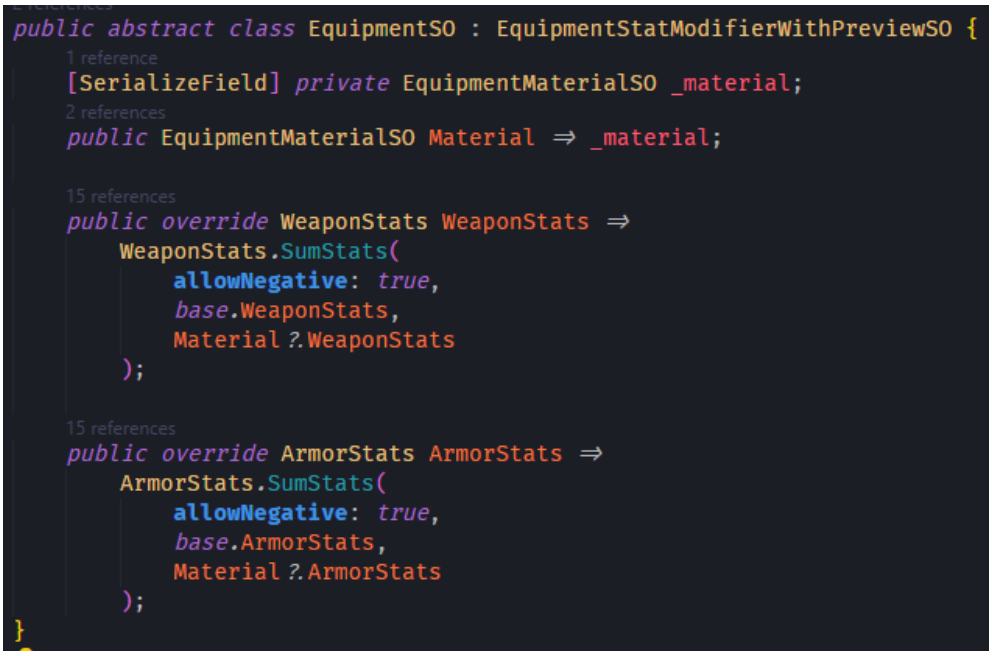
Slika 30. EquipmentSO hijerarhija

EquipmentStatModifierSO je osnovna klasa koju proširuje sva oprema i njeni dijelovi. Ima dva polja tipa WeaponStats i ArmorStats.

EquipmentStatModifierWithPreviewSO služi za prilagođen inspektor koji dodaje botun u inspektor i polja statistike koja služe za pregled sumiranja statistika koje daje element (npr. WeaponTypeSO) i njegova agregacija (npr. WeaponClassificationSO). Prilagođeni inspektori se kreiraju specijalno za pojedine klase s opcijom uključivanja i klasa koje ih proširuju. Na žalost, može stvoriti potrebu za ovakvim razdvajanjem klasa.

EquipmentSO klasa je razdijeljena na osnovnu i generičku klasu iz istog razloga kao CharcterSO, ranije objašnjen.

Oružja i oklopi dijele materijale, tako da EquipmentSO ima agregaciju s EquipmentMaterialSO. Ispod je prikazan kôd (Slika 31) kojim se sumiraju statistike opreme u ovom stadiju preko svojstava. Ako je parametar allowNegative postavljen na istinu, funkcija SumStats neće stegnuti vrijednosti statistika u rasponu od nula do maksimalne vrijednosti statistike.



Slika 31. EquipmentSO

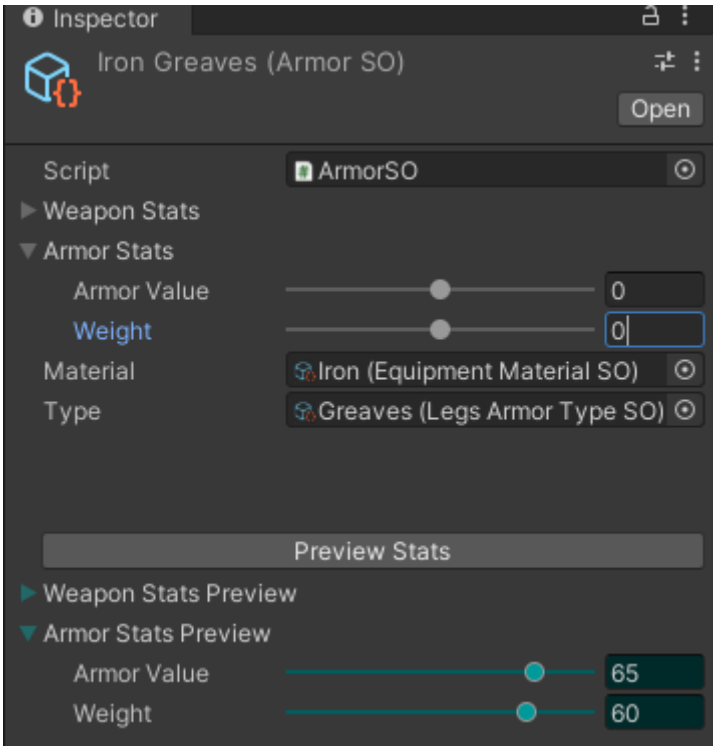
3.1.2.4. ArmorSO

Oklopi su sastavljeni agregacijom s klasom ArmorTypeSO (Slika 32). Oboje su naslijeđene klase od klase EquipmentStatModifierSO i radi toga imaju polja statistika opreme (ArmorStats i WeaponStats).

Slika 32. ArmorTypeSO

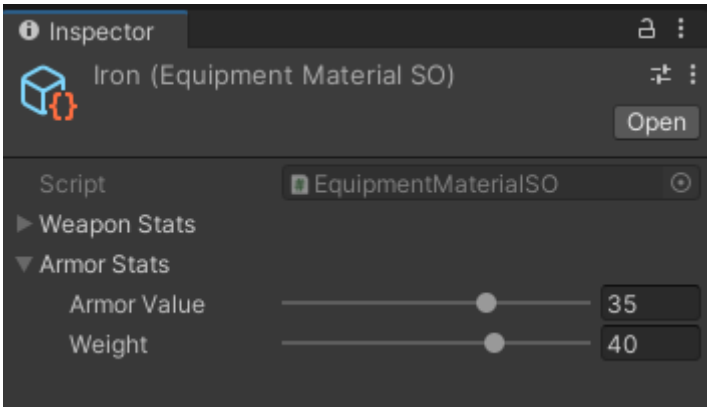
Kako se radi o „završnom proizvodu“ svih potrebnih agregacija, svojstva WeaponStats i ArmorStats imaju parametar allowNegative metode SumStats postavljen na laž. Time se osigurava kako će vrijednosti statistika biti pozitivne ili nula. Prvo će se sakupiti statistike naslijeđenog nabavljača statistike koji nije stegnut, a potom će se nestegnute vrijednosti zbrojiti s nestegnutim statistikama koje pruža ArmorTypeSO, te se na kraju stegnuti u rasponu od nula do maksimalne vrijednosti pojedine statistike. Time je omogućena funkcionalnost negativnih modifikatora, bez da „završna verzija“ ima negativne statistike.

Primjer agregacije: ako imamo oklop „željezni panciri za noge“ (Slika 33), on je stvoren agregacijom EquipmentMaterialSO željezo i ArmorTypeSO pancir za noge.



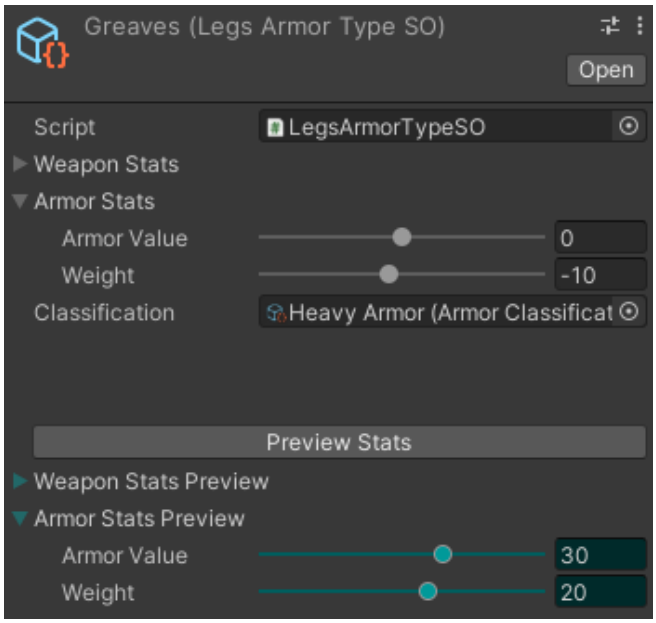
Slika 33. ArmorSO, željezni panciri za noge

U ovom slučaju ArmorSO „željezni pancir za noge“ nema specijalne modifikatore statistika oklopa (ArmorStats), ali se može vidjeti kako njegova finalna agregacija ipak ima statistiku oklopa. MaterialSO „željezo“ ima statistike kako je prikazano ispod (Slika 34).

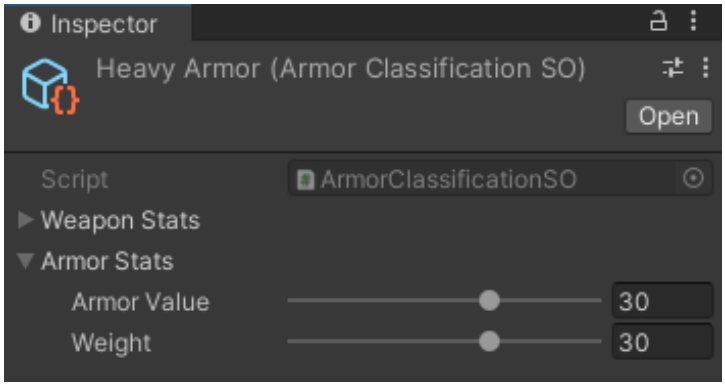


Slika 34. EquipmentMaterial, željezo

ArmorTypeSO „pancir za noge“ ima statistiku oklopa prikazanu ispod (Slika 35) i može se vidjeti kako ima modifikator koji umanjuje težinu oklopa za 10, ali zbog svoje agregacije s ArmorClassificationSO „težak oklop“ (Slika 36), njegova finalna statistika oklopa je drugačija.



Slika 35. LegsArmorTypeSO, pancir za noge

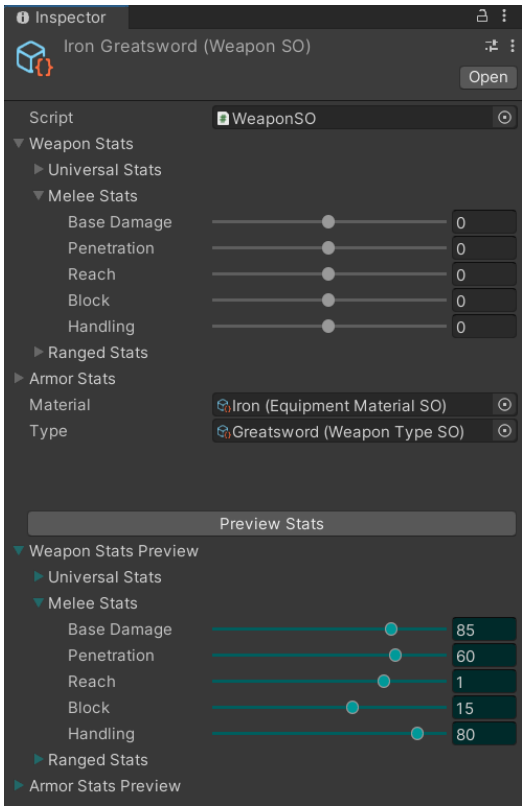


Slika 36. ArmorClassificationSO, težak oklop

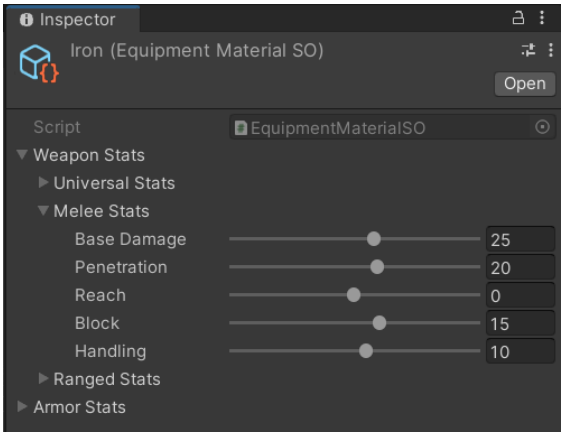
ArmorClassificationSO nije proširena klasa klase EquipmentStatModifierWithPreviewSO, to jest radi se o osnovnoj agregacijskoj jedinici, i zbog toga nema prilagođen inspektor kao ArmorSO.

3.1.2.5. WeaponSO

WeaponSO je nešto kompleksniji kada je riječ o njegovom sastavu, ali sumiranje statistika radi na isti način kao kod ArmorSO-a. Ispod je prikazan primjer za „željezni veliki mač“ (Slika 37).

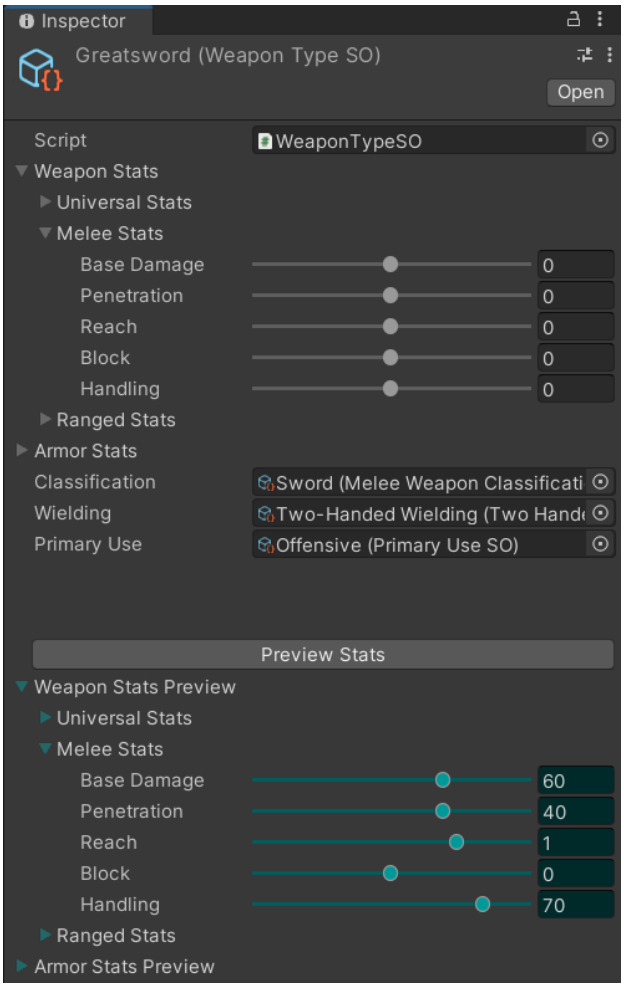


Slika 37. WeaponSO, željezni veliki mač



Slika 38. EqupipmentMaterialSO, željezo (WeaponStats)

Radi se o sastavu materijala „željezo“ (Slika 38) (isti primjer kao i kod ArmorSO-a, samo što je sakrivena statistika oklopa jer ju oružja ne koriste) i WeaponTypeSO „veliki mač“. Slijede slike svih agregacija ovog WeaponSO-a (Slika 39) (Slika 40) (Slika 41) (Slika 42).



Slika 39. Veliki mač

Slika 40. Mač

Slika 41. Oružje za dvije ruke

Slika 42. Napadačko oružje

Na ovakav način je izrada opreme inicijalno kompleksnija, ali kada se jednom dobro definiraju elementi sastava i njihove statistike opreme, buduću opremu je vrlo jednostavno kreirati spajanjem elemenata u cjeline kroz inspektor. S obzirom na to da i „završni proizvodi“ WeaponSO i ArmorSO imaju polja kojima mogu dodatno utjecati na svoje statistike, ako je potrebna manja korekcija statistika, moguće ju je na njima izvesti. To nije preporučeno jer bi prekomjerno korištenje te funkcionalnosti moglo narušiti cijelu svrhu ovakvog sustava agregacija.

3.2. Heksagoni

Kako se bitka izvodi na mreži heksagona potrebno je kreirati heksagone i rasporediti ih u mrežu. Za koordinate su korištene aksijalne koordinate (eng. *axial coordinates*).

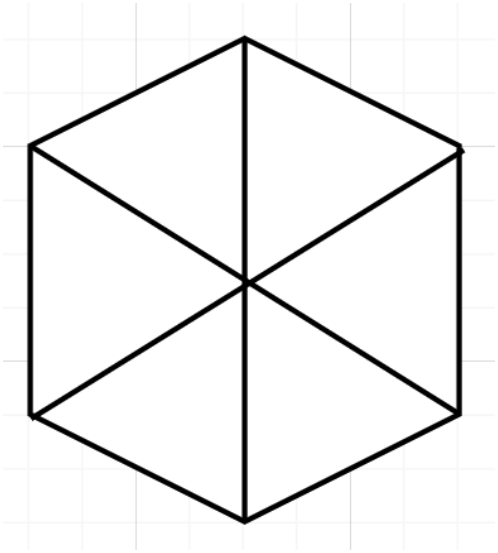
Ako gledamo na mrežu heksagona u 2D prostoru, ona se razlikuje od mreže kvadrata po tome što ju možemo gledati kao da ima tri osi umjesto dvije kod mreže kvadrata. Ako heksagone rasporedimo na ravninu $x + y + z = 0$ možemo dobiti koordinate bilo kojeg

heksagona. Tada možemo koristiti vektorske operacije kako bismo dobili njegove susjede i time imamo kompletan koordinatni sustav. Ovakav tip koordinata se naziva kubne koordinate (eng. *cube coordinates*). [18]

Ako jednadžbu ravnine malo preoblikujemo dobijemo jednadžbu $x + y = -z$. Ovom jednadžbom možemo koordinate definirati preko samo dvije komponente koordinate jer preko njih možemo dobiti treću komponentu koordinate. Promijenimo li nazivlje kako ne bi došlo do zabune s koordinatnim osima sustava, dobivamo formulu $q + r = -s$, to jest koordinate oblika (q, r) i one se zovu aksijalne koordinate. [18]

3.2.1. Prikazivanje heksagona

Potrebno je kreirati objekt koji ima oblik heksagona. Tome služi skripta HexRenderer. Heksagon je sastavljen od šest trokuta kao što je prikazano na ispod (Slika 43). Cilj je stvoriti 2D heksagon koji je orijentiran tako da ime vrh prema gore.



Slika 43. Heksagon podijeljen na trokuta, vrh prema gore

Za stvaranje mrežnog modela Unity koristi trokute kao osnovni građevni blok. Prema tome, ako želimo stvoriti mrežni model heksagona kroz kôd, moramo stvoriti šest trokuta i posložiti ih na način na koji tvore heksagon.

```
4 references
public struct Triangle {
    2 references
    public List<Vector3> Vertices { get; private set; }
    2 references
    public List<int> Triangles { get; private set; }
    2 references
    public List<Vector2> UVs { get; private set; }

    1 reference
    public Triangle(List<Vector3> vertices, List<int> triangles, List<Vector2> uvs) {
        this.Vertices = vertices;
        this.Triangles = triangles;
        this.UVs = uvs;
    }
}
```

Slika 44. Struktura Triangle

Potrebna je struktura Triangle (Slika 44) koja će predstavljati trokut. Mora imati kolekciju vrhova, njihovih poveznica (put od točke do točke i mora tvoriti jedan ili više trokuta) i UV koordinata. UV koordinate služe za mapiranje tekstura na model. Iako za heksagon nisu korištene teksture, one ipak postoje na materijalu, to jest materijal neće ispravno raditi kao ne složimo UV koordinate.

Na slici ispod (Slika 45) je prikazan proces stvaranje šest trokuta.

```
private void CreateTriangles() {
    _trianges = new List<Triangle>();

    for (int point = 0; point < 6; point++) {
        _trianges.Add(CreateTriangle(radius, point));
    }
}

1 reference
private Triangle CreateTriangle(float hexRad, int index) {
    Vector3 pointA = Vector3.zero;
    Vector3 pointB = GetPoint(hexRad, (index < 5) ? index + 1 : 0);
    Vector3 pointC = GetPoint(hexRad, index);

    var vertices = new List<Vector3>() { pointA, pointB, pointC };
    var triangles = new List<int>() { 0, 1, 2 };
    var uvs = new List<Vector2>() { Vector2.one, Vector2.zero, Vector2.right };

    return new Triangle(vertices, triangles, uvs);
}

2 references
private Vector3 GetPoint(float size, int index) {
    float deg = 60 * index - 30;
    float rad = Mathf.PI / 180f * deg;
    return new Vector3(size * Mathf.Cos(rad), size * Mathf.Sin(rad), 0);
}
```

Slika 45. Proces stvaranja 6 trokuta

Metoda CreateTriangle stvara trokut i koristi pomoćnu metodu GetPoint za računanje vrhova. Nužno je koristiti Vector3 jer mrežni model koristi Vector3.

