Unity aplikacija za borbene igre s kartama

Mešić, Eldin

Undergraduate thesis / Završni rad

2022

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:114961

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-12-29**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU

FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Preddiplomski sveučilišni studij

UNITY APLIKACIJA ZA BORBENE IGRE S KARTAMA

Završni rad

Eldin Mešić

Osijek, 2022.

Sadržaj

1. UVOD

Tema ovog završnog rada je izrada desktop aplikacije, odnosno programa koristeći Unity Game Engine kao osnovu rada. Ova aplikacija je vrsta borbene kartaške igre (engl. *Combat Card Game*) sa sličnim principom rada i mogućnostima kao i neke od poznatijih uspješnih borbenih kartaških igara napravljenih od tvrtki vrijednih više milijardi. Glavni izazov ovog završnog bi prema tome bio pokušaj pravljenja funkcionalne i u potpunosti izrađene igre koja posuđuje elemente iz sličnih igara kao *Hearthstone, Magic: The Gathering, Legends of Runeterra, Yu-Gi-Oh: Mater Duel* i ostalih poznatih igara iz žanra borbenih kartaških igara i igri sa razmjenjivim kartama (engl. *Trading Card Games*). Pošto je ova igra napravljena bez ikakvog novčanog fonda i sa znanjem stečenim na fakultetu, nemoguće je da ona dostigne razinu detalja i svih mogućnosti igara napravljenih od velikih tvrtki. Aplikacija koristi razne tehnike programiranja Unity razvojnog jezika kao što su scriptable objekti čija će korist i način stvaranja biti objašnjeni u sljedećim poglavljima zajedno i sa ostalih korištenim tehnikama i rješenjima pojavljenih problema.

Ova aplikacija bi trebala služiti za poticanje kritičnog razmišljanja korisnika u smislu strategije, previđanja koraka unaprijed i razvijanja snalažljivosti korisnika u određenim situacijama, te naravno za samu zabavu korisnika koji vole žanr borbenih kartaških igara.

1.1. Zadatak završnog rada

Za ovaj završni rad potrebno je napraviti borbenu kartašku igru preko Unity razvojnog sustava koristeći C# programski jezik, te bazu podataka iz koje će se preko SQL upita dohvatiti podatci o kartama, korisnicima i njihovim spremljenim postavkama.

Sama desktop aplikacija mora imati mogućnosti registracije i prijave u aplikaciju, kontrole zvuka, pregleda svih karata, pravljenja špilova (engl. *Decks*) koje korisnik može koristiti, borbeni dio igre gdje bi se korisnik borio protiv AI protivnika, te bi svaka karta trebala imati mogućnost naknadno biti dodana u igru preko baze podataka, te posjedovati od nijednog do proizvoljnog broja specijalnih mogućnosti korištenih u borbenoj fazi igre.

2. PREGLED PODRUČJA TEME RADA

Ova desktop aplikacija je napravljena na osnovu TGC i CCG aplikacija koje su unaprijedile format igara baziranih na borbenim kartama. Njihov uspjeh je pokazao kako za interesantnu, zabavnu i tešku igru nije potrebno napraviti kompliciranu aplikaciju sa više stotina mehanika. Važno je imati širok raspon radnji ili sadržaja unutar svake mehanike koje će korisnike zadržavati zainteresiranim, a ujedno time i voljnim podržavati igru u njenim kasnijim pothvatima.

2.1. Hearthstone

Hearthstone je besplatna online digitalna igra bazirana na skupljanju karti koje se kasnije koriste za pravljenje špilova te za borbu protiv drugih igrača. Razvijena je od Blizzard Entertainmenta sa originalnim nazivom *Hearthstone: Heroes of Warcraft.* Prvi put je objavljena u ožujku 2014. godine za Windows i macOS sustave te sljedeće godine za iOS i Android sustave. Igrači mogu igrati jedni protiv drugih preko bilo koje kombinacije raspoloživih platformi na kojima je igra objavljena. Jedino ograničenje je geografska regija igrača.

Hearthstone je igra na poteze između dva protivnika, koristeći špilove od 30 karata, gdje svaki igrač pokušava protivnika dovesti do nula životnih bodova koristeći karte koje predstavljaju čudovišta ili magije. Ova igra umjesto čudovišta koristi takozvane sljedbenike, pomagače koji imaju svoje životne i napadačke bodove te razne efekte koji na neki način utječu na igru. Igrači koriste svoje ograničene mana kristale za odigravanje karata na polje odakle ih mogu koristiti za napad ili obranu. Pobjedom u borbenom dijelu igre, igrači su nagrađeni zlatom s kojim mogu kupiti pakete novih karata kako bi ojačali svoje špilove ili napravili nove. Postoje i razni drugi načini igranja, uključujući obične bitke, rangirane bitke, draft bitke i avanture za jednog igrača. Novi sadržaj za igru uključuje dodavanje novih setova karata u obliku paketa proširenja ili avantura koje igrače nagrađuju sa kolekcijskim kartama.

Veliki dio popularnosti ove igre se može pridodati uspješnosti druge Blizzard igre, *World of Warcraft*, na kojoj je *Hearthstone* baziran i čije likove koristi. Još jedan razlog brze uspješnosti je to što je u igri potpuno izbačena zamka ostalih TGC i CCG igri eliminirajući sve mogućnosti protivnika tijekom poteza igrača i replicirajući osjećaj fizičke kartaške igre unutar korisničkog

sučelja. Mnoge animacije i glasovi likova u različitim situacijama su dodatni razlog zašto *Hearthstone* još uvijek živi i stoji pri vrhu borbenih kartaških igara.

2.2. Legends of Runeterra

Legends of Runeterra je digitalna kolekcionarska kartaška igra objavljenja u travnju 2020. godine od strane Riot Games. Nadahnuti fizičkom kolekcionarskom kartaškom igrom *Magic: The Gathering*, programeri Riot Games-a su nastojali napraviti igru unutar istog žanra koja značajno smanjuje barijeru ulaska. Igra je dostupna za Windows, iOS i Android te je besplatna.

Kao i kod drugih kolekcionarskih kartaških igara, igrači igraju jedan protiv drugoga kako bi protivniku smanjili životne poene na nulu. Dio po kojem se ova igra razlikuje od ostalih iz istog žanra je uvođenjem prvaka (engl. *Champions*) koji su posebna vrsta karti koje se tijekom jedne borbe mogu unaprijediti ako ispune određene uvijete što ih čini jačim.

Isto kao i za *Hearthstone*, velika popularnost ove igre je njezina povezanost sa drugom Riot Games igrom, *League of Legends*, čiji svijet, regije i likove posuđuje. Cijeli svijet odakle dolaze razni likovi iz igre je detaljno opisan te igračima daje osjećaj poznavanja i bliskosti sa likovima.

Igra je dobro prihvaćena od strane kritičara, koji ističu njezin velikodušni sustav napredovanja, pristupačan način igranja i kvalitetne vizualne elemente zbog kojih je osvojila nekoliko nagrada u industriji razvoja igara.

2.3. Magic: The Gathering

Magic: The Gathering je digitalna kolekcionarska i borbena kartaška igra osnovana na istoimenoj stolnoj igri od strane Richard Garfielda. Digitalna verzija igre je objavljena od strane Wizard of the Coast kompanije. Stolna verzija je bila prva igra s kartama za razmjenu i do prosinca 2018. godine je imala preko 35 milijuna igrača.

Igrač preuzima ulogu Planeswalkera, moćnog čarobnjaka koji može putovati između dimenzija multiverzuma, vodeći bitku protiv drugih igrača bacanjem magija, korištenjem artefakta i prizivanjem čudovišta kao što je prikazano na pojedinačnim kartama izvučenim iz njihovih špilova.

Igra ima nekoliko raznih načina igranja, ali cilj je uvijek spustiti protivniku ili protivnicima život na nula bodova. Špilovi u *Magic: The Gathering*-u se sastoje od 40 karata koji ovisno o pravilu koje igrači koriste se prave na jedan od dva stila, ograničeni i konstruirani. Kod ograničenog stila igrači spontano sastavljaju špil od skupa nasumičnih karata, dok kod konstruiranog stila igrači prave špilove normalno, koristeći koje god karte žele iz njihove kolekcije.

Ova igra je uvela posebna pravila borbene faze, specifično obrana i napad. Kada igrač napada na svom potezu, on odabere koje njegove karte će napasti ali ne i šta će one napasti, taj dio obrane bira protivnik koji može obraniti sa svojim kartama igračeve na koji god to način on želio.