Metoda GetPoint radi na slijedećem principu. Unutarnji kut pravilnog heksagona je 60°. Indeksom je određeno o kojem trokutu po redu se radi. S obzirom na to da je željeni heksagon orijentiran tako da ima vrh prema gore, oduzimamo 30°. Slijedeće je potrebno izračunati vrijednost kuta u radijanima koji je potreban metodama za računanje vrijednosti sinusa i kosinusa kuta. Time dobivamo x i y koordinate tražene točke. Z koordinata je 0 jer nije potrebna.

Vratimo li se na metodu CreateTriangle, sada je moguće izračunati točke koje čine trokut. Prva točka može uvijek biti (0, 0), dok je za drugu i treću potrebno koristiti metodu GetPoint.

Unity crta mrežni model u smjeru suprotnom od kazaljke na satu. Poredak vrhova, trokuta i UV koordinata bitan za ispravan izgled mrežnog modela. Ako je nešto krivo spojeno, mrežni model može biti nacrtan naopačke. Točnije ako 2D mrežni model će biti vidljiv s jedne

strane, ali ne i druge; tada će biti proziran. Moguće je da je jedan trokut krivo okrenut i onda nastaje „rupa u modelu“.

Kada su svu trokuti kreirani, metodom CombineTirangles (Slika 46) se spajaju u mrežni model.

```
private void CombineTriangles() {
    var vertices = new List<Vector3>();
    var triangles = new List<int>();
    var uvs = new List<Vector2>();

    for (int i = 0; i < _trianges.Count; i++) {
        vertices.AddRange(_trianges[i].Vertices);
        uvs.AddRange(_trianges[i].UVs);

        int offset = 3 * i;
        foreach(int triangle in _trianges[i].Triangles) {
            triangles.Add(triangle + offset);
        }
    }

    _mesh.vertices = vertices.ToArray();
    _mesh.triangles = triangles.ToArray();
    _mesh.uv = uvs.ToArray();
    _mesh.RecalculateNormals();
}
```

Slika 46. Spajanje trokuta u heksagon

3.2.2. Skripta Hexa

Hexa daje funkcionalnosti heksagon objektu igre (Slika 47). Preko nje se mogu nabaviti koordinate heksagona, može se provjeriti ima li objekt karaktera na sebi, to jest nalazi li se karakter na heksagonu. Referencu na objekt tipa Character nabavlja preko skripte CharacterMB koja je u stvari kripta koja daje funkcionalnosti objektu igre koji predstavlja Character objekt.

```
public class Hexa : MonoBehaviour {
    33 references
    public HexCoords coords;
    9 references
    private CharacterMB _characterMB;
    2 references
    private InteractableHex _interactableHex;
    23 references
    public Character Character => _characterMB?.Character;
    14 references
    public CharacterMB CharacterMB => _characterMB;
    35 references
    public InteractableHex InteractableHex => _interactableHex;

    1 reference
```

Slika 47. Hexa, polja i svojstva

3.2.3. HexCoords

HexCoords je struktura aksijalnih koordinata heksagona (Slika 48).

```
[System.Serializable]
51 references
public struct HexCoords {
    10 references | 16 references
    public int q, r;
    4 references
    public HexCoords(int q, int r) {
        this.q = q;
        this.r = r;
    }

    23 references
    public enum Direction {
        3 references
        west,
        3 references
        east,
        3 references
        northwest,
        3 references
        northeast,
        3 references
        southwest,
        3 references
        southeast
    }

    2 references
    public int GetS() => -q -r;
```

Slika 48. HexCoords, polja i smjerovi

Ona ima nekoliko metoda za dohvaćanje jednog ili svih susjeda s raznim preopterećenjima parametara. Metoda koja vraća koordinate susjednog heksagona je prikazana ispod ().

```
public HexCoords GetNeighbour(Direction direction) {
    int q;
    int r;
    switch(direction) {
        case Direction.west: q = -1; r = 0; break;
        case Direction.east: q = 1; r = 0; break;
        case Direction.northwest: q = -1; r = 1; break;
        case Direction.northeast: q = 0; r = 1; break;
        case Direction.southwest: q = 0; r = -1; break;
        case Direction.southeast: q = 1; r = -1; break;
        default: q = 0; r = 0; break;
    }
    return new HexCoords(this.q + q, this.r + r);
}
```

Slika 49. HexCoords, GetNeighbours()

Također, računa Manhattansku metodu računanja udaljenost između dva heksagona (Slika 50).

```
11 references
public static int GetDistanceBetweenHexes(HexCoords hex1, HexCoords hex2) {
    int q = Mathf.Abs(hex1.q - hex2.q);
    int r = Mathf.Abs(hex1.r - hex2.r);
    int s = Mathf.Abs((hex1.GetS() - hex2.GetS()));

    return (q + r + s) / 2;
}
```

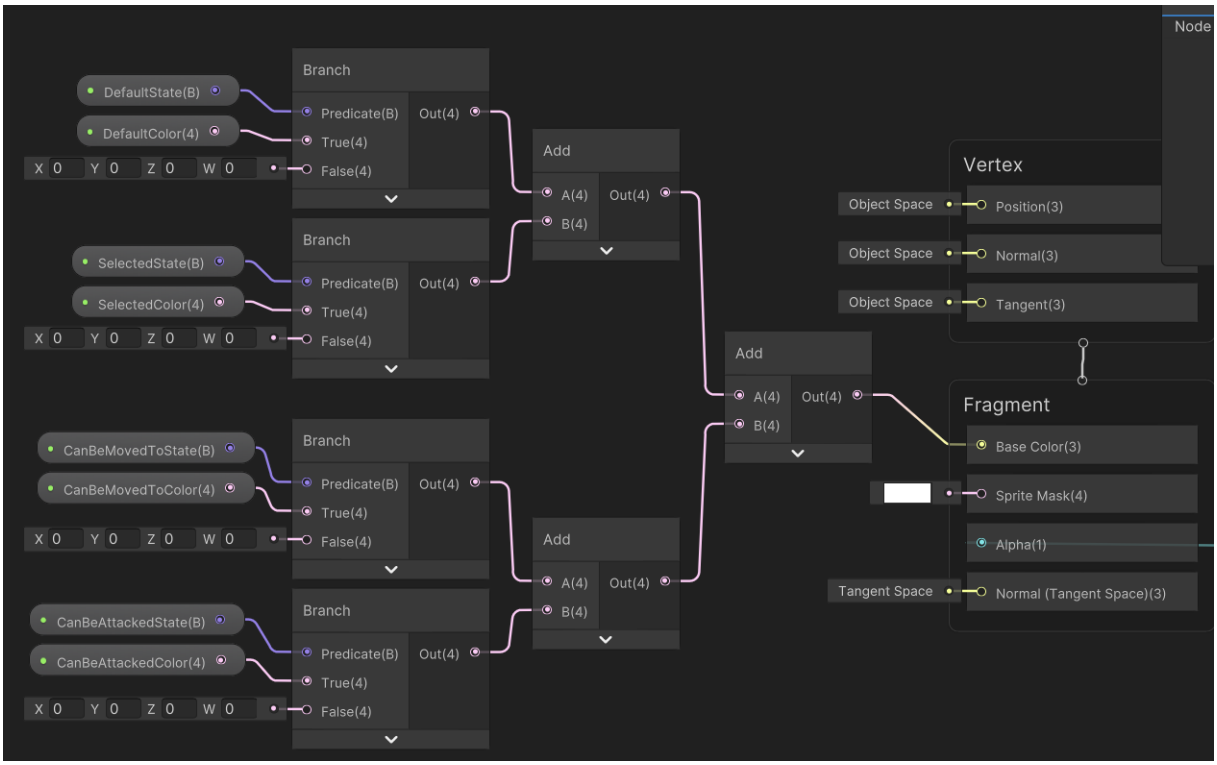
Slika 50. HexCoords, Manhattanska udaljenost

3.2.4. HexState

HexState je statička klasa koja služi direktno upravljanje stanjima (koja su međusobno isključiva) materijala heksagona. Prikazana je ispod (Slika 51).

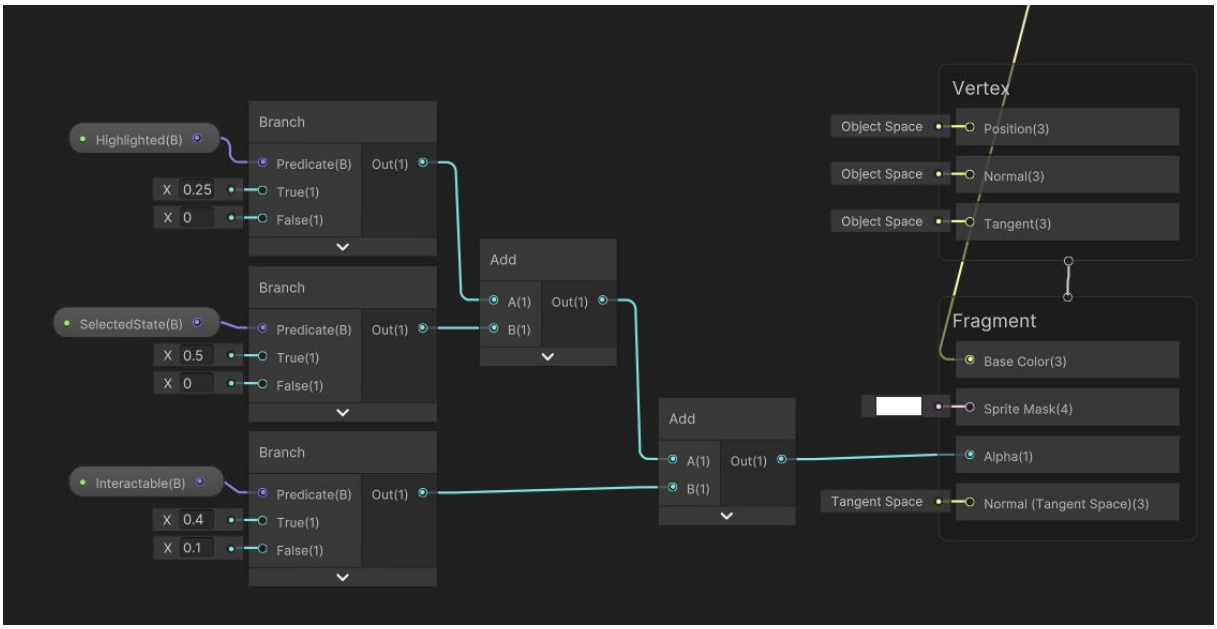
Slika 51. HexState

Preko nabrajača (eng. *enumerator*) objekti koji koriste ovu klasu mogu postavljati stanja materijala heksagona. Svojstva materijala se mijenjaju preko funkcije SetFloat. Za to postoji prilagođen program za računanje sjene (eng. *shader*). Kreiran je pomoću Unity-evog ShaderGraph-a i njegov dijagram je razdvoje na dva dijela radi lakšeg prikaza; mijenjanje boje (Slika 52) i mijenjanja alfa vrijednosti (eng. *alpha*) (Slika 53).



Slika 52. ShaderGraph, boje materijala

Postoje četiri stanja koja mogu biti istina ili laž: DefaultState, SelectedState, CanBeMovedToState i CanBeAttackedStats. Svako od tih stanja ima definiranu boju. Preko čvorova grananja (eng. *branch*) se odlučuje koja boja će proći dalje na čvorove zbrajanja. Ako je DefaultState istina, izlaz čvora grananja će biti SelectedColor. Ako je laž, izlaz će biti crna boja. Isto vrijedi i za ostale čvorove grananja, to jest ako su vrijednosti stanja istina, njihove boje će biti izlaz iz čvora grananja. Preko čvorova zbrajanja se svi izlazi zbrajaju i na kraju materijal poprima boju koja je mješavina svih izlaza čvorova grananja. To nije problem jer je u kôdu implementirano da samo jedno stanje može biti aktivno – neće doći do miješanja boja.



Slika 53. ShaderGrpah, alfa

Ovaj dio grafa razmatra tri dodatna stanja koja nisu međusobno isključiva: Highlighted i Interactable, te ranije spomenuti SelectedState. Zadaća ovog dijela grafa je promjena alfa vrijednost materijala heksagona. Ako je Highlighted istina, iz čvora grananja će izaći vrijednost 0,25, u suprotnome će izaći 0. Ako je SelectedState istina, iz čvora grananja će izaći vrijednost 0,5, u suprotnome 0. Isto vrijedi i za Interactable osim što će izlaz viti 0,4 u slučaju kad je Interactable istina i 0 ako je laž. Izlazi čvorova grananja se zbrajaju kroz čvorove zbrajanja i na kraju zbroj predstavlja alfa vrijednost materijala.

3.2.5. InteractableHex

InteractableHex je skripta koja prati stanje materijala heksagona i može ga mijenjati preko HexState statične klase. Implementacija je prikazana ispod (Slika 54).

```
public class InteractableHex : MonoBehaviour {
    2 references
    private MeshRenderer _meshRenderer;
    5 references
    private HexState.State _currentState;
    3 references
    private bool _checkForMouseOver = false;

    5 references
    public bool IsInteractable { get; private set; } = false;
    4 references
    private Material HexMaterial => _meshRenderer.material;

    0 references
    void Awake() {
        gameObject.AddComponent<MeshCollider>();
        _meshRenderer = GetComponent<MeshRenderer>();
    }

    0 references
    void OnMouseEnter() {
        SetHighlighted(true);
    }
    0 references
    void OnMouseExit() {
        SetHighlighted(false);
    }
    0 references
    void OnMouseOver() {
        if (_checkForMouseOver) {
            SetHighlighted(true);
            _checkForMouseOver = false;
        }
    }
```

Slika 54. InteractableHex, polja i svojstva

Potrebne su komponente MeshRenderer i MeshCollider. MeshRenderer je potreban kako bi se mogla mijenjati svojstva materijala heksagona, a MeshCollider kako bi se mogle detektirati interakcije s heksagonom. Koristi funkcije OnMouseEnter, OnMouseExit i OnMouseOver. Željena funkcionalnost je ta da kada korisnik kursorom prelazi preko heksagona, on promijeni stanje svojeg materijala. Ova skripta ne detektira klik na objekt.

Koristi slijedeće funkcije za kontrolu stanja heksagona (Slika 55).

```
public void SetInteractable() {
    if (!IsInteractable) {
        IsInteractable = true;
        HexMaterial.SetFloat("_Interactable", 1f);
        _checkForMouseOver = true;
    }
}
8 references
public void SetUninteractable() {
    if (IsInteractable) {
        HexMaterial.SetFloat("_Interactable", 0f);
        IsInteractable = false;
    }
}

7 references
public void SetSelectedState() {
    if (_currentState is not HexState.State.SELECTED) {
        SetState(HexState.State.SELECTED);
    }
}
2 references
public void SetMoveableToState() {
    if (_currentState is not HexState.State.MOVEABLE) {
        SetState(HexState.State.MOVEABLE);
    }
}
4 references
public void SetAttackableHexState() {
    if (_currentState is not HexState.State.ATTACKABLE) {
        SetState(HexState.State.ATTACKABLE);
    }
}
5 references
public void SetDefaultState() {
    if (_currentState is not HexState.State.DEFAULT) {
        SetState(HexState.State.DEFAULT);
    }
}

3 references
private void SetHighlighted(bool isHighlighted) {
    HexMaterial.SetFloat("_Highlighted", isHighlighted ? 1f : 0f);
}

4 references
private void SetState(HexState.State state) {
    _currentState = state;
    HexState.SetState(HexMaterial, state);
```

Slika 55. InteractableHex, funkcije postavljanja stanja

3.3. Scene

Klasičan način rada sa scenama u Unity-u je taj da je samo jedna scena učitana. Kada se prelazi na drugi scenu, trenutno otvorena scena se mora istovariti i nova scena se mora

učitati. Ako se želi neki objekt igre koristit u više scena (prisjetimo se: kada se scena istovari, svi objekti igre, a time i njihove komponente se istovaruju iz memorije i time nije moguće održati instancu objekta igre) i bitno ne želi ga se, iz bilo kojeg razloga (npr. očuvanje podataka objekta koji se postavljaju na jadnoj scene i čitaju na drugoj), istovariti i ponovo učitavati, potrebno je koristiti druge metode.

3.3.1. Metode komuniciranja između scena

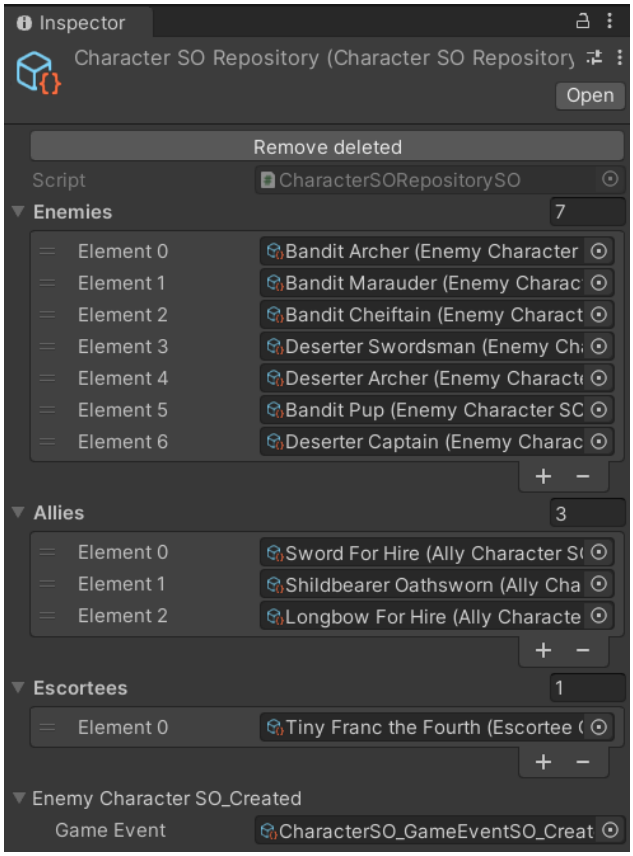
Biti će opisane tri metode rješavanja spomenutog problema i one nisu međusobno isključive, ali imaju svoje prednosti i mane.

3.3.1.1. Metoda DoNotDestroyOnLoad

Jedna od tih metoda je korištenje specijalne metode unutar MonoBehaviour skripte – DoNotDestroyOnLoad koja za parametar prima objekt igre. Ovim pristupom može doći do curenja memorije, pa se treba koristiti razborito. Osobno, ovaj pristup smatram nezgrapnim.

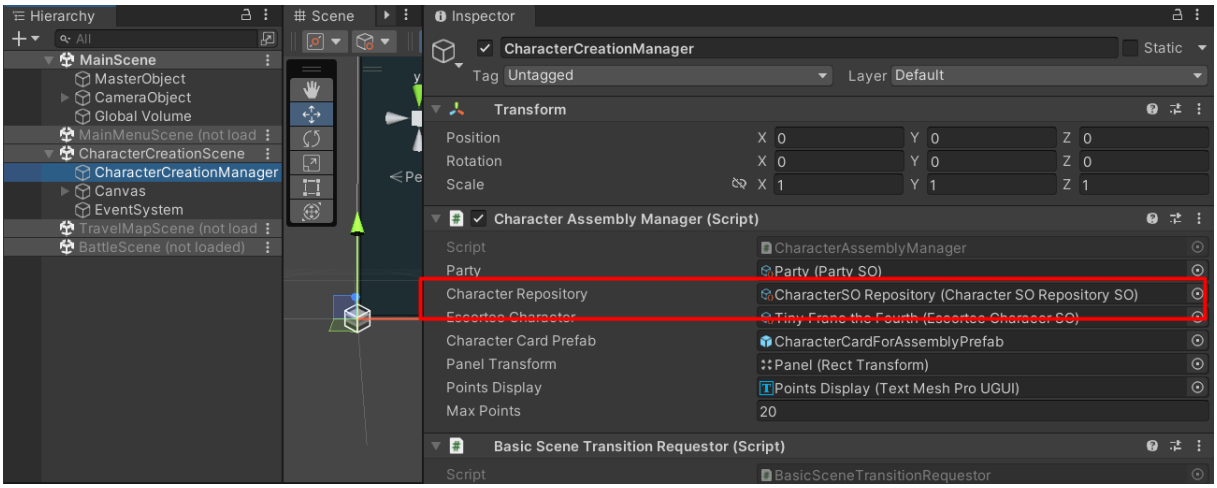
3.3.1.2. Korištenje ScriptableObject-a

Drugi pristup može biti korištenje ScriptableObject-a (ako je to moguće). Primjer takve klase se može vidjeti ispod (Slika 56).