Slika 2.1. Borbeno sučelje igre *Magic: The Gathering*

3. KORIŠTENE TEHNOLOGIJE

U ovom poglavlju opisane su tehnologije koje su korištene u izradi rada. Svaka od ovih tehnologija ima svoje prednosti i svoj ključni dio u izradi ovog završnog rada. Pošto je kombinacija nekih od ovih tehnologija veoma česta u pravljenju desktop aplikacija postoje mnogi načini da se njihov rad poveže, spoji i prenese na drugu tehnologiju. Ne spomenuto kao jedne od tehnologija, ali opet važne da se napomenu su ostale poznate borbene kartaške igre koje su korištene za inspiraciju, prikaz rada funkcionalnost, te dodataka kao slike ili dijelovi korisničkog sučelja.

3.1 Unity Game Engine

Unity razvojno okruženje je jedno od najkorištenijih alata za ravoj web, mobilnih i desktop aplikacija. Prva verzija je objavljena 9. kolovoza 2005. godine, a razvila ga je tada nova tvrtka Unity Technologies. Razvojno okruženje je od svojeg nastanka samo nastalo rasti u popularnosti. Njegov jednostavan alat za razvoj video igara i računalne grafike je olakšao mnogim programerima da se uključe u svijet razvoja komliciranijih aplikacija. Još jedan razlog njegove uspješnosti je to što ga je moguće izvoditi na svim platformama koje se nalaze na računalnom tržištu. Tako su se Unity projekti počeli pojavljivati na osobnim računalima, koji imaju Windowse ili Linux za operacijske sustave te na preglednicima kao web plugini ili konzole i mobilnim uređajima. Njegovo korištenje od strane svih tipova programera na različitim uređajima je osiguralo njegovu uspješnost i popularnost na tržištu [1].

Unity se smatra prvom platformom koja je u potpunosti podržavala sve mogućnosti uređaja i operacijskih sustava. Korisnicima su dani svi alati potrebni za razvoj igara kao što su alati za uređivanje slika, 3D modeliranje i povezivanje sa bilo kojom vrstom koda [2]. Također, unošenjem nativnih formata je omogućeno integriranje vektorske ili neke druge izmjenjive grafike iz programa kao što su Photoshop te korištenje 3D i 2D raznolikih modela iz poznatih alata za modeliranje kao što su Blender i Maya.

Razvojno okruženje je napisano u tada poznatim i popularnim C i C++ programskim jezicima. Skriptiranje i korištenje grafičkog sučelja je olakšano tako što je sagrađen omotač (engl. *Wrapper*) koji postavlja sloj za pristup .NET jeziku. Taj dodatak je najviše utjecao na developere igara koje su bazirane na jezgri sustava.

U svrhu dodatnog proširenja raznovrsnost platforme, osiguran je API za razvoj u C#, Boo-u i Javascriptu [3].

Unity je veoma jak alat koji, zbog ugrađenih funkcionalnosti dostupnih klikom miša, olakšava izradu video igara. Važno je i reći da bi implementacija jedne takve funkcionalnosti mogla trajati čak i do nekoliko tjedana. Neke od tih kompliciranih, ali i korisnih, funkcionalnosti su kolizije, osvjetljene, fizika, animacije, itd. [4].

Što se tiče programskog koda, dolazi s ugrađenim IDE-om pod nazivom MonoDevelop. On je napravljen kako bi podržavao programske jezike Visual Basic .NET, C, C++, C#, F# i Vala.

U ovom radu, korišten je za sam izgled aplikacije, razmještaj i funkcionalnost raznih objekata i elemenata korisničkog sučelja, spajanje svih objekata i dijelova sa C# skriptama napisanim preko Visual Studia, za držanje svih potrebnih resursa, te za sam rad aplikacije.

Slika 3.1. Unity korisničko sučelje

3.2 .NET Framework

.NET Framework je sustav koji nadograđuje mogućnosti samog operativnog sustava. Radi se o posebnoj infrastrukturi koja programerima nudi gotova rješenja i funkcionalnosti da bi ubrzala i pojednostavila razvoj aplikacija svih vrsta i oblika. Instrukcije u programu se u realnom vremenu prevode u izvorni strojni kod koji razumije računalo. Za taj je posao zaslužan JIT-kompajler (engl. *Just In Time*). Upravo prevođenje u izvorni strojni kôd računala, omogućilo je .NET-u prelazak na druge operativne sustave kao što su Linux ili MacOS (putem pomoćnog third-party MONO sustava).

Najvažnija sastavnica .NET Frameworka zove se Common Language Runtime ili skraćeno CLR. CLR je softverski sustav u kojem se kod izvršava. Kada korisnik pokrene aplikaciju pisanu za .NET Platformu, CLR ju izvršava kako bi joj osigurao stabilnost i funkcionalnost [5].

Aplikacije za .NET platformu mogu se pisati u raznim programskim jezicima, gotovo svim poznatijim. CLR, međutim, ne poznaje niti jedan taj jezik - on dobiva naredbe isključivo u jeziku nazvanom Microsoft Intermediate Language (skraćeno MSIL), temeljen na pravilima koja se nazivaju Common Language Specifications (CLS). Stoga je jasno da mora postojati kompajler koji će programski jezik u kojem programer piše kod prevesti u MSIL kako bi ga CLR razumio. Ovi kompajleri nazivaju se IL-kompajleri te su dostupni za velik broj programskih jezika. Microsoft je izdao kompajlere za pet jezika: C#, J#, C++, Visual Basic i JScript, dok su se ostali proizvođači softvera potrudili oko brojnih drugih.

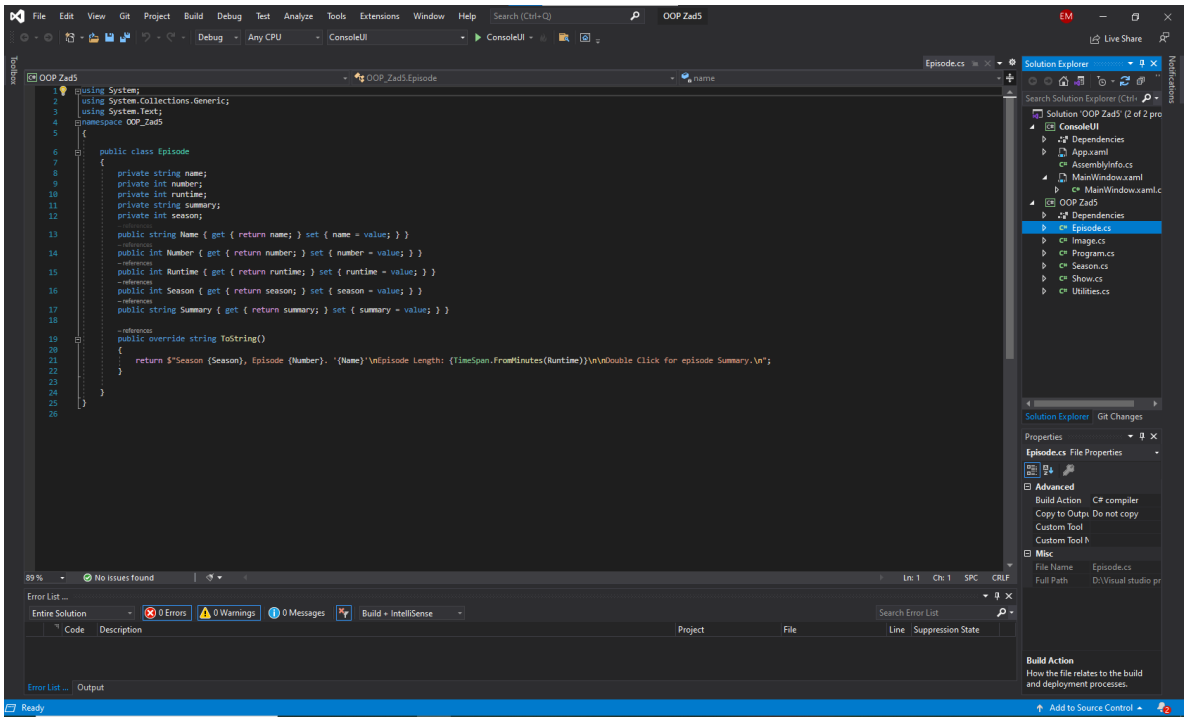
U ovom završnom radu .NET Framework je služio kao infrastruktura Visual Studia i Unity-a, te je korištena verzija .NET Framework-a bila .Net Framework 4.8. koji je dodavao mogućnost korištenja SQL biblioteke u Visual Studiu iako je bio povezan sa starijom verzijom Unity-a.

3.3 Visual Studio

Visual studio je integrirano razvojno okruženje (engl. *Integrated Development Environment*) koje je napravio i razvio Microsoft. Koristi se za razvoj računalnih programa, kao i web stranica, web aplikacija, web servisa i mobilnih aplikacija. Visual Studio koristi razne Microsoft platforme za razvoj softvera kao što su Windows API, Windows Forms, Windows Presentation Foundation, Windows Store i Microsoft Silverlight.