Slika 56. CharacterSORepositorySO, inspektor

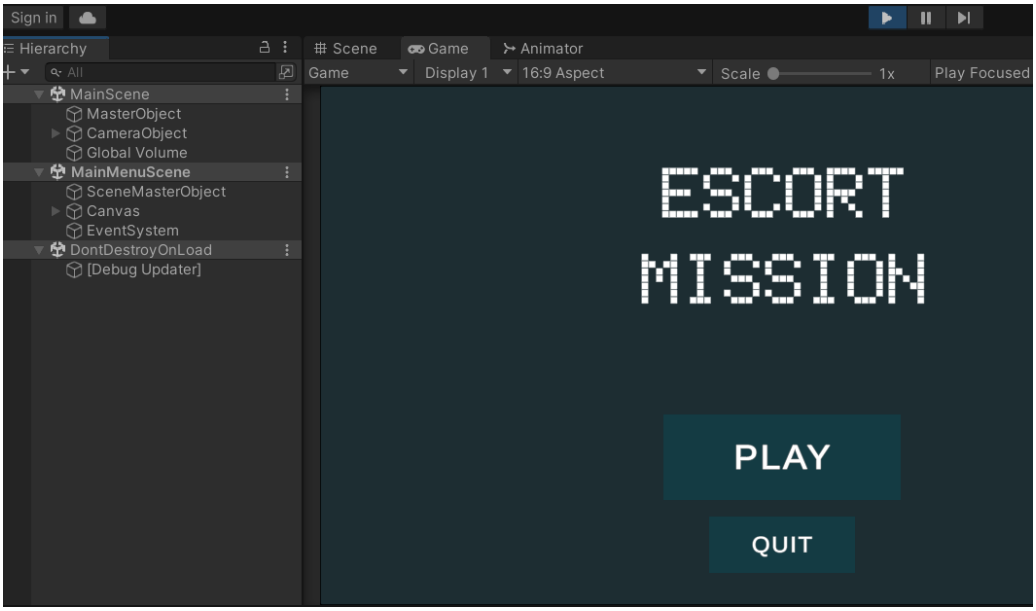
Radi se o ranije spomenutom CharacterSORepositorySO ScriptableObject-u. Njegova zadaća je pružati kolekciju CharacterSO ScriptableObject-a. S obzirom da se radi o ScriptableObject-u, on postoji izvan životnog ciklusa scena i može se referencirati preko polja na MonoBeahaviour skriptama. Takav primjer se može vidjeti ispod (Slika 57) na skripti CharacterAssemblyManager kojoj je potreban popis AllyCharacterSO-eva koji se koriste u igri kako bi mogao generirati listu na kojoj igrač sastavlja družinu.



Slika 57. ScriptableObject instanca kao polje na MonoBehaviour skripti u inspektoru

3.3.1.3. Aditivno učitavanje scena

Moguće je učitavanje više scena bez da se prethodna istovari.



Slika 58. Učitane dvije scene odjednom

Tada se najčešće koristi jedna scena koja će uvijek biti učitana. U slučaju na primjeru (Slika 58) gdje je MainScene scena koja je uvijek učitana i ima objekt igre kameru. Svaka scena treba igre treba kameru. ovim pristupom postoji samo jedna kamera čiji se parametri mijenjaju preko događaja koji upravljač kamere sluša, a doziva se kada se druga scena učita i preko tog događaja se šalje specijalna klasa koja sadrži parametre kamere.

Vidljiva je i scena DontDestroyOnLoad koja je Unity automatski stvorio radi ispisa u konzolu.

Ova metoda također ima mane. Nije moguće direktno referencirati objekte igre iz jedne scene u drugoj kroz inspektor. Ako postoji platno, njemu je potreban sustav događaja, a nije moguće imati dva ili više sustava događaja odjednom (barem ne bez grešaka).

3.4. MainScene

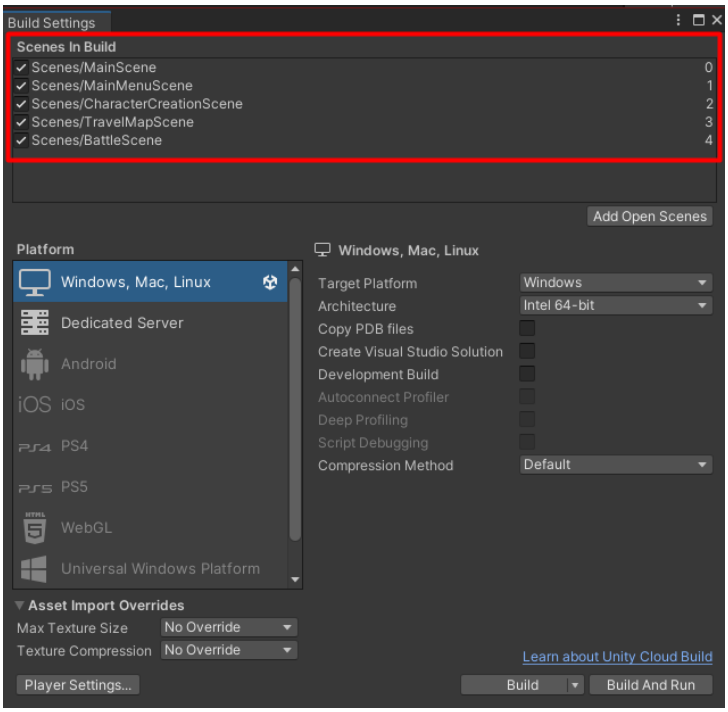
MainScene je scena kojoj je glavna zadaća slušati za zahtjeve prelaska na drugu druge scene i prijelaz izvršiti, to jest zatvoriti scenu kuju treba zatvoriti i otvoriti onu koju treba otvoriti. To je omogućeno preko skripte SceneTransitioner.

3.4.1. SceneTransitioner

```
public enum ScenesInBuild {
    0 references
    MainScene,
    3 references
    MainMenuScene,
    2 references
    CharacterCreationScene,
    3 references
    TravelMapScene,
    3 references
    BattleMapScene
}
```

Slika 59. SceneTransitioner, nabrajač scena

Ona ima nabrajač (Slika 59) koji predstavlja scene u izgradnji igre. Poredak scena nabrajača je bitan i mora se poklapati sa scenama u izgradnju i postavkama izgradnje (Slika 60). Iako se MainScene scena neće nikada istovariti tijekom igre, ona mora biti na popisu jer bez nje neće niti jedna druga scena biti učitana. Unity će automatski učitati prvu scenu kada se izgrađena igra pokrene.

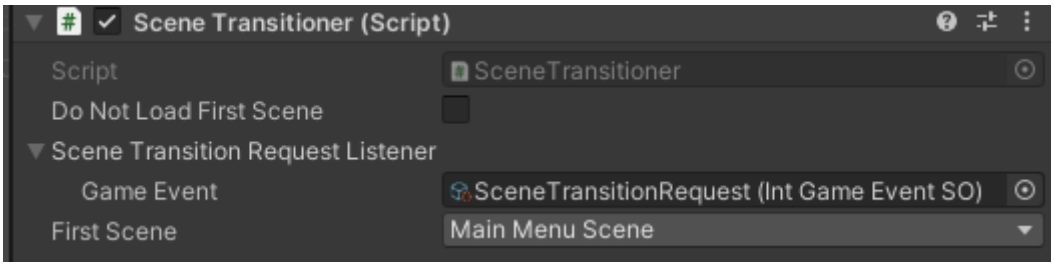


Slika 60. Scene u izgradnji

Kada se objekt igre učita, funkcijom OnEnable se dodaje slušač na ScriptableObject događaj (Slika 61) koji je postavljen kroz inspektor (Slika 62). Funkcijom OnDisable se briše slušač. Uz to, scenu je potrebno označiti kako aktivnu kada se učita. To izvršava funkcija MakeSceneActive koja se poziva kada se dozove sceneLoaded događaj SceneManagera (Unity-ev upravitelj scenama), to jest kada je nova scena učitana.

```
2 references
[SerializeField] private GameEventListeners.CodedGameEventListener<int> _sceneTransitionRequestListener;

0 references
void OnEnable() {
    SceneManager.sceneLoaded += MakeSceneActive;
    _sceneTransitionRequestListener?.Enable(TransitionTo);
}
0 references
void OnDisable() {
    SceneManager.sceneLoaded -= MakeSceneActive;
    _sceneTransitionRequestListener?.Disable();
}
```

Slika 61. Postavljanje scene aktivnom

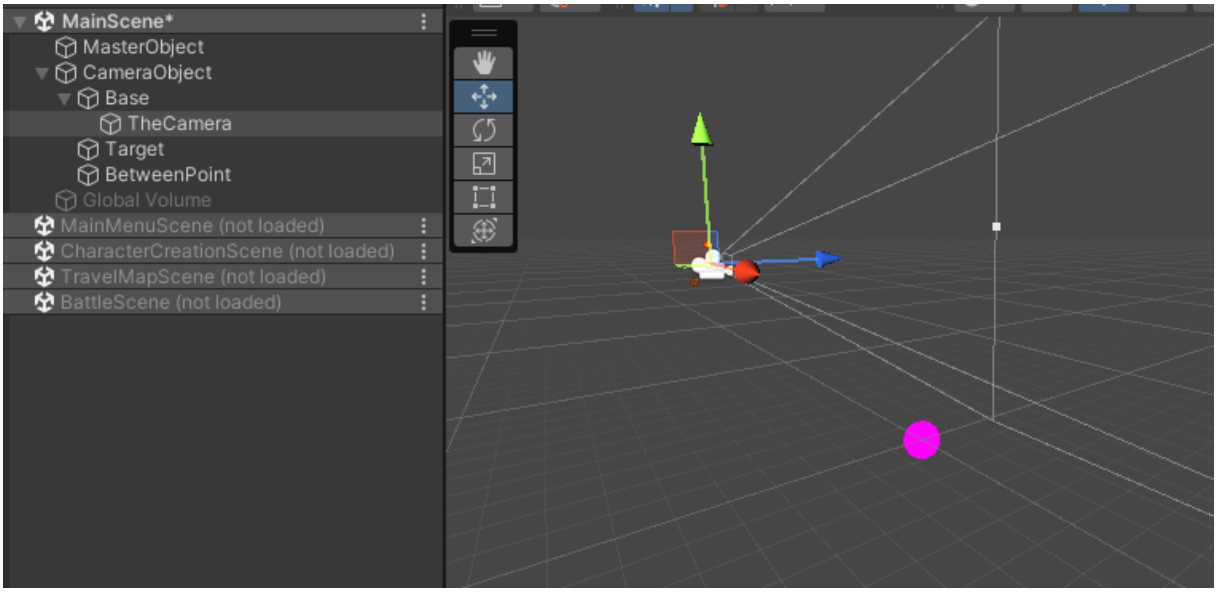


Slika 62. SceneTransitioner kao komponenta u inspektoru

Također, kroz inspektor se može postaviti opcija da se ne učitava prva scene (MainScene je već učitana, a kako se ova skripta nalazi jedino na objektu igre te scene, ova opcija neće utjecati na nju) i scena koja će biti učitana na početku igre.

3.4.2. Kamera

Prisjetimo se da je kamera u stvari komponenta Camera koja je zakačena na objekt igre. Možemo na nju gledati kao na leću kamere i njen stalak možemo složiti po želji. Kamera se sastoji od baze, mete i točke između baze i mete (Slika 63).



Slika 63. Kamera i njen stalak

Radi demonstracije, bazi je dodan mrežni model kako bismo ju mogli vidjeti u prostoru. Baza kamere ima podobjekt TheCamera koji ima Camera komponentu. Na taj način, kada se Base objekt pomiče u prostoru, TheCamera se pomiče s njime. Time je ostvareno horizontalno (x, z ravnina) pomicanje kamere.

Vertikalno pomicanje kamere (približavanje i udaljavanje kamere) tako da se vrijednost pozicije na y osi komponente Transform TheCamera objekta igre smanjuje, to jest povećava.

Rotacija kamere je implementirana tako što se izvodi rotacija Base objekta (u Euleru – po rotacija y-osi).

To je sve implementirano u skripti CameraControllerV2 koja je prevelika i nedovoljno bitna da bismo ju detaljnije prolazili. Ona koristi Unity-ev InputSystem za unos korisnika.

Korisnik u stvari pomiče Target objekt, za kojim se pomoću metode Vector3.SmoothDamp pomiče BetweenPoint objekt kojega onda Base prati. Time je dodana određena inercija pomacima kamere. Ponašanje kamere se može izmijeniti korištenjem ScriptableObject-a CameraParametersSO koji definira sve vrijednosti kamere i njene mogućnosti.

Kada je igra pokrenuta, kamera cijelo vrijeme „gleda“ na BetweenPoint objekt što omogućuje lagana naginjanja kamere.

3.5. MainMenuScene

Vrlo jednostavna scena kod koje se možemo dotaknuti interakcije s ranije spomenutom SceneTransitioner skriptom. Dok SceneTransitioner skripta ima mogućnost tranzicija scena, netko joj takav zahtjev mora dati. To rade skripte koje proširuju klasu AnySceneTransitionRequestor (Slika 64).

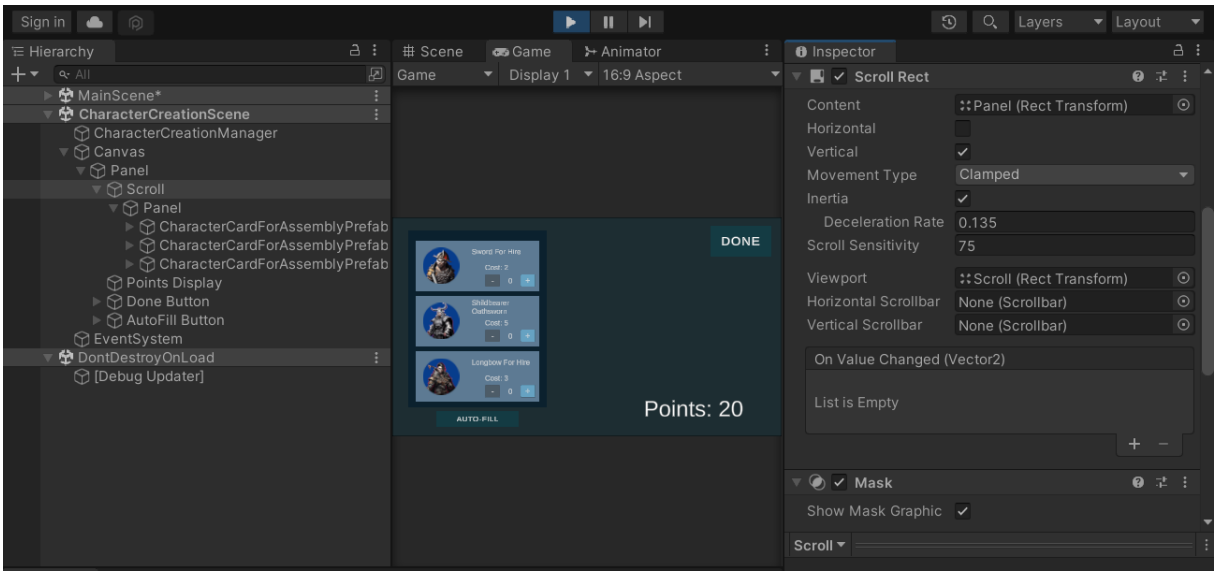
```
5 references
public abstract class AnySceneTransitionRequestor : MonoBehaviour {
    1 reference
    [SerializeField] protected GameEvents.AnyGameEventWithParameterSO<int> _sceneTransitionRequest;

    11 references
    protected void RequestSceneTransition(SceneTransitioner.ScenesInBuild toScene) {
        _sceneTransitionRequest?.Raise((int)toScene);
    }
}
```

Slika 64. AnySceneTransitionRequestor

Metodom RequestSceneTransition se podiže događaj koji sluša SceneTransitioner skripta.

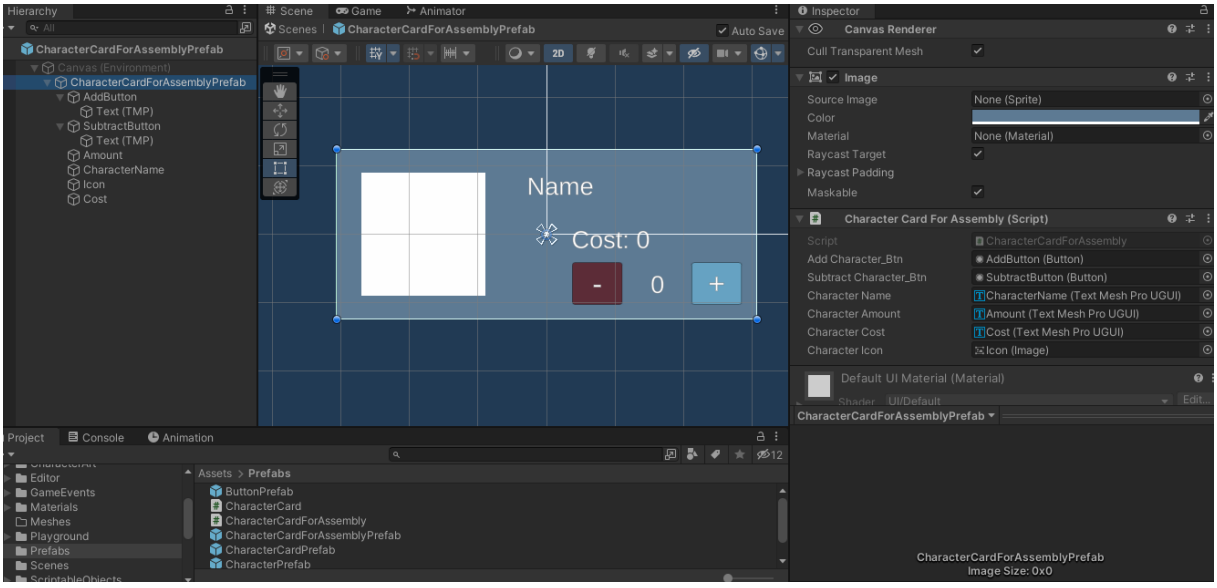
3.6. CharacterCreationScene

Sadržaj ove scene je isključivo u obliku korisničkog sučelja (Slika 65).



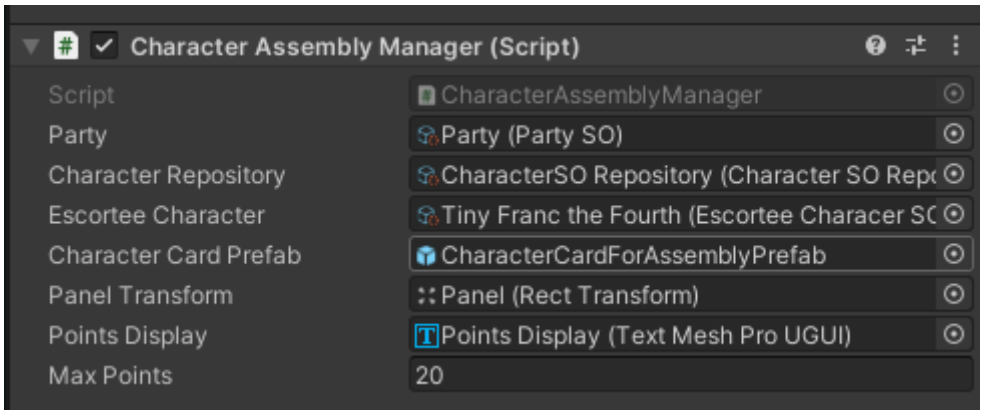
Slika 65. CharacterAssebmly scena u uređivaču

Objekt CharacterCreationManager ima referencu na ScriptableObject CharacterSORespositorySO iz kojeg dohvaća listu AllyCharacterSO-a i metodom Instantiate stvara kopiju unaprijed definiranog objekta CharacterCardForAssemblyPrefab (Slika 66).