Visual Studio sadržava i uređivač koda koji podržava IntelliSense, komponentu za dovršavanje koda, te ujedno koji služi za refaktoriranje samog koda. Ostali ugrađeni alati uključuju alat za profiliranje koda, dizajner za izradu GUI aplikacija, web dizajner, dizajner klasa i dizajner shema baza podataka [5]. Jedna od glavnih mogućnosti Visual Studia je i mogućnost prihvaćanja dodataka (engl. *Plugins*) koji proširuju funkcionalnost na gotovo svakoj razini, uključujući i dodavanje podrške za sustave kontrole izvora kao što su Git i Subversion, te još i dodavanje novih skupova alata kao što su uređivači i vizualni dizajneri.

Visual Studio Podržava 36 različitih programskih jezika i omogućuje uređivaču koda i programu za ispravljanje grešaka da podržavaju skoro pa bilo koji programski jezik, sa nekoliko izuzetaka koje predstavljaju manje korišteni programski jezici ili jezici sa neuređenim kompiliranjem koda. Neki od tih ugrađenih jezika su C, C++, C#, Visual Basic, JavaScript, TypeScript, XML, XSLT, HTML, CSS i razni ostali. Čak i za jezike koji nisu direktno podržani od strane Visual Studia, kao što su Python i Ruby, postoje dodatci koji ih ugrađuju u podržani spektar Visual Studia.



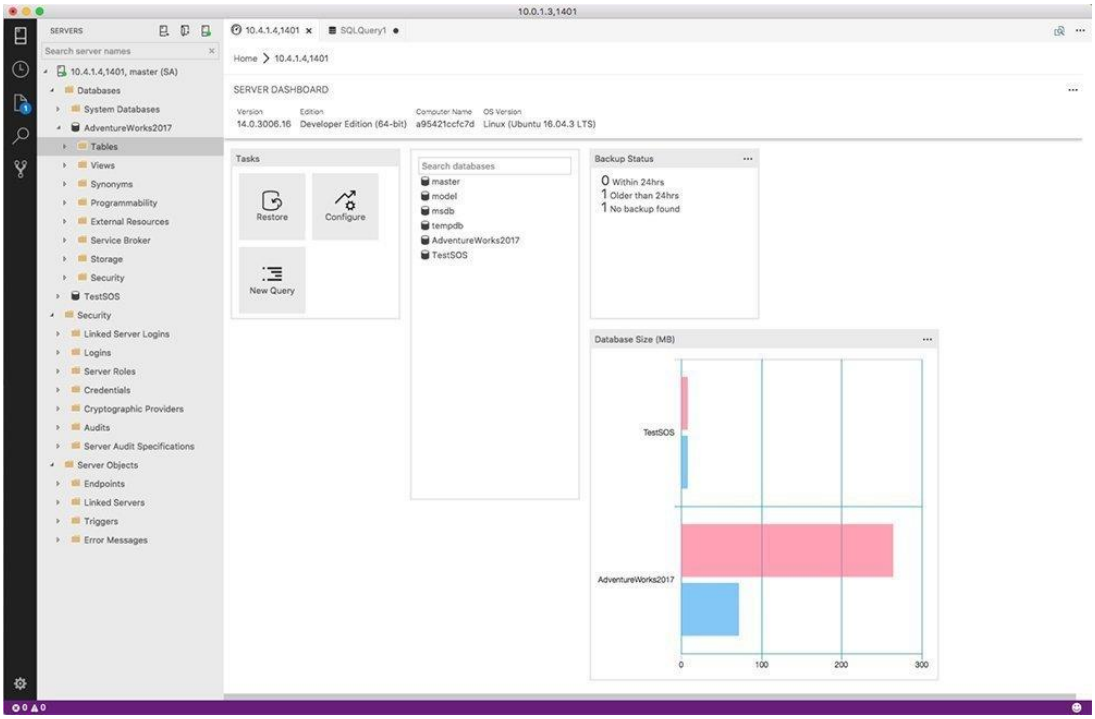
Slika 3.2. Visual Studio korisničko sučelje

Možda i glavni razlog za popularnost Visual Studia je njegova dostupnost svima. Visual Studio je podržan na svim platformama, te je njegovo najosnovnije izdanje, Visual Studio Community, dostupno besplatno što je velik razlog zašto ga mnogi studenti ili početnici kodiranja koriste.

U ovom završnom radu Visual Studio je korišten kao primarni program u kojem se pisao C# kod za funkcionalnost raznih GameObject elemenata Unity okruženja, te za objedinjavanje cijelog koda pod jednim kompajlerom i glavnim sučeljem.

3.4 Azure Data Studio

Azure Data Studio je besplatni Miscrosoftov alat koji se koristi za kreiranje i upravljanje baza podataka SQL servera, Azure SQL baza podataka i Azure SQL skladišta podataka temeljenim na oblaku. To je softver sa jednostavnim načinom korištenja napravljen u svrhu olakšavanja rutinskog razvoja baze podataka, postavljanja upita i administracije. Njegova osnova je bio Microsoftov SQL Server Managment Studio (SSML), koji je bio dosta spor, zahtjevan, te nepregledan. Zbog toga Azure Data Studio nije samo namijenjen administratorima baza podataka, već i softver programerima i početnicima koji uobičajeno ne koriste ili ne znaju dovoljno o samim bazama podataka i njihovom održavanju [6].



Slika 3.3. Azure Data Studio korisničko sučelje

Azure Data Studio koristi Microsoftov Visual Studio uređivač strojnog koda kao svoju osnovu. Integrirani prozor terminala omogućuje korisnicima da izvršavaju naredbe za upravljanje bazom podataka pomoću Bash, PowerShell, sqlcmd i drugih alata naredbenog retka izravno iz korisničkog

sučelja Azure Data Studia. Ovaj alat također uključuje ugrađene značajke poput prozora s više kartica, T-SQL uređivača, pametne navigacije kodom i integracije sa Git sustavom za praćenje izvornog koda. U T-SQL uređivaču korisnici mogu spremati rezultate upita baze podataka kao tekstualne datoteke, JSON datoteke ili Microsoft Excel datoteke [7]. Također, osim mogućnosti razvoja upita, uređivač podržava i stvaranje pohranjenih procedura, skripti i drugih objekata baze podataka.

Korisnici Azure Data Studia mogu stvoriti prilagodljive nadzorne, tj. komadne ploče za praćenje i rješavanje nakupljanja podataka i drugih problema u bazama podataka i na razini poslužitelja. Azure Data Studio sadržava i posebno odvojene nadzore ploče poslužitelja i baze podataka, koje se mogu konfigurirati s dodatcima za uvid kako bi pružili brz, efikasan i detaljan pogled na razine performansi, korištenja sustava i drugih mjerljivih podataka. Osim pregleda tih informacija, korisnici mogu napraviti i sigurnosnu kopiju i vratiti baze podataka, uređivati podatke u tablicama baze podataka i objavljivati druge zadatke upravljanja iz nadzornih ploča.

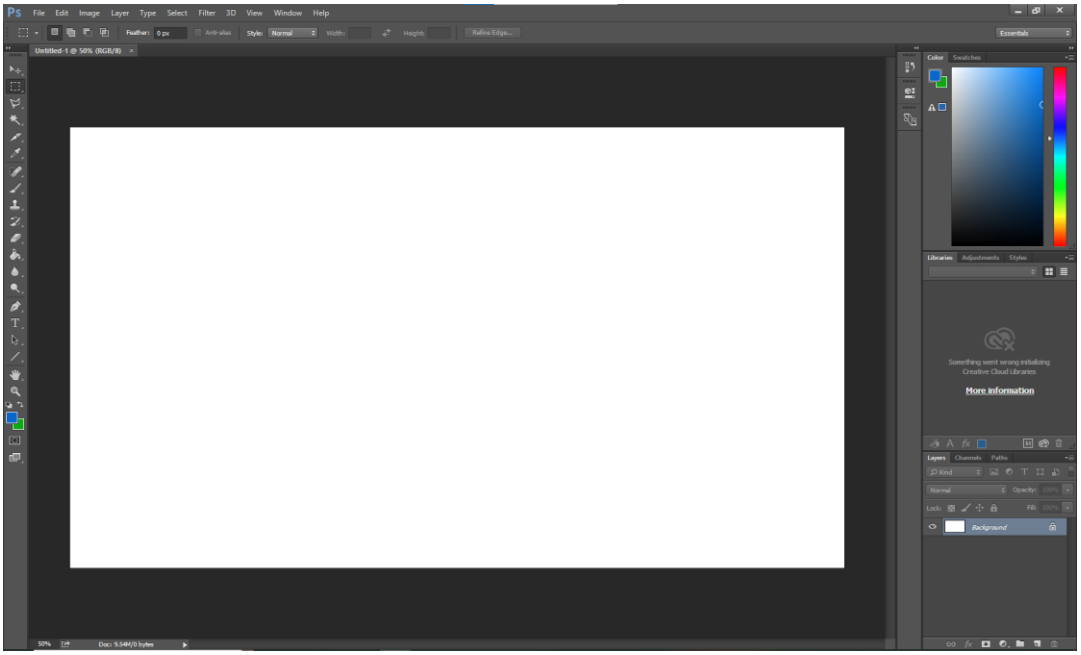
Izvorni kod za Azure Data Studio i njegove pružatelje podataka je dostupan na GitHub-u pod EULA izvornim kodom koji daje prava na izmjenu i korištenje softvera, ali ne i za njegovu redistribuciju ili širenje preko servisa oblaka [7].

U ovom završnom radu Azure Data Studio je imao ključnu ulogu kao poslužitelj baze podataka koja sadržava informacije o svim karata, njihovim efektima, te korisnicima. U bazi se također i veze između karata i efekata, špilova i karata, korisnika i njihovih špilova koje će biti objašnjenje u veće detalje u kasnijim poglavljima završnoga rada. Pošto Azure servis radi u sklopu Visual Studia bio je najbolji izvor za pružanje usluge baze podataka.

3.5 Adobe Photoshop

Program Adobe Photoshop je grafički uređivač razvijen od strane Adobe Inc. za Windows i MacOS. Od njegovog nastanka 1998 je postao industrijski standard ne samo u uređivanju grafike i slika, već i u digitalnoj umjetnosti kao cjelini. Postao je toliko poznat i popularan da se veoma često njegovo ime koristi kao glagol za izmjenu, odnosno, uređivanje slike. Ima mogućnosti uređivanja i sastavljanja slike u više slojeva (engl. *layers*) podržavanja maski, alpha sastavljanja i korištenja nekoliko modela boja uključujući RGB, CMYK, CIELAB, Spot Colour i Duotone.

Za podršku ovih značajki, podatke sprema pod svojim proširenjem i tipom podatka .PSD i .PSB. Također, osim rasterske grafike posjeduje i ograničene mogućnosti za uređivanje ili učitavanje teksta i vektorske grafike, kao i 3D grafike i videa. Kao i svaki program, Photoshop ima svoje nedostatke ali oni se mogu ublažiti ili čak u potpunosti riješiti koristeći dodatke koji proširuju funkcionalnost programa.



Slika 3.4. Korisničko sučelje Adobe Photoshop-a

U korisničkom sučelju programa (kao što se može vidjeti na slici 3.4), s lijeve strane zaslona se nalazi bočna traka sa najčešće koričenim alatima sa višestrukim funkcijama za uređivanje slika. Ti alati najčešće spadaju u kategorije crtanja, slikanja, mjerenja i navigacije, odabiranja, pisanja i retuširanja. Neki alati imaju mali trokut u donjem desnom kutu svoje ikone koji predstavljaju dodatni izbornik za taj alat koji ga mijenja sa drugim alatom ili alatima sličnih mogućnosti. Photoshop je veoma zahtjevan program za koji se treba provesti dugo vremena učeći njegove brojne mogućnosti i funkcionalnosti, pa je zato, u svrhu olakšanja njegovog korištenja novijim korisnicima, u novim verzijama dodan mali video prikaz korištenja svakog alata.

Što se tiče ovog završnog rada, Photoshop je skriveni pomagač čije su se mogućnosti koristile za uređivanje gotovo svake slike, sučelja i grafičkog elementa koji su korišteni unutar Unity Game Engine-a, a time i samog završnog rada.

4. PROGRAMIRANJE APLIKACIJE

Prije samog početka rada, potrebno je odrediti šta sve ova aplikacija treba sadržavati gledajući na modele drugih sličnih aplikacija, te odrediti koje od njihovih mehanika ubaciti u našu aplikaciju, a koje ne.

Aplikacija mora imati glavni izbornik preko kojeg bi korisnik birao da li želi otići na kolekciju gdje može gledati sve karte u aplikaciji, te mijenjati i praviti špilove karata ili otići na sam borbeni dio aplikacije gdje će moći izabrati koji špil želi koristiti u borbenom dijelu. Sam borbeni dio će imati teren na koji će se karte odigrati, prikaz špilova protivnika i igrača, ruka protivnika i igrača, gdje bi karte u ruci protivnika trebale biti sakrivene od igrača kako ne bi znao šta protivnik ima u ruci. Protivnikovo ponašanje će biti veoma jednostavno i o njemu će se reći više u kasnijim poglavljima završnog rada. Protivnik i igrač će imati avatare sa 30 životnih poena (engl. *Health*), te će cilj aplikacije biti spustiti protivnika na nula životnih poena prije nego on sruši igrača na nula životnih poena.

Borbeni dio aplikacije će biti baziran na potezima, što znači da će igrač moći raditi razne akcije kao pozivanje karata, napadanje sa kartama koje su već na polju, provjeravati karte na polju ili naravno, samo odmarati i/ili razmišljati o sljedećem ili sadašnjem potezu bez ikakvog utjecaja sa protivničke strane dok god igrač ne završi svoj potez. Kad igrač završi svoj potez, kreće potez protivnika u kojem on može raditi iste stvari kao i igrač, osim naravno razmišljanja i odmaranja.

Karte će se dijeliti na dvije vrste; Spells (magije) i Minions (sljedbenici) ili ,drugim imenom, Allies (saveznici). Također karte imaju cijenu pozivanja, koliko ECTS bodova je potrebno da bi se karta odigrala, snagu (Power / Attack), Život (Health), naziv i opis šta karta radi, ali taj dio će se ispuniti preko dodavanja efekata (engl. *Effects*) u kasnijem dijelu ovog poglavlja.

4.1 Objekt CCG karte

Kao što je rečeno u uvodnom dijelu poglavlja karte imaju osnovne vrijednosti (engl. *Stats*) koje predstavljaju njihovu cijenu, snagu i život. Karte će također posjedovati svoje ime i sliku, te bi također trebale imati neki otvoreni prazni dio gdje će se kasnije moći upisati opis efekta karte. Pošto ova igra treba moći imati podršku za dodavanjem više karata, pojavljuje se problem popunjavanja

aplikacije objektima. Kako bi riješili taj problem, koristi se tehniku zvanu ispunjivi objekti (engl. *Scriptable Objects*). Ta metoda kaže da napravimo osnovnu formu CCG (Combat Card Game) karte kao klasu, te Unity GameObject tip objekta koji će primiti vrijednosti te klase. Nakon toga ćemo taj GameObject spremiti kao Unity Prefab, što znači da će taj objekt imati izmjenjive vrijednosti, te će se moći inicijalizirati sa različitim vrijednostima, a istom funkcionalnošću.

Na kodu 4.1 može se vidjeti naša *Card* klasa koja sadrži sve potrebne informacije o karti uključujući i polje efekata koje će kasnije biti objašnjeno.

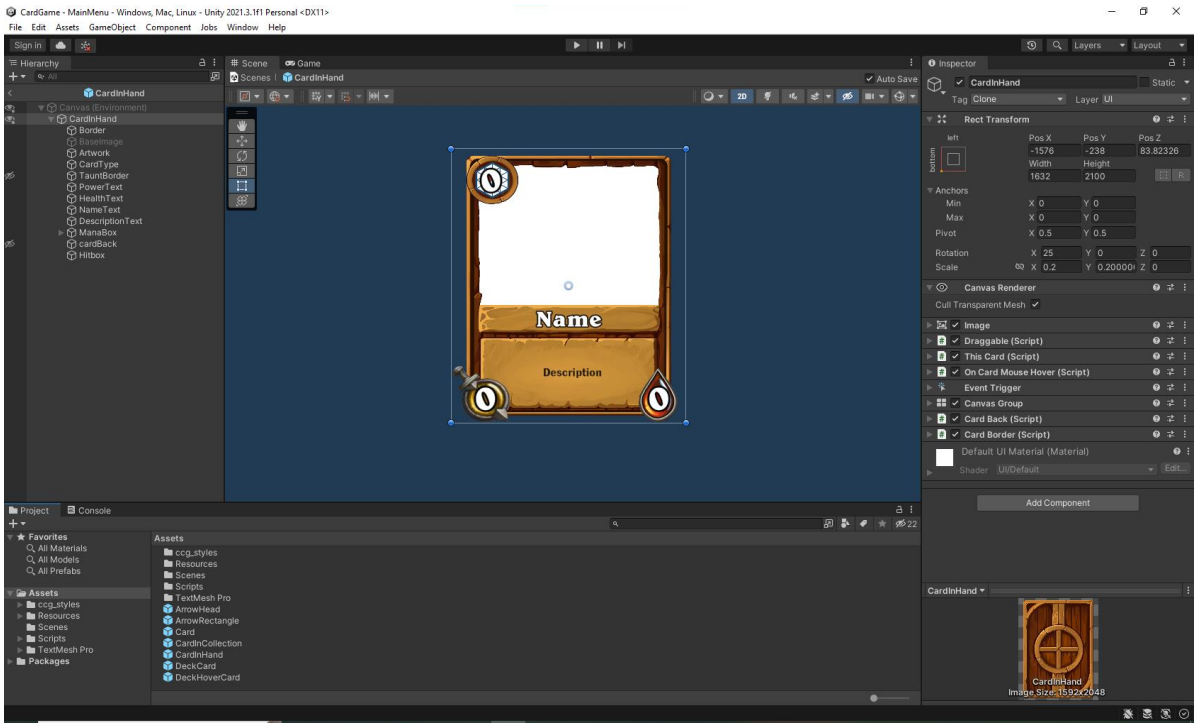
```
public class Card
{
    public int id;
    public string cardName;
    public int cost;
    public int power;
    public int health;
    public string cardDescription;
    public Sprite image;
    public bool isSpell;
    public List<Effect> cardEffects = new List<Effect>();

    public Card(int id, string cardName, int cost, int power, int health,
string
    cardDescription, Sprite image, bool isSpell, List<Effect> cardEffects)
{...}

    public List<Effect> CardEffects {...}
}
```