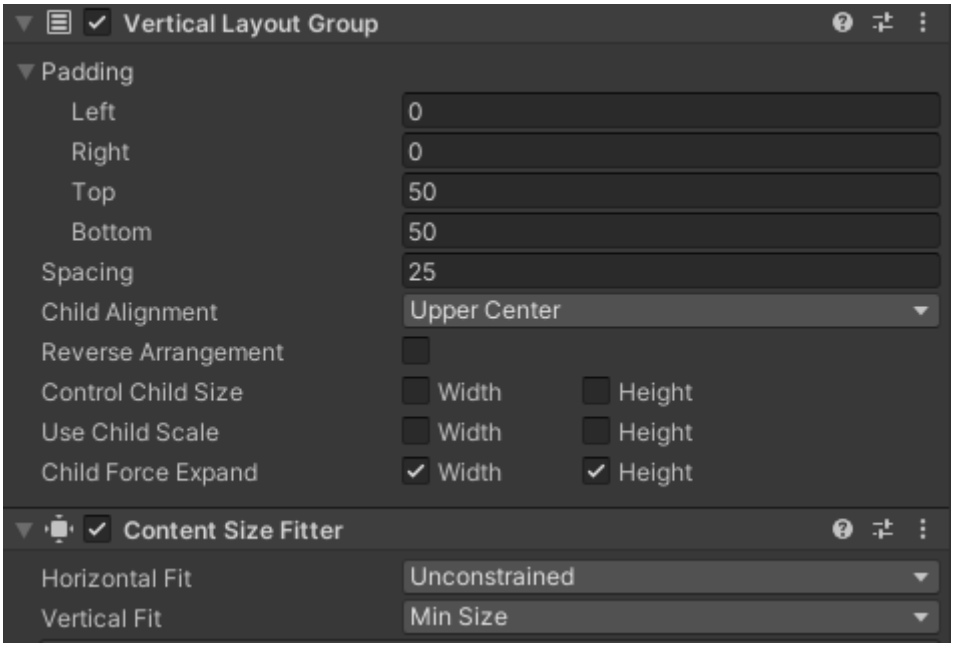
Slika 66. CharacterCardForAssemblyPrefab

Radi se o unaprijed stvorenom objektu igre koji se može pohraniti kao resurs zajedno sa svojim komponentama, podobjektima i njihovim komponentama. Ovi objekti se mogu referencirati direktno preko skripte (Slika 67).



Slika 67. CharacterCardForAssemblyPrefab referenca na polju skripte

Vratimo li se unazad (Slika 66), pomična lista je ostvarena pomoću komponente ScrollRect kojom se specificira Transform komponenta sadržaja i Transform komponenta vidnog područja. Komponentom Mask se maskira sadržaj koji nije unutar vidnog područja. Podobjekt Panel objekta Scroll ima komponente VerticalLayoutGroup i ContentSizeFitter (Slika 68).

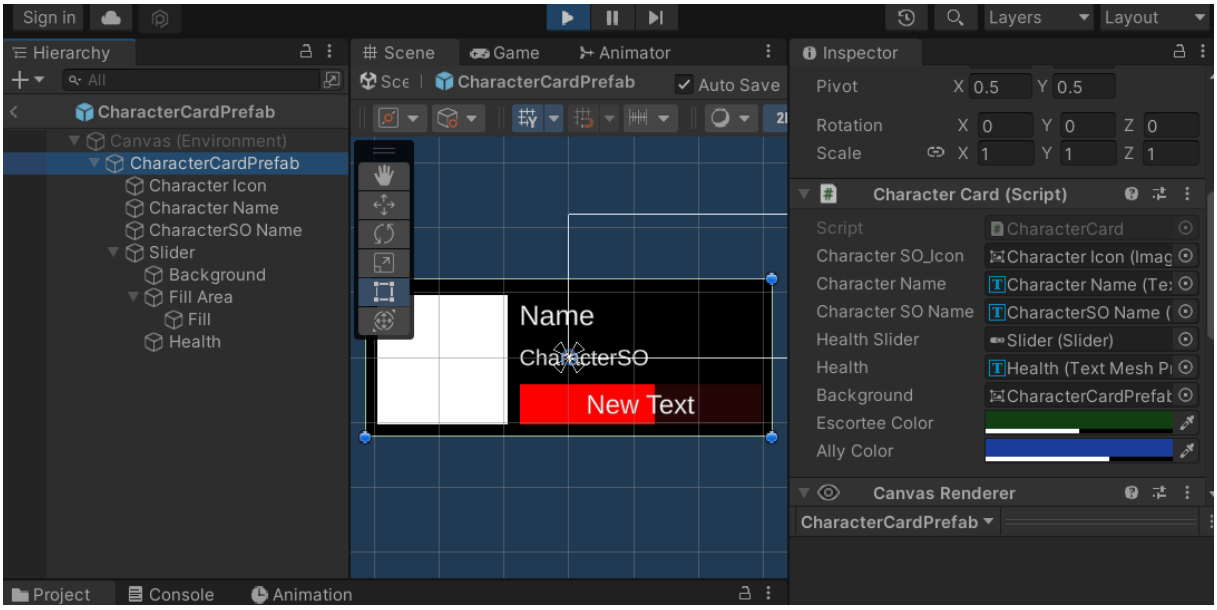


Slika 68. VerticalLayoutGroup i ContentSizeFitter

VerticalLayoutGroup autoamtski pozicionira podobjekte objekta jedan iznad drugog, a ContentSizeFitter povećava dimenzije objekta prema njegovim podobjektima.

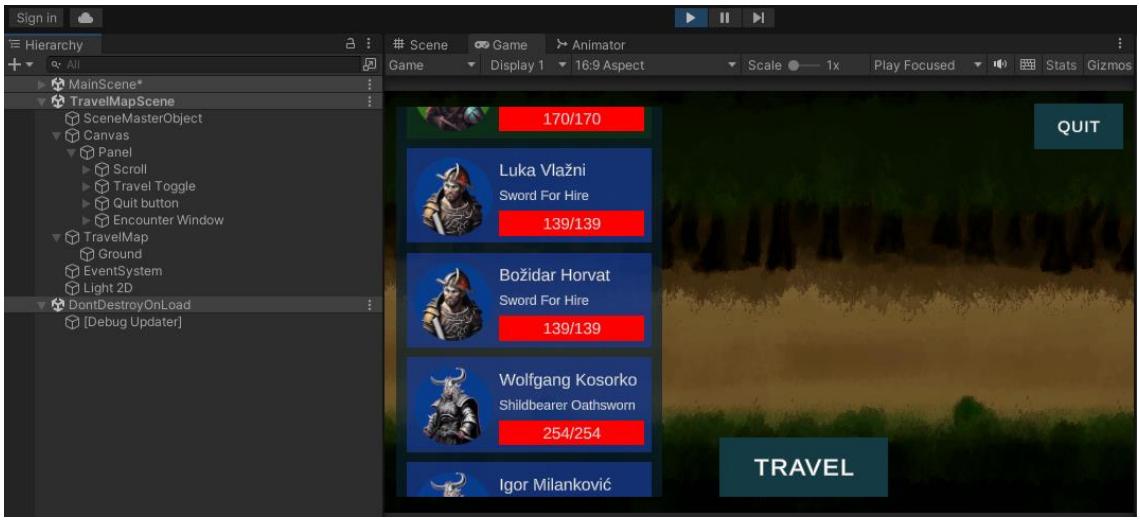
3.7. TravelMapScene

Scena putovanja koristi isti sustav pomične liste kao scena sastavljanja družine, samo što koristi drug, vrlo sličan, unaprijed stvoren objekt igre koji umjesto botuna za povećanje i smanjivanje količine karaktera tipa, prikazuje životne bodove karaktera (Slika 69).



Slika 69. CharacterCardPrefab

Kao i ostale scene i ova ima glavan objekt koji može signalizirati glavnom objektu glavne scene na koju scenu i kada je na nju potrebno prijeći. Drugi bitan objekt je TravelMap koji ima Ground podobjekt, te Light2D (Slika 70) koje stvara izvor svjetlosti i imitira sunčevu svijetlost.



Slika 70. Objekti TravelMap scene

3.7.1. TravelMap objekt

Glavna komponenta ovog objekta igre je komponenta TravelMap (Slika 71).

```
2 references
public class TravelMap : AnyMap {
    2 references
    private TravelableElement[] _mapElements;


    0 references
    void Start() {
        _mapElements = GetComponentsInChildren<TravelableElement>();
    }

    0 references
    void Update() {

    }

    2 references
    public void Travel(float travelSpeed) {
        foreach(var mapElement in _mapElements) {
            mapElement.Travel(travelSpeed);
        }
    }
```

Slika 71. TravleMap skripta

Skripta u Start funkciju prikuplja sve komponente tipa TravelableElement koje se nalaze na njegovim podobjektima. **Napomena**: metoda GetComponentsInChildren pretražuje i trenutni objekt igre i njegove podobjekte.

Također, ima i javnu funkciju kojom pokreće sve TrevelableElement-e.

TrevelableElement, preciznije SquareMeshTravelableElement koji je ovdje korišten je komponenta na objektu Ground. Kada su objekti predaleko od centra scene (pozicije Vector3.zero) radi brojeva s pomičnim zarezom može doći do netočnih kalkulacija. To je nešto što je potrebno imati na umu, pogotovo kada se radi o „beskonačnom“ kretanju u jednu stranu. Dok se takav problem vjerojatno ne bi mogao dogoditi u ovoj igri, svejedno je dobro imati na umu takvu mogućnost. Jedno od rješenja tog problema je koristiti pomicanje kontinuiranih tekstura. SquareMeshTravelableElement provjerava svaki kadar treba li pomicati teksturu. Ako mu je zadana brzina putovanja, ono će pomicati teksturu, a s obzirom na to da je tekstura

kontinuirana, samo će se ponavljati. U GIMP-u sam stvorio takvu teksturu gdje se ne vide prijelazi.

```
0 references
void Update() {
    if(ShouldTravel()) _mRenderer.material.mainTextureOffset += Vector2.right * Time.deltaTime * TravelSpeed;
}

1 reference
public override void Travel(float travelSpeed) {
    SetTravelSpeed(travelSpeed);
}
```

Slika 72. Pomicanje teksture

Pomak se množi s Time.deltaTime (Slika 72). Ono daje vrijeme proteklo od zadnjeg kadra. Time se osigurava podjednako ponašanje kretnje ako se igra izvršava u velikom broju kadrova po sekunda i ako se izvršava u malom broj kadrova po sekundi. Ako je broj kadrova po sekundi veći, Time.deltaTime će izmjeriti manje proteklo vrijeme, do kako je manji, veće proteklo vrijeme, to jest množiti većim brojem što će rezultirati većim pomakom teksture.

3.7.2. SceneMasterObject

Objekt koji, kao što je već rečeno, javlja potrebu za prijelazom na drugu scenu (konkretnije, na scenu bitke ili na scenu glavnog izbornika), generira pomičnu listu karaktera, obrađuje događaj započinjanja putovanja pritiskom na botun „TRAVEL“ (postoji logika oko toga koju nije prikladno obrađivati na ranije spomenutim objektima, stoga je dio odgovornosti na ovom objektu) i podiže događaj koji odašilje CameraParametersSO parametre kamere. Spomenutim funkcionalnostima upravlja skripta TravelMapManager.

U Start funkciji skripta pokreće korutinu koja provjerava je li došlo do sukoba (Slika 73).

```
void Start() {
    BroadcastCameraParameters();

    AssemblePartyCharacterCards();

    _encounterInitiator = StartCoroutine(CheckForEncounter());
}
```

Slika 73.Skripta TravelMapManger, Start ()

CheckForEncounter (Slika 74) je funkcija koja mora imati povratni tip IEnumerator kako bi se mogle koristiti YIELD izjave i time koristiti asinkrono izvršavanje.

```
private IEnumerator CheckForEncounter() {
    while (true) {
        yield return new WaitForSeconds(1f);
        if (_traveling) {
            var randomNumber = Random.Range(0f, 1f);
            if (randomNumber <= _encounterChance) {
                ToggleTravel();
                _canvasManager.CloseAllAndOpenCanvasGroupAndItsChildren(_encounterWindow);
            }
        }
    }
}
```

Slika 74. CheckForEncounter korutina

Korutina se vrti u beskonačnoj petlji i koristi metodu WaitForSeconds kojom pauzira svoje izvršavanje na određeno vrijeme (u ovom slučaju jednu sekundu). Ako je družina nakon te sekunde u stanju putovanja, generira se nasumičan broj između 0,0 i 1,0 (možemo na to gledati kao 0% i 100%) i time se provjerava je li došlo do sukoba. Ako je, poziva funkciju ToggleTravel kojom se započinje ili zaustavlja putovanje. S obzirom na to da je družina morala putovati kako bi došlo do sukoba, Putovanje će se u ovom slučaju zaustaviti, te će se otvoriti prozor koji javlja poruku da je došlo do sukoba. Na tom prozoru je botun koji poziva metodu koja će generirati sukob.

U segmentu ispod (Slika 75) se mogu vidjeti OnEnable i OnDisable funkcije.

```
void OnEnable() {
    _encoiunterManager.event_battleMapParametersModified.AddListener(TransitionToBattleScene);
}

0 references
void OnDisable() {
    _encoiunterManager.event_battleMapParametersModified.RemoveListener(TransitionToBattleScene);
    StopCoroutine(_encounterInitiator);
}
```

Slika 75. TravelMapManager, slušač događaja modifikacije parametara mape borbe

U funkciji OnDisable se zaustavlja korutina. Ona se neće istog trena prestati izvršavati, već će stati s izvršavanjem na slijedećoj yield return liniji. U ovom slučaju to znači da će čekati jednu sekundu i onda stati i time prekinuti beskonačnu petlju. U ovoj funkciji se također miče slušatelj događaja koji podiže skripta koja će biti slijedeća objašnjena.

Radi se o skripti EncounterManager kojoj je zadaća generirati sukob. Ranije spomenuti botun pokreće slijedeću metodu (Slika 76).

```
public void ModifyBattleMapParameters() {
    GenerateEncounter();

    SetDeploymentGap();
    SetBattleMapWidth();
    SetBattleMapHeight();

    _party.IncreaseTargetThreatLevel();

    InvokeBattleMapParametersModifiedEvent();
}
```

Slika 76. ModifyBattleMapParameters procedura

Prije svega, metoda generira sukob pozivom metode GenerateEncounter (Slika 77).

```
private void GenerateEncounter() {
    var enemyFactions = _characterSORepository.GetAllEnemyFactions();
    EnemyFactionSO enemyFaction = GetRandomEnemyFaction(enemyFactions);

    var enemySOs = _characterSORepository.GetEnemiesFromFaction(enemyFaction);
    int currentThreatLevel = 0;
    List<EnemyCharacterSO> selectedEnemySOs = new();
    do {
        EnemyCharacterSO selectedEnemySO = GetWeightedRandomEnemy(enemySOs);
        selectedEnemySOs.Add(selectedEnemySO);
        currentThreatLevel += selectedEnemySO.CharacterThreatLevel;
    } while (currentThreatLevel < _party.TargetThreatLevel);

    _enemies.CreateEnemyCollectionFromEnemySOs(selectedEnemySOs);
}
```

Slika 77. GenerateEncounter

Prvo je potrebno odrediti iz koje frakcije će neprijatelji biti. Funkcija GetRandomEnemyFaction (Slika 78) koristi ponderiranu generaciju nasumičnih brojeva. Svaka frakcija ima polje FactionFrequency gdje veća vrijednost predstavlja veću šansu za biti odabrana. Frakcija bandita ima FactionFrequency postavljen na 3, dok dezerteri imaju FactionFrequency postavljen na 1.

```
private EnemyFactionSO GetRandomEnemyFaction(List<EnemyFactionSO> enemyFactions) {
    if (enemyFactions ≠ null && enemyFactions.Count > 0) {
        int frequencySum = enemyFactions.Select(x ⇒ x.FactionFrequency).Sum();
        int upTo = Random.Range(0, frequencySum + 1);
        int counter = 0;
        foreach (var faction in enemyFactions) {
            if (faction ≠ null) {
                if (counter + faction.FactionFrequency ≥ upTo)
                    return faction;
                else
                    counter += faction.FactionFrequency;
            }
        }
    }
    return null;
}
```

Slika 78. GetRandomEnemyFaction

Slijedeći korak je dohvatiti EnemySO-eve iz repozitorija koji pripadaju odabranoj frakciji. Potom se ulazi u do-while petlju gdje se koristi metoda GetWeightedRandomEnemy (Slika 79). Svaki EnemyCharacterSO ima polje _rarity koje definira koliko je rijetko da se karakter pojavi. Što je broj veći, manje su šanse. Metoda prvo kreira listu tuple-a i grupira EnemyCharacterSO i njegovu rijetkost radi lakšeg snalaženja (ovaj korak nije potreban, ali onda kasnije treba paziti na to da su rijetkosti izražene kao cijeli broj). Neposredno zatim, lista tuple-a se mora nasumično sortirati jer je u ovom slučaju, kada veći broj predstavlja manju šansu za biti odabran, poredak igra važnu ulogu. Karakteri koji imaju slične rijetkosti, a koju su na početku liste imaju puno veće šanse biti odabrani u odnosu na one koji su na kraju liste. Potom se izračuna suma inverznih rijetkosti, generira nasumičan broj između 0 i te sume te odabire EnemyCharacterSO. Moguće je korigirati algoritam po želji, ali mislim da su ove vrijednosti adekvatne.

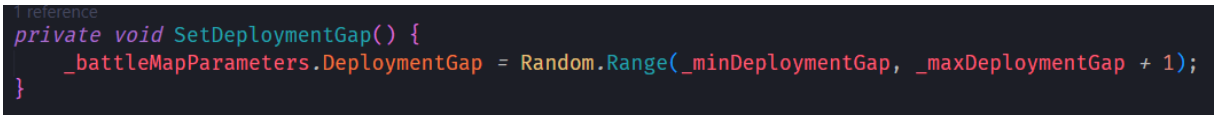
```
1 reference
private EnemyCharacterSO GetWeightedRandomEnemy(List<EnemyCharacterSO> enemies) {
    var enemiesAndRarity = enemies.Select(x => new { enemy = x, rarity = (float)x.Rarity }).ToList();
    enemiesAndRarity.Sort((a, b) => (b.rarity + Random.Range(0f, 0.1f)).CompareTo(a.rarity + Random.Range(0f, 0.1f)));
    float totalInverseRarity = 0f;
    foreach (var enemy in enemiesAndRarity) {
        totalInverseRarity += 1f / enemy.rarity;
    }
    float randomNo = Random.Range(0, totalInverseRarity);
    float cumulativeInverseRarity = 0f;

    foreach (var enemy in enemiesAndRarity) {
        cumulativeInverseRarity += 1f / enemy.rarity;

        if (randomNo <= cumulativeInverseRarity) {
            return enemy.enemy;
        }
    }
    return null;
}
```

Slika 79. GetWeightedRandomEnemy

Vratimo li se na metodu GenerateEncounter, ona dalje pohranjuje odabrani EnemyCharacterSO u listu i poveća trenutnu opasnost sukoba za razinu opasnosti odabranog EnemyCharacterSO-a. Koraci se ponavljaju dok nije pređena ciljana razina opasnosti. Nakon toga se kreiraju EnemyCharacter objekti na temelju EnemyCharacterSO-a i pohranjuju u EnemySO – kolekciju EnemyCharacter objekata. Svrha ove kolekcije je čuvanje instanci neprijatelja i saveznika (PartySO) između scena.

Time su odlučeno koji neprijatelji će dočekati družinu i sada je na redu pripremiti parametre mape na kojoj se će se izvoditi bitka.

Prvo se određuje razmak između neprijateljskog i savezničkog polja razmještanja (Slika 80). Potom se određuje širina mape (Slika 81).



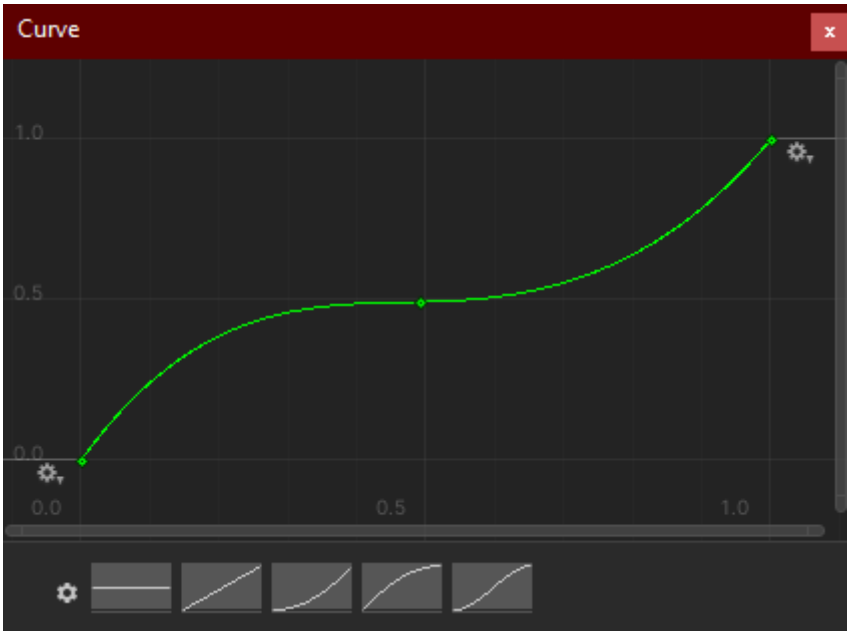
```
1 reference
private void SetDeploymentGap() {
    _battleMapParameters.DeploymentGap = Random.Range(_minDeploymentGap, _maxDeploymentGap + 1);
}
```

Slika 80. SetDeploymentGap

```
1 reference
private void SetBattleMapWidth() {
    float fMapWidth = _maxBmWidth - _minBmWidth;
    fMapWidth *= _mapWidthBias.Evaluate(Random.Range(0f, 1f));
    fMapWidth += _minBmWidth;
    int mapWidth = Mathf.RoundToInt(fMapWidth);
    _battleMapParameters.Width = mapWidth;
}
```

Slika 81. SetBattleMapWidth

_mapWidthBias je AnimationCurve. Radi se o veoma korisnom alatu kojim se može nacrtati krivulja (kroz kôd ili kroz Unity uređivač (Slika 82)) i metodom Evaluate dobiti vrijednost krivulje u trenutku t. Za t generiramo nasumičan broj. Prema ovoj krivulji, mapa će najčešće biti srednje širine, a manje je moguće da će biti vrlo široka ili vrlo uska.