Primjer koda 4.1. *Card* klasa

U Unity-u, koristeći kombinacije praznih objekata, slika i tekst skripti sklopili smo izgled karte. Na glavni vrhovni objekt koji sadrži sve ostale objekte koji čine kartu dodamo Visual Studio C# skriptu naše klase *Card*. Nakon toga taj objekt u potpunosti spremimo kao Prefab pod nazivom *CardInHand*. Dodana je i posebna slika spremljena pod objektom CardType koja izgled karte mijenja ovisno o tome da li je karta magija ili sljedbenik.

Sve informacije o karti, kao i opći detalji funkcionalnosti karte su povezani sa objektom karte preko skripte *ThisCard* koja predstavlja glavni aspekt karte i koja je dodana na vrh našeg ispunjivog objekta.

Kao što se na slici 4.1 može vidjeti karta ima tekstne okvire za njenu cijenu pozivanja, snagu, život, ime i opis karte koji su na početku postavljeni na određene osnovne vrijednosti, te naravno praznu sliku koja će biti popunjena prema željenoj slici prikladne karte.



Slika 4.1. CardInHand Unity Prefab bez dodijeljenih vrijednosti

4.2 Efekti karata

Kako bi se poboljšao sam dizajn igre, te potaknulo igrače na dublje razmišljanje, planiranje i stvaranje strategije potrebno je dodati različite efekte, odnosno, mogućnosti karata. Ovo će također riješiti problem istih karata zato jer karte više nisu samo ograničene sa osnovnim vrijednostima, a samim time i malim ograničenim brojem karata, jer svaka karta mora biti drugačija. Dodavanjem efekata širimo raspon različitosti karata, te namještanjem da karte mogu imati koliko god želimo efekata dobivamo forme karata koje se ne bi trebale ponavljati. Efekti bi trebali imati neki okidač koji određuje kada će se efekt karte aktivirati, te također karta bi trebala moći podržavati više različitih okidača, svaki sa svojim efektom.

Kao što se može vidjeti na kodu 4.2, veoma slično kao i za klasu Card, napravljena je klasa Effect koja predstavlja i sadrži potrebne informacije koje će se koristiti da se odrediti koji tip efekta će biti aktiviran, kada će se taj efekt aktivirati, te sa kojim vrijednostima će utjecati na okruženje borbenog dijela igre.

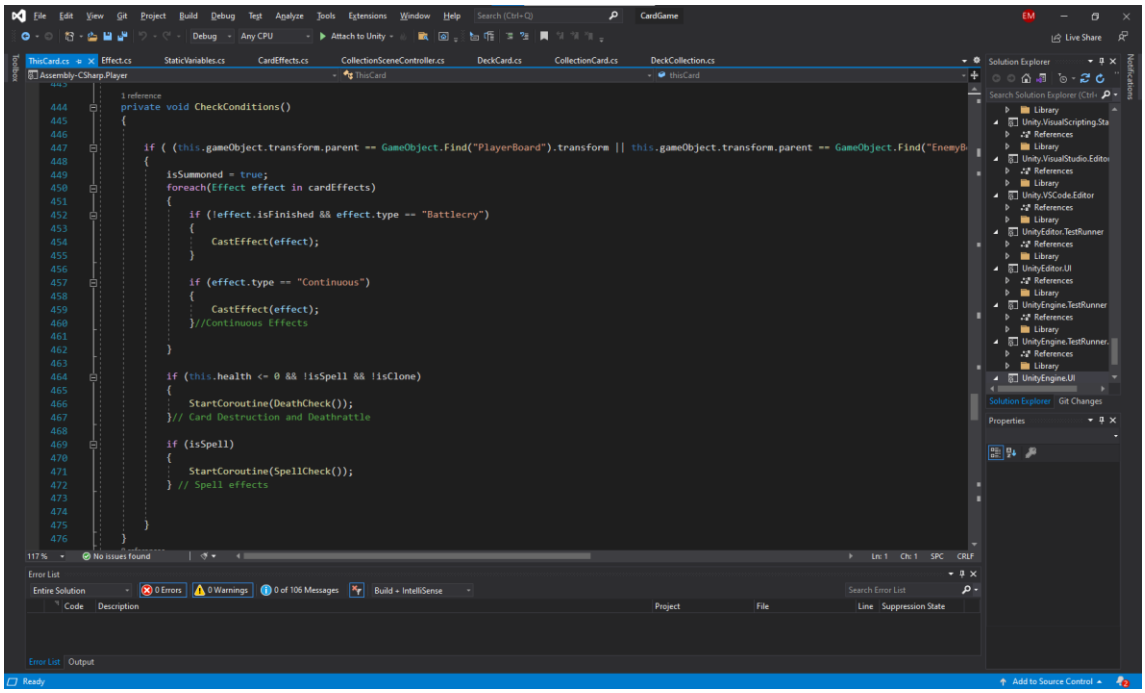
```
public class Effect
{
    public string effectName;
    public bool isFinished = false;
    public int attackMod = 0;
    public int healthMod = 0;
    public int costMod = 0;
    public string effectDescription = "";
    public string type;

    public Effect()
    {
        effectName = "";
        type = "Battlecry";
    }

    public Effect(string name, int costMod, int attackMod, int healthMod,
string type, string description)
    {
        effectName = name;
        this.attackMod = attackMod;
        this.healthMod = healthMod;
        this.costMod = costMod;
        this.type = type;
        effectDescription = description;
    }
}
```

Primjer koda 4.2. Effect klasa

Posebna skripta *ThisCard* je napravljena za kontrolu svih akcija karte. Prilikom inicijaliziranja objekta karte, on će poprimiti vrijednosti karte prema njihovoj poziciji u špilu. Što znači da u špilu nijedna karta nije još stvorena, već će se stvoriti tek kada se postavi u ruku, te prema uvjetu u čiju je ruku karta poslana određuje da li će se inicijalizirati karta na vrhu protivničkog ili igračevog špila.

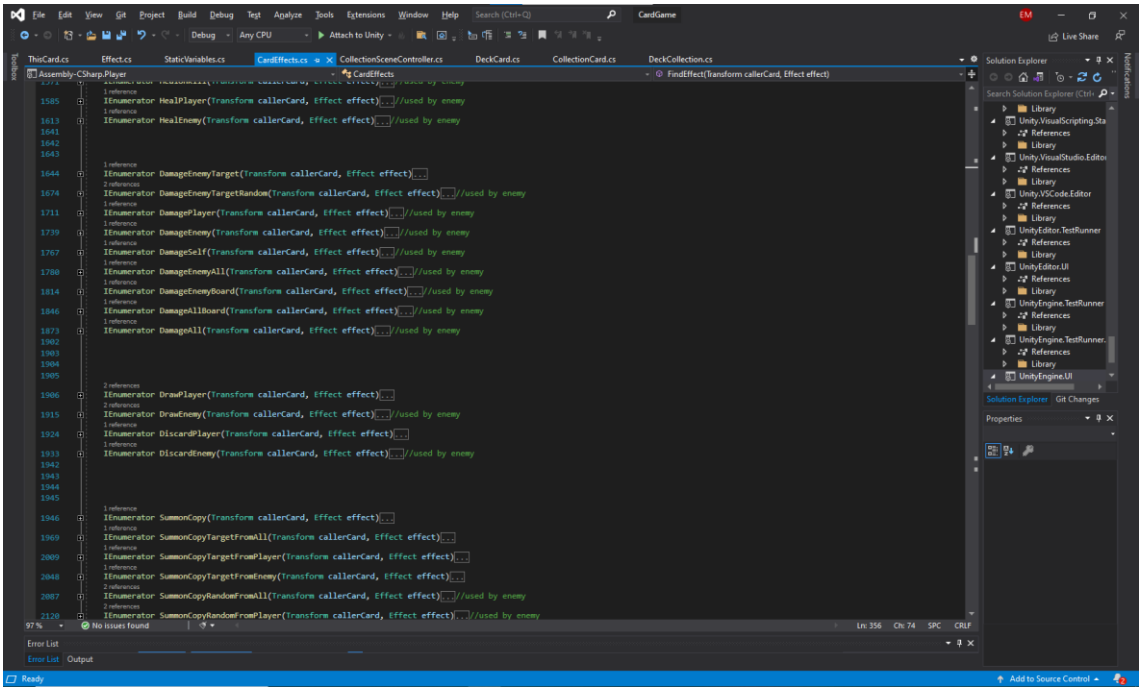
Karta prilikom prve inicijalizacije svrstava sve svoje efekte u različite liste ovisno o vrsti njihovog okidača, koji su za vrijeme pravljenja rada ograničeni na aktivaciju kad se karta odigra (engl. *Summon*), kad karta bude uništena (engl. *Death*), kad karta napadne (engl. *Attack*), kad karta primi štetu (engl. *OnDamage*) i uvijek aktivan, odnosno trajan (engl. *Continuous*). Nakon što ti efekti budu svrstani u svoje liste, *ThisCard* skripta provjerava svaki uvjet okidanja i onda šalje poziv

drugoj skripti, *CardEffects*, koja aktivira efekt prema predanim podatcima o efektu preko specifičnih funkcija napravljenih za svaki tip efekta.



Slika 4.2. Prikaz uvjeta za okidanje efekta

Efekti se moraju različiti po vrijednostima kojima utječu na kartu ili samom tipu efekta. Prema tome, stvorena je nekolicina tipova efekata koji su podijeljeni prema sličnosti rada u grupe. Imamo tip efekta koji modificira osnovne vrijednosti karte ili karata kao što su cijena pozivanja, napad i maksimalni život što ujedno modificira i trenutni život. Napravljen je i tip efekta koji karti ili kartama radi štetu, ujedno kao i tip efekta koji kartama liječi napravljenu štetu i povećava njihov trenutni život. Postoji još nekolicina tipova efekata kao što su izvlačenje karte, odbacivanje karte, uništavanje karte, kopiranje karte, vraćanje karte u ruku, uništavanje karte pod mnogim uvjetima, dobivanje trenutnih ili maksimalnih ECTS bodova, pozivanje drugih karta, te posebni tipovi efekata juriš (engl. *Charge*) i poruga (engl. *Taunt*). Juriš karti omogućava da napadne isti potez kad je pozvana, a poruga tjera protivnika da sa svojim sljedbenicima mora napasti tu kartu ili te karte ako više karata ima efekt poruge.



Slika 4.3. Neke of funkcija rada efekata koje su pozvane iz ThisCard skripte

4.3 Radnje karata

Unity omogućava dodavanje animacije karata putem svojeg Animate dodatka, ali zbog ograničenosti tog dodatka animacije izvlačenja karte i pokazivanja karte na nekoliko sekundi kad je karta odigrana animiranje nekih specifičnih animacija ćemo napraviti u sklopu koda.

Sve u svemu, animiranje određenih događaja ne pridonosi efikasnosti naše igre, ali opet može pomagati igraču u snalaženju u aplikaciji, lakšem praćenju radnji unutar aplikacije i povećanjem poliranja same aplikacije što joj daje ljepši izgled i osjećaj fluidnosti.

Neke od važnijih animacija koje će biti dodane u igru, a da su vezane za karte, su animacija izvlačenja karte iz špila, animacija pozivanja karte na teren i animacija odabira kojeg sljedbenika napasti ili označiti.

```
IEnumerator DrawAnimationPlayer()
{
    animatedCard.GetComponent<CanvasGroup>().blocksRaycasts = false;
    this.gameObject.GetComponent<CanvasGroup>().alpha = 0.0f;
    float thisManyTimes = 30;
    float scaleBy = 0.0f;
    for(int i = 0; i<thisManyTimes; i++)
    {
        if(i == (int)thisManyTimes/2)
        {
            animatedCard.GetComponent<ThisCard>().cardBack = false;
            animatedCard.transform.Rotate(0f, 180f, 0f);
        }
        animatedCard.transform.position = new
        Vector3(animatedCard.transform.position.x + moveByX,
        animatedCard.transform.position.y + moveByY, 0f);
        animatedCard.transform.Rotate(0f, 180f/thisManyTimes, 0f);
        animatedCard.transform.localScale = new Vector3(0.2f + scaleBy,
        0.2f + scaleBy, 0f);
        scaleBy += (0.4f / thisManyTimes);
        yield return new WaitForSeconds(0.01f);
    }
    yield return new WaitForSeconds(1.2f);
    thisManyTimes = 10;
    scaleBy = 0.0f;
    for (int i = 0; i < thisManyTimes; i++)
    {
        animatedCard.transform.position = new
        Vector3(animatedCard.transform.position.x + moveByX,
        animatedCard.transform.position.y + moveByY, 0f);
        animatedCard.transform.Rotate(0f, 0f,
        this.gameObject.transform.rotation.z / thisManyTimes);
        animatedCard.transform.localScale = new Vector3(0.6f - scaleBy,
        0.6f - scaleBy, 0f);
        scaleBy += (0.4f / thisManyTimes);
        yield return new WaitForSeconds(0.01f);
    }
    Destroy(animatedCard);
    this.gameObject.GetComponent<CanvasGroup>().alpha = 1.0f;
    yield return new WaitForSeconds(0.2f);
}
```

Primjer koda 4.3. Funkcija koja animira izvlačenje karte iz špila

Kod 4.3. odmah postavi inicijaliziranu kartu u ruku kako bi zauzela njeno mjesto, ali je napravi nevidljivom i ne interaktivnom kako je igrač ne bi mogao odigrati dok animacija ne završi. Nakon toga kod stvori identičnu kartu izvučenoj toj postepeno mijenja poziciju i veličinu dok ne dođe do fiksne određene pozicije gdje će stajati kratko vrijeme. Da bi se stvorila iluzija izvlačenja karte iz špila potrebno je karti postaviti izgled stražnje strane karte te je okrenuti koristeći *transform.Rotate*

metodu. Problem sa time je to što su ove karte 2D objekti, a ne 3D, pa rotiranjem njih 180 stupnjeva dobijemo samo zrcaljenu, obrnutu sliku. Zato koristimo mali trik da kada se karta okrene za 90 stupnjeva, baš onda kada se ne može vidjeti jer ne posjeduje dimenziju debljine, kartu zarotiramo za 180 stupnjeva i maknemo joj izgled stražnje strane karte. Tako će karta okretati samo jednu stranu sebe cijelo vrijeme, a stražnja strana koja je zrcaljena se zapravo nikada neće vidjeti.

Drugi dio ovoga koda koristi poziciju nevidljive karte u ruci kao fiksnu točku do koje naša, sada uvećana i okrenuta lažna karta mora doći. Taj korak je skoro pa isti kao prošli samo što kartu sada umanjujemo i ne moramo je rotirati. Nakon što je lažna karta došla do pozicije iste karte u ruci, pravu kartu činimo vidljivom, a lažnu kartu uništavamo jer više nije potrebna.

Sljedeći problem sa kartama predstavlja samo micanje karata na polje što je napravljeno preko skripte *Draggable* koja je dodana Prefab objektu karte i omogućava nam da ih povlačimo na teren. Velika pomoć tu su ugrađene metode za povlačenje objekata *OnBeginDrag*, *OnDrag* i *OnEndDrag*.

U kodu 3.4 je prikazan način rada i korištenja ovih metoda. Prilikom početka povlačenja karte, što predstavlja metoda *OnBeginDrag*, napravi se prazna nevidljiva pomoćna karta koja predstavlja mjesto u ruci gdje će se originalna karta vratiti ako je ne spustimo na polje, odnosno odigramo.

Metoda *OnDrag* koja se stalno poziva dok god vučemo kartu stalno našoj odabranoj karti poziciju postavlja na lokaciju miša.