Slika 82. AnumationCurve kroz uređivač

Vratimo li se na metodu ModifyBattleMapParemeters (Slika 76), slijedeći korak je određivanje visine mape. Bitno je da se svi karakteri mogu postaviti na mapu, pa ako je mapa preuska, moramo ju napraviti višom/dužom. To određuje metoda SetBattleMapHeight (Slika 83). Metoda uzima u obzir to da postoje dvije vrste karaktera s pogleda dometa; oni koji se bore na malim udaljenostima i oni koji se bore na velikim udaljenostima. Kako ne bi bilo preskućeno na mapi, dodaje se _extraHeight, a ako je širina veća od visine mape, onda se

visina postavlja na vrijednost u rasponu između širine mape i maksimalne visine mape kako ne bi došlo do jako široke i jako niske mape.

```
1 reference
private void SetBattleMapHeight() {
    int partyMinRowsForCharacterDeployment = _battleMapParameters
        .GetRequiredAmountOfRowsForCharacterDeploymentByCharacterSubgroup(_party
            .GetAmountOfCharactersByDeploymentGroup()).Sum() + 1;
    int enemiesMinRowsForCharacterDeployment = _battleMapParameters
        .GetRequiredAmountOfRowsForCharacterDeploymentByCharacterSubgroup(_enemies
            .GetAmountOfCharactersByDeploymentGroup()).Sum();

    int maxMinRows = Mathf.Max(partyMinRowsForCharacterDeployment, enemiesMinRowsForCharacterDeployment);

    int mapHeight = maxMinRows + _battleMapParameters.DeploymentGap + _extraHeight;

    if (_battleMapParameters.Width > mapHeight) {
        mapHeight = Mathf.Clamp(mapHeight, _battleMapParameters.Width, _maxBmHeight);
    }

    _battleMapParameters.Height = mapHeight;
}
```

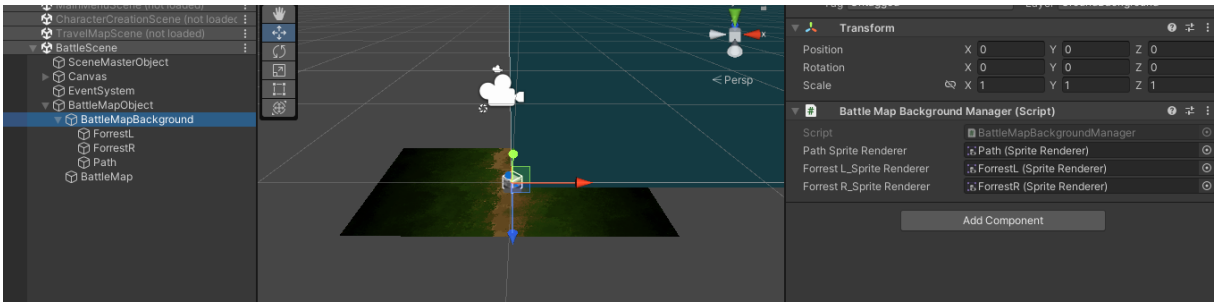
Slika 83. SetBattleMapHeight

Slijedeći korak (Slika 76) je povećati buduću ciljanu razinu opasnosti sukoba kako bi slijedeći sukob bio teži. I na kraju se odašilju signal da su parametri mape bitke postavljeni. Ako se prisjetimo, taj događaj sluša TravelMapManager skripta i tada traži tranziciju na scenu bitke.

3.8. BattleScene

Na sceni bitke se izvodi borba. Kao i na svakoj sceni, postoji glavni objekt scene koji može zatražiti tranziciju na druge scene (u ovom slučaju to su scene putovanja i glavnog izbornika). Osim toga upravlja sustavom korisničkog unosa, pokreće proces generiranja mape prema parametrima BattleMapParametersSO-a, postavlja karaktere na mapu i stvara njihove modele. To se sve izvodi preko skripte BattleMapManager. Ona dalje koristi druge klase kako bi izvršila potrebne radnje, to jest nastojano je očuvati princip jedne odgovornosti. Na objektu postoji i skripta BattleMapGameplayManager koja upravlja tijekom igre.

Postoji i objekt BattleMapObject koji već na sebi ima objekte koji predstavljaju teren (Slika 84). To objekti se moraju pravilno rasporediti to, jest moraju se raširiti kako bi odgovarali dimenzijama mape. Time upravlja skripta BatleMapBackgroundManager i vrlo je jednostavna, te ju nećemo prolaziti.



Slika 84. BattleMapBackground

3.8.1. Problem s novim sustavom unosa

Ovdje je bitno napomenuti kako novi sustav unosa ne podržava IMGUI i UIElement-e. [19] U međuvremenu je taj problem riješen korištenjem UI Toolkit-a. [10] Rješenje koje je primijenjeno u radu je preuzeto sa StackOverflow-a koje je kao odgovor dao korisnik Lowelltech. [20]

Problem je taj da klikovi na UI elemente prolaze kroz njih i pokreću događaje objekata igre. Korišteno rješenje je MonoBehavoiur skript koju je potrebno zakvačiti na objekt platna i koja pruža funkciju pomoću koje metode koje upravljaju događajima sustava unosa mogu provjeriti je li kursor iznad objekta korisničkog sučelja i obraditi događaj u skladu s time.

3.8.2. Generiranje mape

Proces generiranja mape je razdijeljen na nekoliko klasa. Kako bi mapa bila generirana potrebno je odrediti koje heksagone je potrebno kreirati (s kojim koordinatama), kako ih rasporediti u hijerarhiji scene, kako ih rasporediti u svijetu scene i postaviti vrijednosti Hexa komponente.

3.8.2.1. BattleMapLoader

Ovo je klasa koja služi za generiranje mreže heksagona (Slika 85). Prvo dohvaća komponentu BattleMap koja služio kao objedinjenje, to jest kolekcija heksagona koji će biti generirani i postavlja joj osnovne vrijednosti. Potom koristi kvadratnu metodu stvaranja heksagona, to jest klasu RectagleHexSpawnMehod. Nakon što su stvoreni heksagoni, oni nisu na pravim pozicijama, te ih je potrebno razvrstati. Trenutno su samo stvoreni i dodijeljene su im odgovarajuće koordinate (o koordinatama detaljnije u poglavlju Heksagoni).

```
public class BattleMapLoader {
    1 reference
    public BattleMap GenerateMap(GameObject hexPrefab, Transform parent, int mapWidth, int mapHeight, int deploymentGap) {
        var battleMap = parent.gameObject.GetComponent<BattleMap>();

        battleMap.width = mapWidth;
        battleMap.height = mapHeight;
        battleMap.deploymentGap = deploymentGap;

        var spawnMethod = new RectangleHexSpawnMethod(hexPrefab, parent);
        var hexesList = spawnMethod.SpawnHexes(mapWidth, mapHeight);

        var hexDisplacer = new DirectHexDisplacer();
        hexDisplacer.DisplaceHexes(hexesList);

        battleMap.RegisterNewHexList(hexesList);

        return battleMap;
    }
}
```

Slika 85. BattleMapLoader

3.8.2.2. RectangleSpawnMethod

Klasa RectagleSpawnMethod je prikazana ispod (Slika 86). Klasa definitivno ima manjkavosti. S obzirom na to da heksagoni koriste aksijalne koordinate, proces određivanja koordinata bi trebao biti bolji. Ideja je da metoda SpawnHexes kreće od sredine mape (po visini) i paralelno kreira heksagone za red ispod i red iznad s time da svaka dva red radi pomak. Ovo priliči sustav pomaknutih koordinata, ali je svrsishodno.

```
1 reference
public List<Hexa> SpawnHexes(int width, int height) {
    List<Hexa> hexesList = new();
    int qStart = 1;
    int rowWidth = width * 2;

    for (int r = 0; r <= height; r++) {
        if (r%2 == 0) qStart--;

        for (int j = 0; j <= rowWidth; j++) {
            hexesList.Add(CreateHex(-width + qStart + j, r));
            if (r != 0) hexesList.Add(CreateHex(-width + qStart + j + r, -r));
        }
    }
    return hexesList;
}

2 references
private Hexa CreateHex(int q, int r) {
    Hexa goHexaComp;

    if (parent != null) goHexaComp = GameObject.Instantiate(hexPrefab, parent, false).GetComponent<Hexa>();
    else goHexaComp = GameObject.Instantiate(hexPrefab).GetComponent<Hexa>();

    goHexaComp.gameObject.layer = goHexaComp.transform.parent.gameObject.layer;

    goHexaComp.SetCoords(q, r);

    return goHexaComp;
}
```

Slika 86. SpawnHexes i CreateHexes metode RectangleSpwanMethod klase

Funkcija CreateHex kreira heksagon, postavlja ga kao podobjekt objekta s BattleMap komponentom, kako ne bi došlo do isjecanja kod prikazivanja, postavlja ih na prikladan sloj prikazivanja i postavlja koordinate.

3.8.2.3. HexDisplacer

Klasa HexDisplacer (Slika 87) dalje postavlja heksagone na odgovarajuće pozicije. Ovo isto nije najadekvatniji pristup, te bi se dalo pametnije napraviti.

```
public class DirectHexDisplacer : IHexDisplacer<Hexa> {
    1 reference
    public void DisplaceHexes(List<Hexa> hexesList) {
        if (hexesList != null && hexesList.Count > 0) {
            var hexDimensions = hexesList[0].GetComponent<HexRenderer>().GetHexDimensions();
            float horizontalHalf = hexDimensions.horizontal / 2f;

            foreach (var hex in hexesList) {
                float hShift = hex.coords.r * horizontalHalf;
                hex.transform.localPosition =
                    new Vector3(hex.coords.q * hexDimensions.horizontal + hShift, hex.coords.r * hexDimensions.vertical, 0f);
            }
        }
    }
}
```

Slika 87. DirectHexDisplacer

Nakon što je mreža heksagona postavljena BattleMapManager odašilje parametre kamere i javlja BattleMapBackgroundManager skripti da prilagodi pozdainu. Nakon toga započinje proces instanciranja objekata karaktera na mapu (Slika 88).

```
void Start() {
    _battleMapGameplayManager = Get
    _hexDimensions = _hexPrefab.Get

    GenerateMap();
    SetCameraBounds();
    BroadcastCameraBounds();
    AdjustBackground();

    InitiateDeploymentPhase();
}
```

Slika 88. BattleMapManager Start funkcija

3.8.3. Instanciranje objekata karaktera na mapi

Ispod je prikazan proces instanciranja objekata karaktera na mapi (Slika 89).

```
private void InitiateDeploymentPhase() {
    field1 = _map.GetDeploymentField(BattleMap.field.field1);
    field2 = _map.GetDeploymentField(BattleMap.field.field2);

    DeployCharactersOnField(field1, _party);
    DeployCharactersOnField(field2, _enemies);

    InitiateDeploymentEditingPhase();
}
```

Slika 89. InitiateDeploymentPhase metoda

Postoje dva polja mape. Polja su u stvari zone u kojima će se objekti karaktera instancirati. Savezničko polje je field1, a polje neprijatelja je field2. Savezničko polje započinje od donjeg dijela mape i proteže se do vrijednsoti polja deploymentGap BattleMap komponente. Obrnuto vrijedi za polje neprijatelja.

Metoda DeployCharactersOnField je prikazana ispod (Slika 90).

```
private void DeployCharactersOnField<T>(List<HexCoords> field, CharacterCollectionSO<T> characterCollection)
    where T : Character {
        Character[][] charactersBySubgroup = characterCollection.GetCharactersByDeploymentGroup();
        if(field != null && field.Count > 0) {
            bool negativeField = field[0].r < 0;
            int firstR = negativeField ? field.Max(x => x.r) : field.Min(x => x.r);
            int rowWidth = _battleMapParameters.ActualWidth;

            int[] requiredRowsForSubgroups = _battleMapParameters
                .GetRequiredAmountOfRowsForCharacterDeploymentByCharacterSubgroup(
                    characterCollection.GetAmountOfCharactersByDeploymentGroup());
            Character[][] charactersByRow = new Character[requiredRowsForSubgroups.Sum()][];
            for(int i = 0, k = 0; i < charactersBySubgroup.Length; i++) {
                for (int j = 0; j < requiredRowsForSubgroups[i]; j++, k++) {
                    charactersByRow[k] = PopulateRow(charactersBySubgroup[i].Skip(j * rowWidth)
                                                .Take(rowWidth).ToArray(), rowWidth);
                }
            }

            InstantiateCharactersOnMap(charactersByRow, field, negativeField, firstR);
        }
}
```

Slika 90. DeployCharactersOnField

Prvo se kreira neravno dvodimenzijalno polje tipa Character koje grupira karaktere prema dometu njihovih oružja. Potom se analizira polje na kojemu se objekti karaktera moraju instancirati i izračuna broj redova za svaku podgrupu karaktera. Ako je previše karaktera jedne podgrupe da bi svi stali u jedan red, onda se prelazi u drugi red i kreće ispočetka. Ovaj sustav

je implementiran i kod računanja potrebnih dimenzija mape, tako da se neće dogoditi situacija gdje se ne mogu svi objekti karaktera instancirati. Varijabla charactersByRow predstavlja matricu karaktera kako bi izgledali na kvadratnoj mreži i puni se red po red iz varijable charactersBySubgroup pomoću metode PopulateRow (Slika 91).

```
1 reference
private Character[] PopulateRow(Character[] characterSet, int rowWidth) {
    int arrMid = rowWidth / 2;
    Character[] charRow = new Character[rowWidth];
    for (int i = 0, j = 0; i < characterSet.Length; i++) {
        if(characterSet[i] != null) {
            charRow[j + arrMid] = characterSet[i];
            if(i % 2 == 0) j++;
            j *= -1;
        }
    }
    return charRow;
}
```

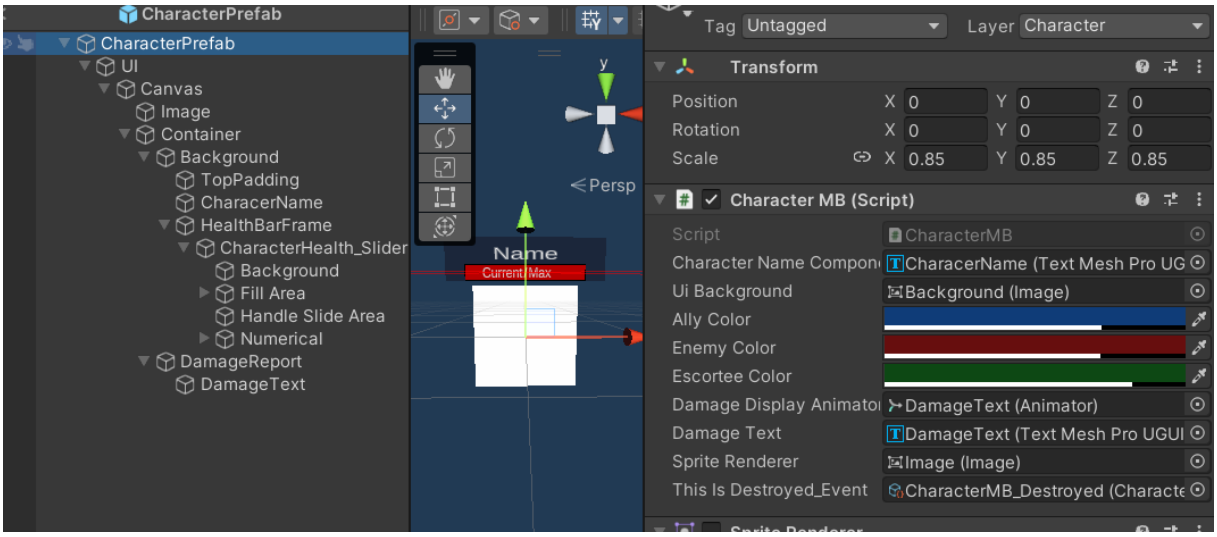
Slika 91. PopulateRow metoda

Ova metoda bi se također trebala unaprijediti na način da bude više u skladu s aksijalnim koordinatama. Ovdje se koriste pomaci.

Metoda InstantiateCharactersOnMap (Slika 92) konačno kreira instance objekata karaktera, to jest kreira unaprijed stvoren objekt igre _characterPrefab (Slika 93) i postavlja njegove vrijednosti. Ovdje se susrećemo sa skriptom Hexa koju ima svaki heksagon objekt.

```
1 reference
private void InstantiateCharactersOnMap(Character[][] charactersByRow, List<HexCoords> field, bool negativeField, int firstR) {
    int row = 0;
    int j = 0;
    for (int i = 0; i < charactersByRow.Length; i++) {
        if(negativeField) row = firstR - i;
        else row = firstR + i;
        int beginQ = field.Where(x => x.r == row).Min(x => x.q);
        for (j = 0; j < charactersByRow[i].Length; j++, beginQ++) {
            if(charactersByRow[i][j] != null) {
                Character c = charactersByRow[i][j];
                Hexa hex = _map.hexCollection.GetHex(beginQ, row);

                var characterObjecInstance = Instantiate(_characterPrefab);
                var charMB = characterObjecInstance.GetComponent<CharacterMB>();
                charMB.SetCharacter(c);
                charMB.GetComponentInChildren<CharacterHealthBarManager>()
                    .InitializeHealthBarManager(c.HealthChanged, c.MaxHealth, c.Health);

                hex.SetOccupant(charMB);
                charMB.UpdateHex(hex);
            }
        }
    }
}
```

Slika 92. InstantiateCharactersOnMap metoda



Slika 93.CharacterPrefab u uređivaču

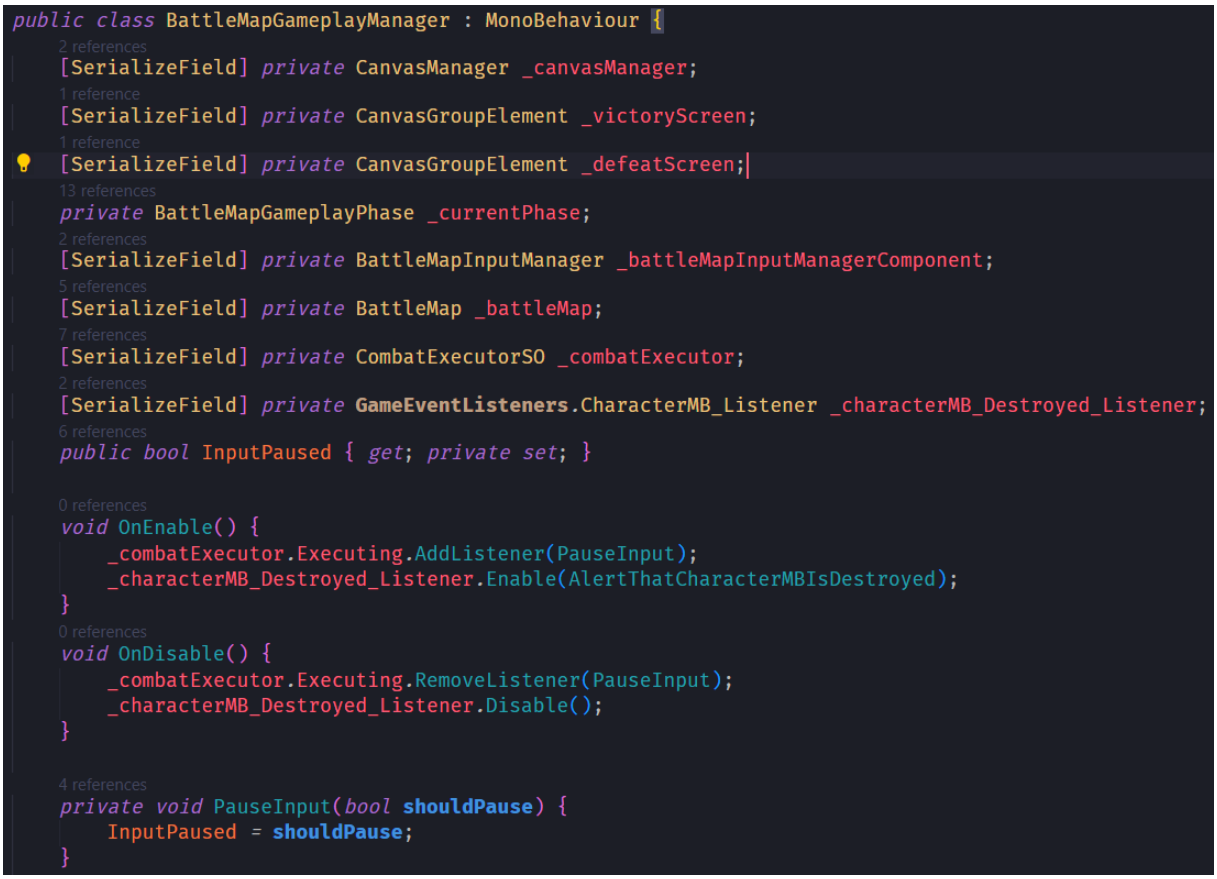
Nakon što su objekti karaktera instancirani, odgovornost se predaje skripti BattleMapGameplayManager metodom InitiateDeploymentEditingPhase (Slika 94).

```
1 reference
private void InitiateDeploymentEditingPhase() {
    var editingField = field1;
    _battleMapGameplayManager.StartDeploymentPhase();

}
```

Slika 94. InitiateDeploymentEditingPhase

3.9. Životni ciklus bitke

Upravljanje bitkom je odgovornost skripte BattleMapGameplayMaanger i njena polja i svojstvo su prikazani ispod (Slika 95).