I na samom kraju kada kartu prestanemo vući pozove se metoda *OnEndDrag*. Ta metoda gleda da li se naša karta nalazi iznad određenog mjesta za stavljanje karte, što je u ovom slučaju igračev dio borbene ploče. Ako se karta nalazi na mjestu za pozivanje, onda će se pomoćna karta izbrisati, a našoj karti za njezinog roditelja (engl. *Parent*) postaviti objekt koji drži karte na ploči što će i samu kartu postaviti na dobru poziciju na ploči za igru. U slučaju da je karta puštena iznad nekog neodobrenog mjesta, onda će se vratiti na mjesto pomoćne karte.

```
public void OnBeginDrag(PointerEventData eventData)
{
    placeholder = new GameObject();
    placeholder.transform.SetParent(this.transform.parent);
    this.transform.SetParent(this.transform.parent.parent);
    this.transform.rotation = Quaternion.identity;
}

public void OnDrag(PointerEventData eventData)
{
    this.transform.position = eventData.position;
    this.transform.rotation = Quaternion.identity;
}


public void OnEndDrag(PointerEventData eventData)
{
    this.transform.SetParent(parentToReturnTo);
    Destroy(placeholder);
}
```

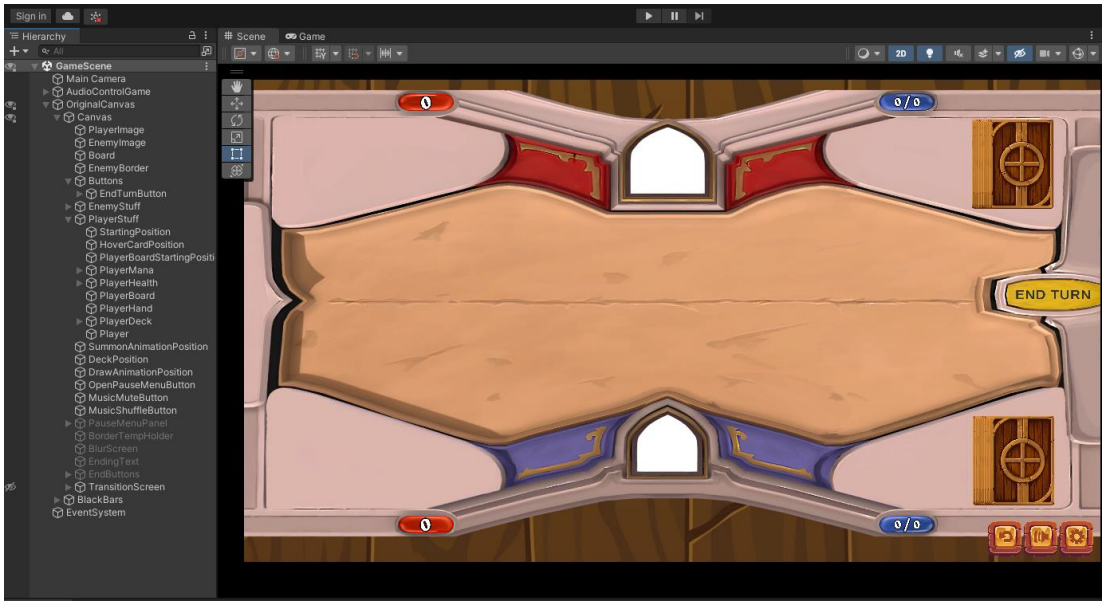
Primjer koda 4.4. Metode za povlačenje karte

4.4 Borbeni dio CCG-a

Glavni dio našeg CCG-a (engl. *Combat Card Game*) je sam dio borbe protiv protivnika. Tu se koriste izgrađeni špilovi, te odvija većina strategije. Cijelu borbenu fazu pravimo u njezinoj sceni. Scene u Unity-u predstavljaju određen pogled aplikacije odvojen od drugih scena napravljen za odvajanje različitih dijelova igre u zasebne segmente. Važno je napomenuti da prilikom učitavanja scene, svi njezini objekti i skripte se izbrišu i ponovno učitaju sa početnim vrijednostima.

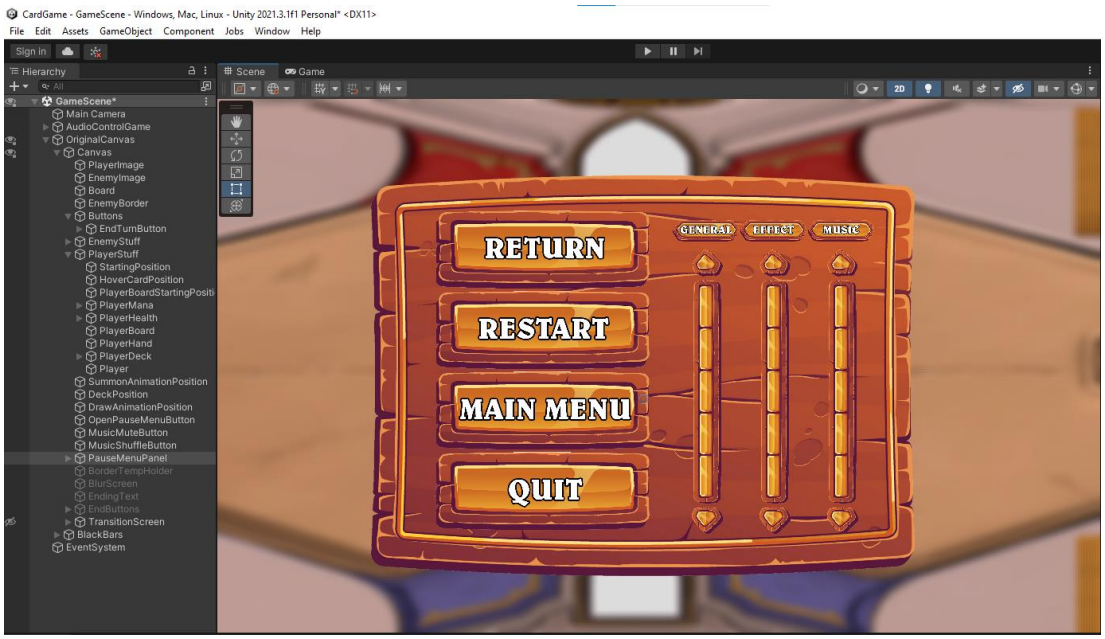
Borbeni dio igre mora sadržavati lagano preglednu borbenu mapu na kojoj se može raspoznati protivnikova ruka karata, igračeva ruka karata, protivnikov špil, igračev špil, protivnikov dio ploče za pozivanje karata, igračev dio ploče za pozivanje karata, dugme za završetak poteza, pomoćni dugmići za gašenje muzike, promjene muzike i otvaranje prozora za pauzu (engl. *Pause Menu*) koji u sebi ima opcije za kontrolu glasnoće zvuka, restarta, odlaska u glavni izbornik (engl. *Main Menu*) i izlaska iz igre.

Dodavanjem raznih objekata slika i raznih besplatnih grafičkih dodataka koje smo rasporedili u različite grupe preko veze roditelj-dijete koju Unity podržava dobili smo glavni izgled borbenog dijela.



Slika 4.4. Izgled borbenog dijela sa podjelom u grupe

Na lijevoj strani slike 4.4. vidimo kako su svi objekti raspoređeni po grupama, a neki od njih, specifično *PauseMenuPanel*, su blijeđe sive boje. To označava da su ti objekti deaktivirani, tj. oni su nevidljivi, nisu interaktivni, te njihove skripte se neće aktivirati dok god se oni sami ne aktiviraju, što se može uraditi preko koda. U ovom slučaju, *PauseMenuPanel* će se aktivirati tek kada korisnik pokuša otvoriti prozor za pauzu.



Slika 4.5. prozor za pauziranje

Također, na slici 4.4. blizu svakog kuta može se vidjet tekstni element. Na lijevoj strani je jedan broj koji predstavlja životne poene igrača i protivnika, a na desnoj strani se nalazi pratitelj ECTS bodova. Broj prije kose crte predstavlja trenutne ECTS bodove koje igrač ili protivnik posjeduje, a broj nakon kose crte predstavlja maksimalni broj ECTS bodova za trenutni potez. Ti ECTS bodovi se koriste za pozivanje sljedbenika ili odigravanje magija. Na početku igre svaki igrač ima jedan maksimalni broj ECTS bodova koji se povećava za jedan svaki potez.

Svaki potez kada se maksimalni broj ECTS bodova poveća, trenutni broj posjedovanih ECTS bodova se postavi na taj maksimum što omogućava njihovo ponovno korištenje. Napravljena je posebna skripta *TurnSystem* koja prati mijenjanje poteza, prepuštanje kontrole poteza, interakciju povećanja ECTS bodova opisanu gore, te još neke sitne mogućnosti.