Slika 95.BattleMapGameplayManager polja i svojstvo

CanvasManager je skripta koja upravlja elementima korisničkog sučelja. Elemente korisničkog sučelja koji imaju skriptu CanvasGroupElement može otvarati i zatvarati. Ovdje mu je uloga otvoriti prozor pobjede ili poraza kada bitka završi.

BattleMapInputManager je skripta koja podiže događaje kada igrač komunicira s heksagonima – detektira klik na heksagon objekt.

CombatExecutorSO je ScriptableObject koji procesira napade karaktera. Ima javni događaj koji označava početak i kraj izvođenja kalkulacija (dok se one izvode potrebno je onemogućiti unos korisnika) na koji se BattleMapGameplayManager pretplaćuje.

CharacterMB_Listener je slušatelj događaja uništenja objekta igre koji ima CharacterMB komponentu, točnije kada karakter ostane bez životnih bodova, njegova fizička reprezentacija (CharacterMB) će biti uništena zvanjem metode Destroy, te će tada biti oglašen ovaj događaj kako bi se kolekcije karaktera mogle ažurirati, to jest da se heksagon na kojemu više nema karaktera jer je ostao bez životnih bodova ne smatra kao da i dalje ima karaktera. Nadalje, iako je objekt igre uništen, on i dalje postoji u drugim objektima, pa ga je potrebno dereferencirati kako bi ga C#-ov sakupljač smeća pokupio i time oslobodio memoriju.

InputPaused je svojstvo s javnim nabavljačem i privatnim postavljačem kojim se može provjeriti prihvaća li sustav trenutno korisničke unose.

BattleMapGameplayManager skripta je strukturirana kao stroj stanja (Slika 96).

```
4 references
private void TransitionToPhase(BattleMapGameplayPhase phase) {
    _currentPhase?.EndPhase();

    _currentPhase = phase;

    _currentPhase?.StartPhase();
}
```

Slika 96. BattleMapManager stroj stanje TransitionToPhase metoda

3.9.1. BattleMapGameplayPhase

Apstraktna klasa koja predstavlja tip stanja BattleMapGameplayManager stroja stanja i prikazana je ispod (Slika 97).

```
public abstract class BattleMapGameplayPhase {
    15 references
    protected List<Hexa> _affectedHexes;
    10 references
    protected BattleMapGameplayManager _battleMapGameplayManager;

    3 references
    public BattleMapGameplayPhase(BattleMapGameplayManager battleMapGameplayManger, List<Hexa> affectedHexes) {
        _affectedHexes = affectedHexes;
        _battleMapGameplayManager = battleMapGameplayManger;
    }

    1 reference
    public abstract void StartPhase();
    2 references
    public virtual void EndPhase() {
        UnsubscribeFromAll();
    }

    2 references
    public abstract void UnsubscribeFromAll();
    1 reference
    public abstract void HandeleCharacterMBDestroyed(CharacterMB destroyedCharacterMB);
}
```

Slika 97. BattleMapGameplayPhase

Metode StartPhase i EndPhase se pozivaju kada stroj stanja prelazi iz jednog u drugo stanje. UnsubscribeFromAll metoda osigurava da ne dođe do curenja memorije kod promjene stanja i HandleCharacterMBDestroyed upravlja događajem uništenja CharacterMB-a. Kako se

ne radi o MonoBehavoiur klasi, ne može se spojiti taj događaj kroz inspektor, već ga je potrebno ili nabaviti preko BattleMapGameplayManagera ili se jednostavno pretplatiti na novi događaj koji će podizati BattleMapGameplayManager čime je moguće uvesti i filtraciju.

Polje _affectedHexes predstavlja dio mape koji je zahvaćen trenutnom fazom. Na primjer, kod postavljanja karaktera prije početka bitke, to je samo polje saveznika.

Bitno je imati referencu na BattleMapGameplayManager skriptu jer su ponekad potrebne MonoBehavoiur funkcionalnosti. Dobra ideja za unaprjeđenje ovog sustava pretvaranje ove klase u MonoBehavoiur objekt. Time bi se omogućilo postavljanje vrijednosti kroz inspektor i moguće da ne bi bilo potrebe za referencom na BattleMapGameplayManager skriptu.

3.9.2. PlayerActionBattlePhase

Radi se o također apstraktnoj klasi koja proširuje klasu BattleMapCombatPhase funkcionalnostima vezanim uz igračeve akcije (Slika 98).

```
4 references
public abstract class PlayerActionBattlePhase : BattleMapGameplayPhase {
    5 references
    protected BattleMapInputManager _inputManager;
    27 references
    protected Hexa SelectedHex { get; set; }
    1 reference
    protected bool InputPaused => _battleMapGameplayManager.InputPaused;

    2 references
    public PlayerActionBattlePhase(BattleMapInputManager inputManager,
        BattleMapGameplayManager battleMapGameplayManager, List<Hexa> affectedHexes) :
            base(battleMapGameplayManager, affectedHexes) {
        _inputManager = inputManager;
    }

    1 reference
    protected abstract void SubscribeToInputManagerEvents();
    1 reference
    protected abstract void UnsubscribeFromInputManagerEvents();

    2 references
    public override void UnsubscribeFromAll() {
        UnsubscribeFromInputManagerEvents();
    }
```

Slika 98. PlayerActionBattlePhase

Kako ova klasa radi s korisničkim unosima, potrebna je referenca na BattleMapImputManager skriptu.

SelectedHex je svojstvo kojim se prati odabrani heksagon, ne u smislu koji je kliknut, nego kada igrač odabere karaktera i izvršava s njime akcije, heksagon na kojemu je karakter mora biti odabrani heksagon.

SubscribeToInputManagerEvents i UnsubscribeFromInputManagerEvents su apstraktne metode kojima konkretne implementacije ove klase mogu specificirati koje događaje se žele osluškivati, a time i moraju prestati osluškivati kada stanje završi.

```
1 reference
public override void StartPhase() {
    SubscribeToInputManagerEvents();
    SetAlliedHexesAsInteractable();
}
2 references
public override void EndPhase() {
    base.EndPhase();
    DisableAllHexesAndResetState();
}

1 reference
protected void DisableAllHexes() {
    foreach (var hex in _affectedHexes) {
        hex.InteractableHex.SetUninteractable();
    }
}
8 references
protected void DisableAllHexesAndResetState() {
    foreach (var hex in _affectedHexes) {
        hex.InteractableHex.SetUninteractable();
        hex.InteractableHex.SetDefaultState();
    }
}
3 references
protected void SetAlliedHexesAsInteractable() {
    foreach (var hex in GetAlliedHexes()) {
        hex.InteractableHex.SetInteractable();
    }
}
```

Slika 99. Funkciuje upravljanje stanjima heksagona za faze

Klasa pruža funkcije s kojima se upravlja stanjima heksagona kako bi se igraču moglo pokazati koje akcije može izvršiti (Slika 99). Također, potrebne su metode kojima će se

heksagon moći označiti kao odabran i kojima će se moći pratiti koji je heksagon odabran (Slika 100).

```
private List<Hexa> GetAlliedHexes() {
    return _affectedHexes.Where(x => x.HasOccupant() && x.Character is not EnemyCharacter).ToList();
}

3 references
protected void SelectHex(Hexa selectedHex) {
    SelectedHex = selectedHex;
    SelectedHex.InteractableHex.SetSelectedState();
}

5 references
protected void DeselectHex() {
    SelectedHex.InteractableHex.SetDefaultState();
    SelectedHex = null;
}
```

Slika 100. Select, Deselect i GetAll Hex

3.9.3. BattleMapDeploymentEditPhase

Radi se o fazi gdje igrač može raspodijeliti karaktere na svojem dijelu mape. Klasa proširuje PlayerMapGameplayPhase. Definira procedure koje su prikazane ispod (Slika 101). Kada je podignut događaj odabira heksagona, metoda HandeHexSelectedEvent provjerava je li heksagon već odabran. Ako nije, postavlja kliknuti heksagon kao odabrani. Ako je već odabran, ali je kliknut isti heksagon kao onaj koji je odabran, poništava odabrani heksagon. Ako ništa od navedenog nije istina, pokreće RunHexSwapProcedure. Unatoč nazivu, funkcija ne zamjenjuje heksagone, već karaktere koji su na njima.

```
2 references
private void HandleHexSelectedEvent(Hexa selectedHex) {
    if (SelectedHex is null)
        RunSelectHexProcedure(selectedHex);
    else if (SelectedHex == selectedHex)
        RunDeselectHexProcedure();
    else
        RunHexSwapProcedure(selectedHex);
}

1 reference
private void RunSelectHexProcedure(Hexa selectedHex) {
    EnableAllAffectedHexesAndSetThemToMoveableState();
    SelectHex(selectedHex);
}
1 reference
private void RunDeselectHexProcedure() {
    DeselectHex();
    DisableAllHexesAndResetState();
    SetAlliedHexesAsInteractable();
}
1 reference
private void RunHexSwapProcedure(Hexa selectedHex) {
    SwitchHexOccupants(SelectedHex, selectedHex);
    DeselectHex();
    DisableAllHexesAndResetState();
    SetAlliedHexesAsInteractable();
}

1 reference
private void EnableAllAffectedHexesAndSetThemToMoveableState() {
    foreach (var hex in _affectedHexes) {
        hex.InteractableHex.SetInteractable();
        hex.InteractableHex.SetMoveableToState();
    }
}
```

Slika 101. BattleMapDeplyomentEdit metode

Kako bi klasa mogla znati kada je neki heksagon kliknut, mora osluškivati događaj (Slika 102).

```
1 reference
protected override void SubscribeToInputManagerEvents() {
    _inputManager.hexSelected.AddListener(HandleHexSelectedEvent);
}

1 reference
protected override void UnsubscribeFromInputManagerEvents() {
    _inputManager.hexSelected.RemoveListener(HandleHexSelectedEvent);
}
```

Slika 102. Pretplata na događaj klika sustava unosa

3.9.4. BattleMapCombatPhase

Predstavlja potez igrača. Proširuje PlayerMapGameplayPhase klasu i strukturirana je kao stroj stanja (Slika 103).

```
9 references
public partial class BattleMapCombatPhase : PlayerActionBattlePhase {
    2 references
    private MoveMode _moveMode;
    2 references
    private AttackMode _attackMode;
    8 references
    private CombatMode _currentMode;
    7 references
    protected List<CharacterMB> _allyTracker;
    2 references
    protected int _remainingEnemies;

    1 reference
    public BattleMapCombatPhase(BattleMapGameplayManager battleMapGameplayManager,
            BattleMapInputManager inputManager,
            List<Hexa> affectedHexes,
            CombatExecutorSO combatExecutor)
            : base(inputManager, battleMapGameplayManager, affectedHexes) {

        _moveMode = new(this);
        _attackMode = new(this, combatExecutor);

        CollectCharacterMBs();
        TransitionToMoveMode();

        _remainingEnemies = _affectedHexes.Where(x => x.Character is EnemyCharacter && !x.Character.IsDead).Count();
    }

    1 reference
    protected virtual void CollectCharacterMBs() {
        _allyTracker = new();

        foreach (var hex in _affectedHexes) {
            if (hex.HasOccupant()) {
                if (hex.Character is not EnemyCharacter) {
                    _allyTracker.Add(hex.CharacterMB);
                    hex.Character.ResetMovementPoints();
                }
            }
        }
    }
```

Slika 103. BattlMapCombatPhase

Može imati samo jedno od dva stanja tipa CombatMode; MoveMode i AttackMode. Koristi listu fizičkih reprezentacija karaktera na mapi borbe. Ovdje potencijalno može nastati do curenja memorije kada je objekt s komponentom CharacterMB uništen. Kako se to ne bi dogodilo BattleMapGameplayManager može javiti da je neki CharacterMB uništen i onda ova klasa može ukloniti referencu na taj CharacterMB (Slika 104).

```
1 reference
public override void HandeleCharacterMBDestroyed(CharacterMB destroyedCharacterMB) {
    if (destroyedCharacterMB.Character is EnemyCharacter) {
        RemoveDeletedCharacterMBFromTrackers(destroyedCharacterMB);

        if (--_remainingEnemies == 0) {
            _battleMapGameplayManager.AlliesWin();
        }
    }
}

1 reference
protected void RemoveDeletedCharacterMBFromTrackers(CharacterMB destroyedCharacterMB) {
    if (destroyedCharacterMB.Character is AllyCharacter) {
        if (_allyTracker.Contains(destroyedCharacterMB)) {
            _allyTracker.Remove(destroyedCharacterMB);
        }
    }
}
```

Slika 104. Upravljanje referencama CharacterMB-a koji su uništeni

Poljem _remainingEnemies se prati koliko je neprijatelja preostalo. Kada vrijednost polja padne na nulu to označava kako je družina pobijedila i poziva se metoda AlliesWin BattleMapGameplayManager-a.

3.9.4.1. CombatMode

Radi se o ugniježdenoj apstraktnoj klasi koja predstavlja stanje stroja stanja BattleMapCombatPhase. Implementacija je prikazana ispod (Slika 105).

```
public partial class BattleMapCombatPhase {
    5 references
    private abstract class CombatMode {
        35 references
        protected BattleMapCombatPhase _battleMapCombatPhase;

        1 reference
        public abstract void HandleHexSelectedEvent(Hexa selectedHex);

        2 references
        public abstract void RunModeStartProcedure();
        2 references
        public abstract void RunModeEndProcedure();

        2 references
        public CombatMode(BattleMapCombatPhase bmcf) {
            _battleMapCombatPhase = bmcf;
        }
    }
}
```

Slika 105. CombatMode

3.9.4.2. MoveMode

Omogućuje igrači pomicanje karaktera. Način na koji kalkulira gdje se karakter može pomaknuti je prikazan ispod (Slika 106).

```
2 references
private List<(Hexa hex, int distance)> GetHexesThatCharacterCanMoveTo(Hexa selectedHex) {
    var maxMovementDistance = selectedHex.Character.AvailableMovementThisTurn;

    List<(Hexa, int)> visitedHexes = new();
    visitedHexes.Add((selectedHex, 0));

    List<List<Hexa>> levels = new();
    levels.Add(new List<Hexa>() { selectedHex });

    for (int i = 1; i <= maxMovementDistance; i++) {
        levels.Add(new());
        foreach (var hex in levels[i - 1]) {
            foreach (var dir in System.Enum.GetValues(typeof(HexCoords.Direction))) {
                var targetCoords = hex.coords.GetNeighbour((HexCoords.Direction)dir);
                var targetHex = _battleMapCombatPhase._affectedHexes.FirstOrDefault(x => x.coords == targetCoords);
                if (targetHex != null) {
                    if (visitedHexes.Where(x => x.Item1 == targetHex) is not null && !targetHex.HasOccupant()) {
                        visitedHexes.Add((targetHex, i));
                        levels[i].Add(targetHex);
                    }
                }
            }
        }
    }
    return visitedHexes;
}
```

Slika 106. GetHexesThatCharacterCanMoveTo metoda

Radi se o algoritmu obilaska u širinu (eng. breadth-first). Prvo dohvaća preostalu udaljenost koju karakter može prijeći ovaj potez. Potom kreira listu parova posjećenih heksagona i njihovih udaljenosti od odabranog heksagona, te dodaje odabrani heksagon. Zatim kreira dvodimenzijalnu listu razina i dodaje novu listu koja ima samo odabrani heksagon kao element. Slijedeća lista u listi razine će biti svi heksagoni (koji ispune uvjet) koji su udaljeni za jedan od odabranog heksagona, slijedeća lista će biti svi heksagoni koji su udaljeni za 2 od odabranog heksagona itd.

Maksimalnu udaljenost imamo i petljom se okupljaju susjedstva. Koristi se refleksija kako bi se prošli svi smjerovi nabrajanja HexCoords.Direction i za svaki HexCoords provjerava postoji li heksagon s tim koordinatama, ako postoji provjerava se je li već u posjećenima. Ako nije dalje se provjerava ima li karaktera. Ako nema karakter na odabranom heksagonu se može pomaknuti na taj heksagon i on se dodaje u listu posjećenih, zajedno s trenutnom vrijednošću brojača, i u zadnju listu listi razina. Lista posjećenih heksagona predstavlja sve heksagone na koje se karakter može pomaknuti i njihove udaljenosti, a da su unutar njegovog dosega pomaka.

Označavanje heksagona na koje se karakter na odabranom heksagonu može pomaknuti i način na koji se karakter pomiče na željeni heksagon su prikazani ispod (Slika 107).

```
1 reference
private void MoveToHex(Hexa hex) {
    int distance = GetHexesThatCharacterCanMoveTo(_battleMapCombatPhase.SelectedHex).Where(x => x.hex == hex).First().distance;
    var character = _battleMapCombatPhase.SelectedHex.Character;
    var characterMB = _battleMapCombatPhase.SelectedHex.RemoveOccupant();
    hex.SetOccupant(characterMB);

    _battleMapCombatPhase.DeductMovement(character, distance);
}

3 references
private void ShowWhereCharacterCanMoveTo(Hexa originHex) {
    _battleMapCombatPhase.DisableAllHexesAndResetState();

    var moveableTo = GetHexesThatCharacterCanMoveTo(originHex);
    foreach (var hex in moveableTo) {
        hex.Item1.InteractableHex.SetInteractable();
        hex.Item1.InteractableHex.SetMoveableToState();
    }

    _battleMapCombatPhase.SelectedHex.InteractableHex.SetSelectedState();
}
```

Slika 107. MoveToHex i ShowWhereCharacterCanMoveTo metode

Funkcijom MoveToHex se karakter na odabranom heksagonu prebacuje na željeni heksagon i od raspoloživih bodova za kretanje ovaj potez mu se oduzima udaljenost izvornog i odredišnog heksagona. Funkcijom ShowWhereCharacterCanMoveTo se svim heksagonuma u karakterovom dometu postavlja stanje na interaktivno i na MOVEBLE stanja. Na slici ispod (Slika 108) je prikazana procedura upravljanja događaja kada je kliknut interaktivan heksagon

procedura koja se izvršava kada BattleMapCombatPhase pređe na ovo stanje. Procedura se svodi na to da se resetiraju heksagoni na normalno stanje i postave na način koji ovo stanje zahtjeva (označeni karakteri koji nisu završili potez ili ako je odabran heksagon, gdje se karakter može pomaknuti).

Napomena: ako heksagon nije interaktivan, to jest ako mu je stanje interaktivan laž, InputManager neće podizati događaj.

```
public partial class BattleMapCombatPhase {
    2 references
    private class MoveMode : CombatMode {
        1 reference
        public override void HandleHexSelectedEvent(Hexa selectedHex) {
            if (_battleMapCombatPhase.SelectedHex is null) {
                _battleMapCombatPhase.SelectHex(selectedHex);
                ShowWhereCharacterCanMoveTo(selectedHex);
            } else if (_battleMapCombatPhase.SelectedHex == selectedHex) {
                _battleMapCombatPhase.DisableAllHexesAndResetState();
                _battleMapCombatPhase.SetHexesWithAlliesThatHaveNotEndedTheirTurnAsInteractable();
                _battleMapCombatPhase.DeselectHex();
            } else {
                _battleMapCombatPhase.DisableAllHexesAndResetState();
                MoveToHex(selectedHex);
                _battleMapCombatPhase.SelectHex(selectedHex);
                ShowWhereCharacterCanMoveTo(selectedHex);
            }
        }

        2 references
        public override void RunModeStartProcedure() {
            _battleMapCombatPhase.DisableAllHexesAndResetState();
            if (_battleMapCombatPhase.SelectedHex is not null) {
                _battleMapCombatPhase.SelectedHex.InteractableHex.SetSelectedState();
                ShowWhereCharacterCanMoveTo(_battleMapCombatPhase.SelectedHex);
            } else {
                _battleMapCombatPhase.SetHexesWithAlliesThatHaveNotEndedTheirTurnAsInteractable();
            }
        }
```

Slika 108. Obrada događaja klika na heksagon u MoveMode fazi

3.9.4.3. AttackMode

Attack mode je drugo stanje u kojem BattleMapCombatPhase stroj stanja može biti. Kroz ovo stanje se izvode napadi i radi toga je potreban je CombatExecutorSO (ScriptableObject) koji može raditi izračune napada. Konstruktor je prikazan ispod (Slika 109).

```
public partial class BattleMapCombatPhase {
    3 references
    private class AttackMode : CombatMode {
        4 references
        private CombatExecutorSO _combatExecutor;

        1 reference
        public AttackMode(BattleMapCombatPhase bmcf, CombatExecutorSO combatExecutor) : base(bmcf) {
            _combatExecutor = combatExecutor;
        }
```

Slika 109. AttackMode konstruktor

Način na koji klasa dohvaća heksagone koji su u dometu napadačevog oružja je prikazan ispod (Slika 110).

```
private List<Hexa> GetHexesThatCharacterCanAttack(Hexa selectedHex) {
    var selectedCharater = _battleMapCombatPhase.SelectedHex.Character;
    Weapon charactersWepon;
    if (selectedCharater.Equipment.Armament.TwoHanded is not null) {
        charactersWepon = selectedCharater.Equipment.Armament.TwoHanded;
    } else if (selectedCharater.Equipment.Armament.MainHand is not null) {
        charactersWepon = selectedCharater.Equipment.Armament.MainHand;
    } else if (selectedCharater.Equipment.Armament.OffHand is not null) {
        charactersWepon = selectedCharater.Equipment.Armament.OffHand;
    } else {
        charactersWepon = null;
    }

    int distanceToHighlightEnemies =
        charactersWepon is not null ?
            (selectedCharater.IsMelee ? charactersWepon.WeaponStats.MeleeStats.Reach :
                 charactersWepon.WeaponStats.RangedStats.Range) : 1;

    List<Hexa> moveableHexes = new();

    List<Hexa> visited = new();
    visited.Add(selectedHex);

    List<List<Hexa>> fingers = new();
    fingers.Add(new List<Hexa>() { selectedHex });

    for (int i = 1; i <= distanceToHighlightEnemies; i++) {
        fingers.Add(new());
        foreach (var hex in fingers[i - 1]) {
            foreach (var dir in System.Enum.GetValues(typeof(HexCoords.Direction))) {
                var targetCoords = hex.coords.GetNeighbour((HexCoords.Direction)dir);
                var targetHex = _battleMapCombatPhase._affectedHexes.FirstOrDefault(x => x.coords == targetCoords);
                if (targetHex != null) {
                    if (!visited.Contains(targetHex)) {
                        visited.Add(targetHex);
                        fingers[i].Add(targetHex);
                    }
                }
            }
        }
    }
    return visited;
```

Slika 110. GetHexesThatCharacterCanAttack metoda AttackMode faze

Potrebno je odgonetnuti kakvo oružje karakter ima. Ako ima oružje za borbu prsa o prsa, onda je domet oružja svojstvo Reach, ako je dalekometno oružje onda mu je domet Range. Potom se provodi algoritam obilaska u širini (slično kao i kod MoveMode klase, ali sad

se ignoriraju prepreke). S obzirom na to da je ovo stanje u koje se mora prijeći iz stanja MoveMode, bojanje heksagona se izvodi samo u RunModeStartProcedure metodi i prikazano je ispod (Slika 111). Stanje svih heksagona u dometu se postavlja na ATTACKABLE, ali se samo ona s neprijateljskim karakterima postavljaju kao interaktivna. Može se vidjeti i to da se u toj metodi dodaje slušač koji prati izvršavanje kalkulacija napada, a on se uklanja u metodi, na istoj slici, RunModeEndProcedure.

```
2 references
public override void RunModeStartProcedure() {
    _battleMapCombatPhase.DisableAllHexesAndResetState();
    _battleMapCombatPhase.SelectedHex.InteractableHex.SetInteractable();
    _battleMapCombatPhase.SelectedHex.InteractableHex.SetSelectedState();

    var hexesWithEnemiesInRange = GetHexesThatCharacterCanAttack(_battleMapCombatPhase.SelectedHex);
    _battleMapCombatPhase.DisableAllHexes();
    foreach (var hex in hexesWithEnemiesInRange) {
        if (hex.HasOccupant() && hex.Character is EnemyCharacter) hex.InteractableHex.SetInteractable();
        hex.InteractableHex.SetAttackableHexState();
    }
    _battleMapCombatPhase.SelectedHex.InteractableHex.SetInteractable();
    _battleMapCombatPhase.SelectedHex.InteractableHex.SetSelectedState();
    _combatExecutor.Executing.AddListener(EndCharacetrsTurn);
}

2 references
public override void RunModeEndProcedure() {
    _combatExecutor.Executing.RemoveListener(EndCharacetrsTurn);
}
```

Slika 111- RunModeStartProcedure AttackMode faze

Upravljanje događaja klika na interaktivan heksagon se izvodi u funkciji HandleHexSelectedEvent prikazano ispod (Slika 112).

```
2 references
private void EndCharacetrsTurn(bool combatExecutormExecuting) {
    if (!combatExecutormExecuting)
        _battleMapCombatPhase.EndCharactersTurn();
}

1 reference
public override void HandleHexSelectedEvent(Hexa selectedHex) {
    if (_battleMapCombatPhase.SelectedHex == selectedHex) {
        _battleMapCombatPhase.DeselectHex();
        _battleMapCombatPhase.TransitionToMoveMode();
    } else {
        _combatExecutor.AttackCharacter(
            attackerHex: _battleMapCombatPhase.SelectedHex,
            defenderHex: selectedHex
        );
    }
}
```

Slika 112. HandleHexSelectedEvent AttackMode faze

3.9.5. EscorteeBattlePhase

Radi se o „međupotezu“ gdje računalo kontrolira pratioca i jedan je od stanja BattleMapGameplayManager stroja stanja. Metoda StartPhase je prikazana ispod (Slika 113). S obzirom da pratilac može napadati, potreban je CombatExecutorSO.

```
1 reference
public override void StartPhase() {
    if (_remainingEnemies == 0) _battleMapGameplayManager.AlliesWin();
    else {
        _combatExecutor.Executing.AddListener(EndTurn);

        _escorteeMB = _affectedHexes.Where(x => x.Character is EscorteeCharacter).First().CharacterMB;

        _battleMapGameplayManager.StartWaitForEscorteeMovement();
    }

}
```

Slika 113.EscorteeBattlePhase StartPhase metoda

Kako se njegov potez ne bi izvršio u jednom trenutku, potrebna je korutina kojom će se izvođenje malo usporiti. Kako ova klasa ne proširuje MonoBehavoiour, ne može započinjati korutine. Ta odgovornost je ovdje podignuta na BattleMpaGameplayManager i on započinje

korutinu. Iako je korutinu nemoguće započeti iz klase koja ne proširuje MonoBehaviour, to ne znači da ju u ovoj klasi ne možemo definirati. Funkcija je prikazana ispod (Slika 114).

```
1 reference
public IEnumerator WaitAndTryToAttack() {
    yield return new WaitForSeconds(1f);
    _escorteeMB.Hex.InteractableHex.SetSelectedState();
    yield return new WaitForSeconds(1f);
    DoSomething();
}
```

Slika 114. WaitAndTryToAttack korutina

Prvo se čeka jednu sekundu, potom se označi heksagon na kojemu je pratilac, opet čeka jednu sekundu i onda pratilac vrši svoj potez. Metoda DoSomething je prikazana ispod (Slika 115).

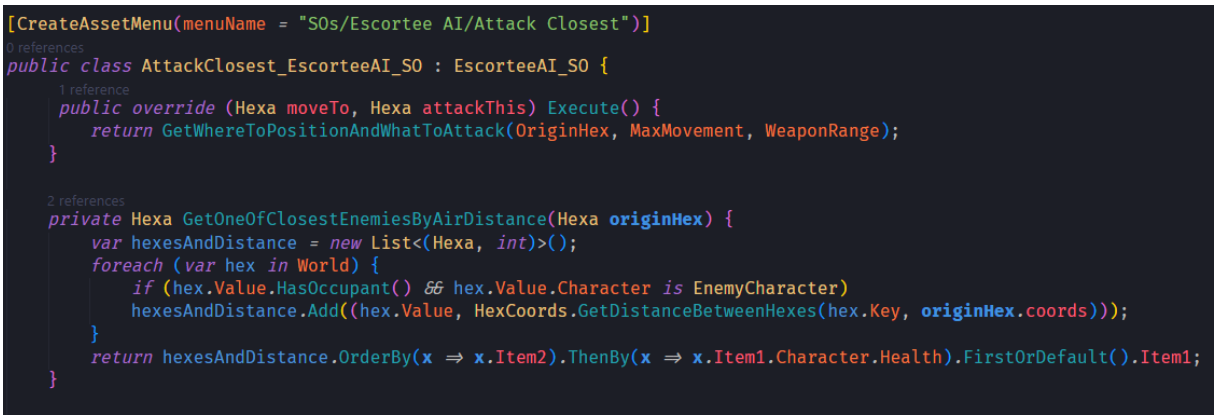
```
1 reference
private void DoSomething() {
    var ai = ((EscorteeCharacter)_escorteeMB.Character).AI;
    ai.SetHexWorld(_affectedHexes);

    var weapons = _escorteeMB.Character.Equipment.GetWeapons();
    int weaponRange = 1;
    if (_escorteeMB.Character.IsMelee) {
        if (weapons.mainHand is not null) {
            weaponRange = Mathf.Max(weapons.mainHand.WeaponStats.MeleeStats.Reach, weaponRange);
        }
        if (weapons.offHand is not null) {
            weaponRange = Mathf.Max(weapons.offHand.WeaponStats.MeleeStats.Reach, weaponRange);
        }
    } else {
        if (weapons.mainHand is not null) {
            weaponRange = Mathf.Max(weapons.mainHand.WeaponStats.RangedStats.Range, weaponRange);
        }
        if (weapons.offHand is not null) {
            weaponRange = Mathf.Max(weapons.offHand.WeaponStats.RangedStats.Range, weaponRange);
        }
    }

    ai.InitializeAI(_escorteeMB.Hex, _escorteeMB.Character.MovementPointsPerTurn, weaponRange);

    var result = ai.Execute();
    if (result.moveTo is not null) {
        MoveToHex(_escorteeMB.Hex, result.moveTo);
    }
    if (result.attackThis is not null) {
        _combatExecutor.AttackCharacter(_escorteeMB.Hex, result.attackThis);
        _escorteeMB.Hex.InteractableHex.SetUninteractable();
    }
    else {
        EndTurn(true);
    }
}
```

Slika 115. DoSomething metoda

Kako bi računalo moglo izvršiti akcije, potrebno mu je znanje o svijetu, potom mu je potrebno znanje o karakterovom dometu i o dometu njegovog oružja nakon čega računalo stvara potez.

3.9.5.1. AttackClosest_EscorteeAI_SO

Ovaj ScriptableObject definira vrlo jednostavno ponašanje pratioca - „Napadni najbližeg neprijatelja“. Izvođenje se poziva metodom Execute (Slika 116). Metoda vraća par heksagona gdje je prvi onaj na koji se karakter treba pomaknuti (ako je null, onda se ne pomiče), a drugi heksagon je heksagon koji treba napasti (ako je null, onda ne napada). Na istoj slici je i metoda GetOneOfClosestEnemiesByAirDistance koja vraća heksagon jednog od najbližih neprijatelja gledajući udaljenosti zračnom duljinom, to jest ne razmatra prepreke. Ako su dva neprijatelja jednako udaljena, odabrati će onoga s manje životnih bodova.



Slika 116. Execute metoda AttackClosest_EscorteeAI_SO

Na slijedećim slikama je metoda GetWhereToPositionAndWhatToAttack koja je malo predugačka za jednu sliku pa ju je potrebno razdvojiti.

Slika 117. GetWhereToPositionAndWhatToAttackl_1

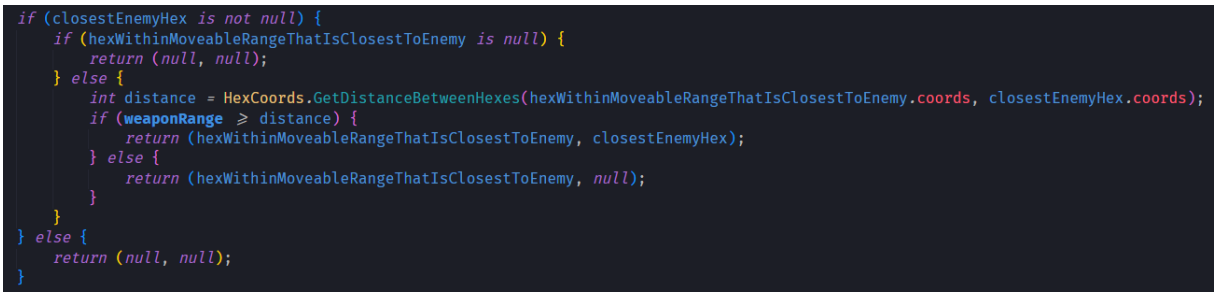
Prvo se traži najbliži neprijatelj na cijeloj mapi (Slika 117). Potom se računa zračna udaljenost između dva heksagona korištenjem Manhattanske udaljenosti i ako je najbliži neprijatelj unutar dometa oružja, funkcija vraća par (null, heksagon najbližeg neprijatelja). U suprotnom se radi priprema za algoritam obilaska u širinu. Varijable frontier predstavlja heksagone koji su zadovolji uvjete i može se nastaviti pretraga u njihovom susjedstvu. Rječnik distances će održavati registar posjećenih heksagona i kolika im je udaljenost od izvorišnog heksagona. Ako se ne pronađe neprijatelj kojeg je moguće napasti ovaj potez, onda moramo odabrati gdje se pomaknuti kako bi potencijalno slijedeći potez neprijatelj mogao biti napadnut. To predstavlja varijabla hexWithinMoveableRangeThatIsClosestToEnemy.

Nastavak funkcije je ispod (Slika 118).

```
while (frontier.Count > 0) {
    var currentHex = frontier.Dequeue();
    int distance = distances[currentHex];

    var neighbourCoords = HexCoords.GetNeighboursAtRandom(currentHex.coords);
    foreach (var coord in neighbourCoords) {
        if (World.ContainsKey(coord)) {
            var neighbourHex = World[coord];

            if (neighbourHex.HasOccupant()) {
                continue;
            }

            if (distance + 1 > maxMovement) {
                break;
            }

            Hexa newClosestEnemyHex = GetOneOfClosestEnemiesByAirDistance(neighbourHex);
            if (newClosestEnemyHex is not null) {
                int newAirDistanceToClosestEnemyHex = HexCoords.GetDistanceBetweenHexes(newClosestEnemyHex.coords, neighbourHex.coords);
                if (!distances.ContainsKey(neighbourHex) && newAirDistanceToClosestEnemyHex <= shortestAirDistanceToEnemyHex) {

                    frontier.Enqueue(neighbourHex);
                    distances[neighbourHex] = distance + 1;

                    if (newAirDistanceToClosestEnemyHex < shortestAirDistanceToEnemyHex) {
                        hexWithinMoveableRangeThatIsClosestToEnemy = neighbourHex;
                        closestEnemyHex = newClosestEnemyHex;
                        shortestAirDistanceToEnemyHex = newAirDistanceToClosestEnemyHex;
                    }
                }
            }
        }
    }
}
```

Slika 118. GetWhereToPositionAndWhatToAttackl_2

Svaku iteraciju glavne petlje se bilježi udaljenost heksagona koji je izvađen iz reda frontier u rječnik distances. Kako kretanje ne bi bilo pre očito, susjedi se dohvaćaju nasumično. To je ostvareno metodom GetNeighboursAtRandom. Ta metoda vraća listu koordinata susjednih heksagona. Zatim se provjerava postoje li heksagoni s tim koordinatama u svijetu i za svaki od tih heksagona se provjerava imaju li karaktera, ako imaju (očito nije neprijatelj jer bi već imali rješenje) preskačemo tog susjeda. Slijedeće se provjerava je li udaljenost plus jedan veća od dometa kretanja, ako je prekida se traženje. U suprotnom se traže najbliži neprijatelji po zračnoj udaljenosti i ako takav postoji računa se zračna udaljenost od trenutnog susjeda do najbližeg heksagona s neprijateljem. Potom se provjerava imamo li taj susjed već bio posjećen (ako je preskačemo ga) i ako nije provjerava se je li nova zračna udaljenost manja ili jednaka trenutnoj najbližoj udaljenosti do najbližeg neprijatelja. Ako je, susjed se dodaje u red i udaljenost do njega se bilježi u rječnik. Slijedi dodatna provjera gdje se gleda je li ova nova najbliža udaljenost do najbližeg neprijatelja manja od trenutne. Ako je bilježi se trenutni susjed kao najbliži heksagon neprijatelju koji je u dometu kretanja i neprijateljev heksagon se bilježi kao najbliži heksagon s neprijateljem. Također bilježi se da je nova najkraća udaljenost od najbližeg neprijatelja trenutna najbliža. Na slici ispod je nastavak funkcije (Slika 119).

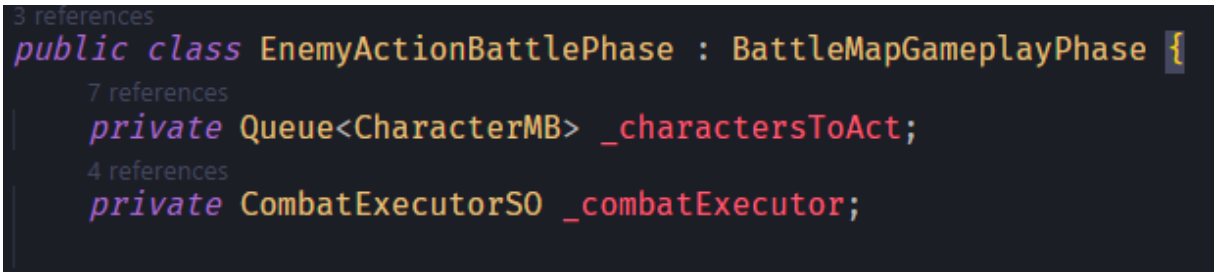


Slika 119. GetWhereToPositionAndWhatToAttackl_3

Ako je pronađen najbliži neprijatelj provjerava se postoji li heksagon koji je najbliži neprijatelju. Ako ne postoji vraća se par (NULL, NULL). U suprotnom računa se udaljenost između najbližeg heksagona unutar dometa kretanja i najbližeg heksagona s neprijateljem. Ako je domet oružja veći ili jednak toj udaljenosti vraća se par (najbliži heksagon unutar dometa kretanja, najbliži neprijatelj), u suprotnom vraća se (najbliži heksagon unutar dometa kretanja, NULL).

3.9.6. EnemyActionBattlePhase

Posljednje stanje BattleMapGameplayManager stroja stanja i vrlo je slično EscorteeActionBattlePhase stanju. Razlika je u tome što postoji više neprijatelja i svi moraju izvršiti svoj potez. Kako bi svaki neprijatelj izvršio potez koristi se red (Slika 120) i kada je karakter na redu za izvršavanje poteza, iz reda gase vadi.



Slika 120. EnemyActionBattlePhase korištenje reda

Neprijatelji su pobijedili kada je podignut događaj uništenja objekta CharacterMB čiji je Character tipa EscorteeCharacter (Slika 121).

```
1 reference
public override void HandeleCharacterMBDestroyed(CharacterMB destroyedCharacterMB) {
    if (destroyedCharacterMB.Character is not EnemyCharacter) {
        if (destroyedCharacterMB.Character is EscorteeCharacter) {
            _blockFurtherActions = true;
            _battleMapGameplayManager.EnemiesWin();
        }
    }
}
```

Slika 121. HandleCharacterMBDestroyed kod EnemyCombatPhas-e

3.9.6.1. AttackClosest_EnemyAI_SO

Radi se o vrlo sličnoj klasi kao AttackClosest_EscorteeAI_SO. Jedina razlika je u metodi GetOneOfClosestEnemiesByAirDistance i prikaza je ispod (Slika 122).

```
private Hexa GetOneOfClosestEnemiesByAirDistance(Hexa originHex) {
    var hexesAndDistance = new List<(Hexa, int)>();
    foreach (var hex in World) {
        if (hex.Value.HasOccupant() && hex.Value.Character is not EnemyCharacter)
        hexesAndDistance.Add((hex.Value, HexCoords.GetDistanceBetweenHexes(hex.Key, originHex.coords)));
    }
    return hexesAndDistance.OrderBy(x => x.Item2)
        .ThenBy(x => x.Item1.Character is EscorteeCharacter ? 0 : 1)
        .ThenBy(x => x.Item1.Character.Health).FirstOrDefault().Item1;
}
```

Slika 122. GetOneOfClosestEnemiesByAirDistance metoda

Razlika je u vrsti karaktera koji se smatraju neprijateljima. Neprijatelji moraju napadati one koji nisu „njihovi“, stoga ovdje se provjerava je li karakter drugog tipa od EnemyCharacter i ako je ga se dodaje u list. Također, kako bi neprijatelji prioritizirali pratioca, karakter kojega metoda vraća je prvo onaj najbliži, pa onda pratilac, a tek onda onaj s najmanje životnih bodova.

3.10. CombatExecutorSO

ConbatExecutorSO u OnEnable funkciji dodaje slušač na događaj koji se oglašava kada je animacija ispisivanja rezultata napada gotova (Slika 123). Također, ako nije, stvara novu instancu svojeg događaja Executing.

Rezultat napada može biti jedan od nabrajača HitResults. (Slika 123)

```
0 references
void OnEnable() {
    Executing ??= new();
    FloatingAnimationFinished_EventListener.Enable(ContinueNormalOperation);
}
0 references
void OnDisable() {
    FloatingAnimationFinished_EventListener.Disable();
}

35 references
public enum HitResult {
    0 references
    NotDetermined,
    11 references
    Hit,
    4 references
    Grazed,
    4 references
    Missed,
    2 references
    Blocked,
    2 references
    Dodged
}
```

Slika 123. CombatExecutorSO OnEnable, OnDisable i HitResults

Prije ikakvih kalkulacija se podiže događaj kojim se označava da se napad izvodi (Slika 124). Potom se provjerava je li karakter naoružan, ako nije podiže se događaj da je izračun završio i staje ikakva danja kalkulacija.

```
public void AttackCharacter(Hexa attackerHex, Hexa defenderHex) {
    Executing.Invoke(true);
    if (attackerHex.Character is EnemyCharacter && _enemyOneShot) {
        int damage = 100000;
        DealDamageToCharacter(defenderHex.CharacterMB, damage);
        DisplayHitResult(defenderHex.CharacterMB, (HitResult.Hit, damage));
        return;
    } else if (_allyOneShot) {
        int damage = 100000;
        DealDamageToCharacter(defenderHex.CharacterMB, damage);
        DisplayHitResult(defenderHex.CharacterMB, (HitResult.Hit, damage));
        return;
    }

    var distance = HexCoords.GetDistanceBetweenHexes(attackerHex.coords, defenderHex.coords);

    if (attackerHex.Character.IsArmed) {
        DoAttack(attackerHex, defenderHex);
    } else {
        Executing.Invoke(false);
    }
}
```

Slika 124. AttackCharacter metoda

Ako napadač ima dva oružja, ponovo se provodi proces kako bi se izračunao napad s drugim oružjem, a onda s drugim. Karakteri s dva oružja napadaju s oba (Slika 125)

```
private void DoAttack(Hexa attackerHex, Hexa defenderHex) {
    var attacker = attackerHex.Character;
    var defender = defenderHex.Character;
    var distance = HexCoords.GetDistanceBetweenHexes(attackerHex.coords, defenderHex.coords);

    if (attacker.Equipment.Armament.TwoHanded is not null) {
        var result = ExecuteAttack(attacker.Stats, attacker.Equipment.Armament.TwoHanded, defender, distance);
        DealDamageToCharacter(defenderHex.CharacterMB, result.damageDealt);
        DisplayHitResult(defenderHex.CharacterMB, result);
    } else {
        List<(HitResult hitResult, int damageDealt)> results = new();
        if (attacker.Equipment.Armament.MainHand is not null) {
            var result = ExecuteAttack(attacker.Stats, attacker.Equipment.Armament.MainHand, defender, distance);
            results.Add(result);
            DealDamageToCharacter(defenderHex.CharacterMB, result.damageDealt);
        }
        if (attacker.Equipment.Armament.OffHand is not null) {
            var result = ExecuteAttack(attacker.Stats, attacker.Equipment.Armament.OffHand, defender, distance);
            results.Add(result);
            DealDamageToCharacter(defenderHex.CharacterMB, result.damageDealt);
        }
        DisplayHitResult(defenderHex.CharacterMB, results);
    }
}
```

Slika 125. DoAttack metoda

Proces kojim se određuje rezultat napada je slijedeći: prvo se gleda preciznost napadačevog oružja. Ako se radi o oružju za borbu prosa o prsa onda se gleda statistika oružja Handling, ako je oružje za dalekometnu borbu onda se gleda statistika oružja Accuracy. Kako bi se odredilo je li napadač uspio pogoditi svoju metu (Slika 126) koristi se metoda TryToHitWithMeleeWeapon za oružja koja su namijenjena za borbu prsa o prsa, to jest metoda TryToHitWithRangedWeapon za oružja koja su namijenjena za dalekometnu borbu.

```
1 reference
private HitResult TryToHitWithRangedWeapon(Weapon weapon, int distance) {
    float accuracy = weapon.WeaponStats.RangedStats.Accuracy / 100f;
    float weaponRange = weapon.WeaponStats.RangedStats.Range;

    var accuracyDrop = _weaponAccuracyDropOnRangeCurve.Evaluate(distance / weaponRange);
    float modifiedAccuracy = accuracy - accuracyDrop;
    modifiedAccuracy = Mathf.Clamp(modifiedAccuracy, 0f, 1f);

    if (modifiedAccuracy < Random.Range(0f, 1f)) {
        return HitResult.Missed;
    } else {
        return HitResult.Hit;
    }
}
1 reference
private HitResult TryToHitWithMeleeWeapon(Weapon weapon) {
    if (weapon.WeaponStats.MeleeStats.Handling / 100f < Random.Range(0f, 1f)) {
        return HitResult.Missed;
    } else {
        return HitResult.Hit;
    }
}
```

Slika 126. TryToHitWithMelee/RangedWeapon metode

Ako je napadač uspio pogoditi metu, branitelj se ima priliku obraniti na jedan od dva načina (Slika 127).

```
private HitResult TryToScoreHit(Weapon attackersWeapon, Character defender, int distance) {
    bool rangedAttack = attackersWeapon.EquipmentSO.Type.Classification is RangedWeaponClassificationSO;

    float dodgeChance = GetDodgeChance(defender);
    float blockChance = rangedAttack ?
        GetRangedBlockChance(defender.Equipment.Armament.TwoHanded, defender.Equipment.Armament.MainHand, defender.Equipment.Armament.OffHand) :
        GetMeleeBlockChance(defender.Equipment.Armament.TwoHanded, defender.Equipment.Armament.MainHand, defender.Equipment.Armament.OffHand);


    if ((rangedAttack ? TryToHitWithRangedWeapon(attackersWeapon, distance) : TryToHitWithMeleeWeapon(attackersWeapon)) == HitResult.Hit) {
        if (PrioritizeBlockingAttack(dodgeChance, blockChance)) {
            return TryToBlock(blockChance);
        } else {
            return TryToDodge(dodgeChance);
        }
    } else {
        return HitResult.Missed;
    }
}
```

Slika 127. TryToScoreHit metoda

Ako ima veće šanse izbjeći napad nego ga blokirati, probati će ga izbjeći; i obrnuto. Ako se odluči probati izbjeći napad, koristi se metoda ispod (Slika 128).

```
1 reference
private HitResult TryToDodge(float dodgeChance) {
    float grazeSave = 0.1f;

    float randomValue = Random.Range(0f, 1f);

    if (dodgeChance >= randomValue) {
        return HitResult.Dodged;
    } else if (Mathf.Clamp(dodgeChance + grazeSave, 0f, 1f) >= randomValue) {
        return HitResult.Grazed;
    } else {
        return HitResult.Hit;
    }
}
```

Slika 128. TryToDodge metoda

Ako se odluči probati blokirati napad, koristi se metoda ispod (Slika 129).

```
1 reference
private HitResult TryToBlock(float blockChance) {
    if (blockChance < Random.Range(0f, 1f)) {
        return HitResult.Hit;
    } else {
        return HitResult.Blocked;
    }
}
```

Slika 129. TryToBlock metoda

Ako je napad nije uspješno blokiran ili izbjegnut onda se smatra da se radi o pogotku (ili ogrebotini ako ne branitelju malo nedostajalo da izbjegne napad. Onda se rješava pitanje koji dio karaktera je pogođen (Slika 130).

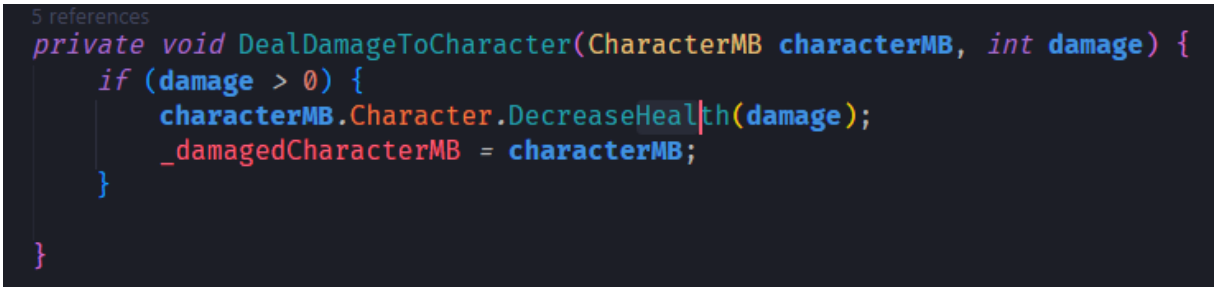
```
private (HitResult hitResult, int damageDealt) ExecuteAttack(CharacterStats attackersStats, Weapon attackersWeapon, Character defender, int distance) {
    HitResult hitResult = TryToScoreHit(attackersWeapon, defender, distance);
    int damageDealt = 0;

    if (hitResult == HitResult.Hit || hitResult == HitResult.Grazed) {
        var hitArmor = GetHitArmor(defender);
        int armorValue = hitArmor is not null ? hitArmor.ArmorStats.ArmorValue : 0;

        int baseDamage;
        int penetration;
        int scalingStat;

        if (attackersWeapon.EquipmentSO.Type.Classification is RangedWeaponClassificationSO) {
            baseDamage = attackersWeapon.WeaponStats.RangedStats.BaseDamage;
            penetration = attackersWeapon.WeaponStats.RangedStats.Penetration;
            scalingStat = attackersStats.Agility;
        } else {
            baseDamage = attackersWeapon.WeaponStats.MeleeStats.BaseDamage;
            penetration = attackersWeapon.WeaponStats.MeleeStats.Penetraion;
            scalingStat = attackersStats.Strenght;
        }

        damageDealt = hitResult == HitResult.Hit ?
            CalculateHitDamage(armorValue, baseDamage, penetration, scalingStat) :
            CalculateGrazeDamage(armorValue, baseDamage, penetration, scalingStat);

        return (hitResult, damageDealt);
    } else {
        return (hitResult, 0);
    }
}
```

Slika 130. ExecuteAttack metoda

I na kraju se računa šteta koju je napadač nanio branitelju (Slika 131). Količina štete ovisi o vrijednosti oklopa pogođenog dijela tijela i o statistici napadača i njegovog oružja.

```
private int CalculateHitDamage(int armorValue, int baseDamage, int piercing, int scalingStat) {
    float damage = baseDamage + baseDamage * _bonusDamageFromScalingCharacterStat.Evaluate(scalingStat / (float)CharacterStats.SkillCap);

    if (piercing >= armorValue) {
        float armorValueToPiercingRatio = armorValue / (float)piercing;
        float chanceToGetDamageReduction = _chanceToGetDamageReductionCurve.Evaluate(armorValueToPiercingRatio);
        if (chanceToGetDamageReduction > Random.Range(0f, 1f)) {
            damage *= _damageReductionCurve.Evaluate(Random.Range(0f, 1f));
        }
    } else {
        float piercingRatioToArmorValue = piercing / (float)armorValue;
        float chanceToPenetrate = _chanceToPenetrateToughArmor.Evaluate(piercingRatioToArmorValue);
        if (chanceToPenetrate > Random.Range(0f, 1f)) {
            damage *= _damageReductionCurve.Evaluate(Random.Range(0f, 1f));
        } else {
            damage *= _minimalDamagePercentage;
        }
    }
    return Mathf.FloorToInt(damage);
}
1 reference
private int CalculateGrazeDamage(int armorValue, int baseDamage, int piercing, int scalingStat) {
    return Mathf.FloorToInt(CalculateHitDamage(armorValue, baseDamage, piercing, scalingStat) * _grazeDamagePercentage);
}
```

Slika 131. CalculateHit/GrazeDamage metode



```
5 references
private void DealDamageToCharacter(CharacterMB characterMB, int damage) {
    if (damage > 0) {
        characterMB.Character.DecreaseHealth(damage);
        _damagedCharacterMB = characterMB;
    }

}
```

Slika 132. DealDamageToCharacter metoda

Kada je sve izračunato, šteta se oduzima od karakterovih životnih bodova (Slika 132) i ispisuje (Slika 133).

```
private void DisplayHitResult(CharacterMB characterMB, (HitResult hitResult, int damageDealt) hit) {
    characterMB.DisplayDamage(AssembeDamageReportText(hit));
}

1 reference
private void DisplayHitResult(CharacterMB characterMB, List<(HitResult hitResult, int damageDealt)> hits) {
    string text = string.Empty;
    for (int i = 0; i < hits.Count; i ++) {
        text += AssembeDamageReportText(hits[i]);
        if (i + 1 < hits.Count) {
            text += "\n";
        }
    }
    characterMB.DisplayDamage(text);
}

2 references
private string AssembeDamageReportText((HitResult hitResult, int damageDealt) hit) {
    string text = GetString(hit.hitResult);
    if (hit.hitResult == HitResult.Hit || hit.hitResult == HitResult.Grazed) {
        text += ": -" + hit.damageDealt + "HP";
    } else {
        text += "!";
    }
    return text;
}
```

Slika 133. DisplayHitResult metode

4. Zaključak

Iako rad u Unity-u ponekad zna biti noćna mora radi nekakvog problema koji mi nikada ne bi pao na pamet i dalje mi je jedan od najdražih alata. Imao sam situacije gdje bih potrošio tjedne na dizajn neke funkcionalnosti samo kako bih zapeo zato što to Unity ne može napraviti i cijeli dizajn bi trebalo početi raditi nanovo.

Na kraju, aplikacija ima jako puno mjesta za unaprijeđena kao što su refaktorizacija kôda i dodavanje funkcionalnosti. Žao mi je što nisam kreirao sustav gdje igrač može mijenjati opremu karaktera i različite, zanimljive vrste susreta koji nisu samo bitke. Također, igra je postavljana na 2,5D svijet i bilo bi zanimljivo kreirati fizičke reprezentacije karaktera. Sustav osvjetljenja je isto mogao podići cjelokupni dojam.

Brojke je potrebno dalje balansirati, bitke se znaju previše protezati, što zbog lošeg balansa, a što zbog neugodnih čekanja prije izvršavanja poteza.

Jako sam zadovoljan sustavom opreme, to jest jednostavnom modularnom izradom opreme kroz inspektor. Zadovoljan sam i sustavom borbe i načinom ponašanja pratioca. Naučio sam jako puno o Unity-evoj serijalizaciji objekata (na težak, ali fer način). Smatram da sam uložio previše vremena u ovaj rad i mogu sa sigurnošću reći da bi bilo puno lakše kada bi se ovakvi projekti radili u timu. Naravno, to nije nepoznato ali, priznajem, mislio sam da će biti puno jednostavnije (možda se radi o Dunning–Kruger efektu). Sve u svemu, uzeo bih ovu temu ponovo.

Popis Literature

[1] „Unity Personal“. https://unity.com/products/unity-personal (pristupljeno 09. travanj 2023.).

[2] „Unity - Developing Your First Game with Unity and C# | Microsoft Learn“. https://learn.microsoft.com/en-us/archive/msdn-magazine/2014/august/unity-developing-your-first-game-with-unity-and-csharp (pristupljeno 09. travanj 2023.).

[3] U. Technologies, „Unity - Manual: Important Classes - MonoBehaviour“. https://docs.unity3d.com/Manual/class-MonoBehaviour.html (pristupljeno 10. travanj 2023.).

[4] U. Technologies, „Unity - Scripting API: MonoBehaviour.Awake()“. https://docs.unity3d.com/ScriptReference/MonoBehaviour.Awake.html (pristupljeno 10. travanj 2023.).

[5] U. Technologies, „Unity - Scripting API: MonoBehaviour.OnEnable()“. https://docs.unity3d.com/ScriptReference/MonoBehaviour.OnEnable.html (pristupljeno 10. travanj 2023.).

[6] U. Technologies, „Unity - Manual: Order of execution for event functions“. https://docs.unity3d.com/Manual/ExecutionOrder.html (pristupljeno 10. travanj 2023.).

[7] U. Technologies, „Unity - Scripting API: MonoBehaviour.Start()“. https://docs.unity3d.com/ScriptReference/MonoBehaviour.Start.html (pristupljeno 10. travanj 2023.).

[8] U. Technologies, „Unity - Scripting API: MonoBehaviour.FixedUpdate()“. https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html (pristupljeno 10. travanj 2023.).

[9] U. Technologies, „Unity - Scripting API: MonoBehaviour.Update()“. https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html (pristupljeno 10. travanj 2023.).

[10] „Input System | Input System | 1.5.1“. https://docs.unity3d.com/Packages/com.unity.inputsystem@1.5/manual/index.html (pristupljeno 10. travanj 2023.).

[11] U. Technologies, „Unity - Scripting API: MonoBehaviour.LateUpdate()“. https://docs.unity3d.com/ScriptReference/MonoBehaviour.LateUpdate.html (pristupljeno 10. travanj 2023.).

[12] U. Technologies, „Unity - Scripting API: MonoBehaviour.OnDisable()“. https://docs.unity3d.com/ScriptReference/MonoBehaviour.OnDisable.html (pristupljeno 10. travanj 2023.).

[13] U. Technologies, „Unity - Scripting API: MonoBehaviour.OnDestroy()“. https://docs.unity3d.com/ScriptReference/MonoBehaviour.OnDestroy.html (pristupljeno 10. travanj 2023.).

[14] U. Technologies, „Unity - Scripting API: ScriptableObject“. https://docs.unity3d.com/2023.2/Documentation/ScriptReference/ScriptableObject.html (pristupljeno 10. travanj 2023.).

[15] „Visual Studio: IDE and Code Editor for Software Developers and Teams“, *Visual Studio*. https://visualstudio.microsoft.com (pristupljeno 09. travanj 2023.).

[16] „GIMP - About GIMP“. https://www.gimp.org/about/introduction.html (pristupljeno 09. travanj 2023.).

[17] „Future Tools - Leonardo.ai“. https://www.futuretools.io/tools/leonardo-ai (pristupljeno 09. travanj 2023.).
[18] A. J. Patel, „Hexagonal Grids“, Red Blob Games, 2013. Pristupljeno: 11. travanj 2023. [Na internetu]. Dostupno na: https://www.redblobgames.com/grids/hexagons/
[19] „UI support | Input System | 1.0.2“. https://docs.unity3d.com/Packages/com.unity.inputsystem@1.0/manual/UISupport.html (pristupljeno 11. travanj 2023.).
[20] Lowelltech, „Answer to ‚Unity3D New Input System: Is it really so hard to stop UI clickthroughs (or figure out if cursor is over a UI object)?‘“, *Stack Overflow*, 03. kolovoz 2022. https://stackoverflow.com/a/73217597 (pristupljeno 11. travanj 2023.).

Popis Slika