Na kodu 4.5. ispod je prikaz kod za promjenu poteza i povećanje ECTS bodova, a na kodu 4.6 ispod koda 4.5 je metoda koja na početku igre dodjeljuje i igraču i protivniku 5 karata.

```
public void ChangeTurn()
{
    playerTurn = !playerTurn;
    if (playerTurn)
    {
        numberOfTurns++;
        StartCoroutine(Draw(1));
        playerMaxMana = playerMaxMana >= 10 ? 10 : playerMaxMana + 1;
        playerCurrMana = playerMaxMana;
    }
    else
    {
        StartCoroutine(DrawEnemy(1));
        enemyMaxMana = enemyMaxMana >= 10 ? 10 : enemyMaxMana + 1;
        enemyCurrMana = enemyMaxMana;
    }
    foreach (Transform card in allCards)
        {
            if (card.GetComponent<ThisCard>().isSummoned)
                if (card.GetComponent<ThisCard>().turnWhenSummoned !=
                    numberOfTurns)
                    card.GetComponent<ThisCard>().canAttack = true;
        }
}
```

Primjer koda 4.5. Metoda koja mijenja potez

```
IEnumerator StartGame()
{
    yield return new WaitForSeconds(0.01f);
    for (int i = 0; i < 5; i++)
    {
        yield return new WaitForSeconds(0.5f);
        AudioControl.PlayDrawAudio();
        Instantiate(CardInHand, PlayerHand.transform);
        AudioControl.PlayDrawAudio();
        Instantiate(CardInHand, EnemyHand.transform);
    }
}
```

Primjer koda 4.6. Dijeljenje početne ruke karata na početku igre

Kod CCG igara postoji mehanika koja sprječava korisnika da napadne sa sljedbenikom koji je tek pozvan na teren. Ta mehanika je poznata kao bolest pri prizivanju (engl. *Summoning Sickness*) koja forsira pozivatelja karte da čeka jedan potez prije nego može napasti sa kartom. To je napravljeno kako bi protivnik imao vremena da reagira na tu kartu tijekom svojeg poteza. Postoji poseban tip efekta zvan juriš (engl. *Charge*) koji karti daje mogućnost napada na istom potezu kada je ta karta pozvana, naravno sa nekom posljedicom kao što je manji napad ili život u usporedbi sa cijenom karte, neki negativni dodatni efekt, ili jednostavno, veća cijena karte.

U kodu 4.5. ima dodatni dio na dnu, poslije izmjene ECTS bodova koji gleda sve pozvane sljedbenike, te provjera da li je njihov napad veći od nule i da li je trenutni potez različit od poteza kad su pozvane. U slučaju da su ta oba uvjeta istinita, skripta omogućuje tom sljedbeniku da napadne ovaj potez.

Što se tiče same borbe sljedbenika, potrebno je odrediti koje mehanike, te koji pristup borbi želimo imati. U slučaju ovog završnog rada odlučili smo koristiti najčešća pravila u kojima igrač može sa svojim sljedbenicima napasti protivnika izravno imao li protivnik svoje sljedbenike ili ne, osim u veoma specifičnom uvjetu kada neka protivnikova karta posjeduje Taunt efekt. Kada se vodi bitka između sljedbenika, obadva sljedbenika primaju štetu jednaku napadu druge karte. Taj dio ćemo napraviti koristeći kod 4.7.

```
if (isAttacking && card.transform.parent == EnemyBoard.transform)
{
    isAttacking = false;
    canAttack = false;
    target = card;

    retaliationDamage = card.GetComponent<ThisCard>().attack;

    card.GetComponent<ThisCard>().health -=
    gameObject.GetComponent<ThisCard>().attack;
    gameObject.GetComponent<ThisCard>().health -= retaliationDamage;

    foreach (Effect effect in attackEffects)
        CastEffect(effect);
 }
```

Primjer koda 4.7. Prikaz koda kada igračev sljedbenik napada protivnikovog

U kodu 4.7. jasno je vidljivo kako nakon napada, karti se canAttack varijabla postavi na false što znači da nakon što karta jednom napadne, ona ne može napasti opet. Vrijednost štete koju će karta primiti od protivnikove karte spremamo u varijablu prije samog napada zbog mogućnosti promjene napadne vrijednosti protivnikove karte zbog primanja štete.

Nakon toga protivnikovoj karti smanjimo život za vrijednost jednaku napadu naše napadačke karte, a našoj karti smanjiti život za spremljenu vrijednost koja predstavlja napad karte koju napadamo prije momenta napada.

Na samom kraju ove funkcije pošto se desio napad, našoj karti aktiviramo okidač za efekte koji se aktiviraju prilikom napada. To radimo na način da prođemo kroz svaki takav efekt koji smo ranije spremili u posebnu listu, te ih aktiviramo pozivom metode *CastEffect* opisane u ranijim poglavljima.

4.4.1 Ponašanje protivnika

Kako bi aplikacija imala interaktivno igranje, potrebno je napraviti protivnika koji bi trebao znati koje karte ima u ruci, kako ih najbolje odigrati, znati stanje na borbenoj ploči, odlučivati kada da napadne protivnika direktno a kada njegove sljedbenike, najefikasniji redoslijed napadanja sljedbenika, te odlučivanje kojeg protivničkog sljedbenika napasti.

Pravljenje prave umjetne inteligencije, te sa time i AI (Umjetna inteligencija, engl. *Articifial Intelligence*) protivnika nije jednostavno, te u ovom trenutku za ovaj završni niti moguće, ali zato

korištenjem nekolicine algoritama i predodređenih ideja o tome šta zapravo potez čini dobrim odnosno pametnim možemo napraviti lažnu umjetnu inteligenciju, poznatu kao glupi AI (engl. *Dumb AI*) koja će za ovu aplikaciju biti sasvim dovoljna.

Napravili smo skriptu *EnemyBehaviour* u kojoj je definirana metoda pokazana na kodu 4.8. koja predstavlja jedan protivnikov potez.

```
IEnumerator EnemyTurn()
{
    List<Transform> cardsInHand = new List<Transform>();
    List<Transform> cardsOnBoard = new List<Transform>();

    foreach (Transform card in EnemyHand.transform)
    {
        if (card.GetComponent<ThisCard>().cost <= TurnSystem.enemyCurrMana)
            cardsInHand.Add(card);
    }

    SortList(cardsInHand);

    foreach (Transform card in EnemyBoard.transform)
    {
        cardsOnBoard.Add(card);
    }

    //SUMMON BEHAVIOUR
    yield return StartCoroutine(SummonCards());

    yield return new WaitForSeconds(1.5f);
    //ATACK BEHAVIOUR
    yield return StartCoroutine(AttackPhase());


    // END TURN
    systemManager.GetComponent<TurnSystem>().ChangeTurn();
}
```

Primjer koda 4.8. Glavno ponašanje protivnika

Protivniku na samom početku dajemo dvije liste koje predstavljaju njegove karte u ruci i njegove karte na polju da sa njima može rukovati. Postavili smo metode na takav način da će protivnik prvo pokušati pozvati najveći broj karata iz ruke koji može pozvati preko metode *SummonCards*. Ta metoda je prevelika da bi se stavila u ovaj završni rad, ali njeno ponašanje se može objasniti.

SummonCards radi na način da napravi glavnu listu karata iz ruke te iz nje izbaci sve karte koje imaju veću cijenu pozivanja nego što protivnik ima (u ovom slučaju protivnik se odnosi na samog protivnika, a ne igrača). Glavno razmišljanje koje dajemo našem glupom AI-u je da iskoristi što više

ECTS bodova, sljedeće po listi važnosti je da odigra najskuplje karte koje može i na samom kraju naše liste zahtjeva je da sljedbenici imaju više važnosti, odnosno prednost za pozivanjem nego magije. Taj zadnji uvjet je napravljen zato jer mnoge magije imaju efekt da ojačaju druge karte.

Lažna umjetna inteligencija samo ima informacije o osnovnoj karti, kao što su cijena pozivanja, napad i životni bodovi, a ne o njezinim efektima. Ta ograničenost nam olakšava posao određivanja koju kartu će glupi AI odigrati, jer njega zanima samo cijena pozivanja karte, a ne šta ta karta radi. Nažalost, zbog te ograničenosti naša lažna umjetna inteligencija će praviti pogreške koje običan čovjek ne bi pravio, na primjer protivnik koji ima praznu borbenu ploču bi mogao pozvati sljedbenika koji ojačava drugog sljedbenika i tek nakon toga pozvati drugog sljedbenika, na kraju potpuno izgubiti taj efekt ojačavanja. Moguće je i također da protivnik odigra magiju koja uništi igračevog sljedbenika iako igrač nema nijednog sljedbenika i na taj način samo potrošiti tu kartu i svoje ECTS bodove, a ne dobiti ništa zauzvrat. Pošto ova aplikacije služi samo kao prikaz onoga šta se može odraditi sa jednostavnim znanjem Unity-a i C# programskog jezika, takve greške, odnosno mane, su u potpunosti prihvatljive.

Sada kada znamo način na koji će naša lažna umjetna inteligencija odigrati karte samo moramo to implementirati. Naš algoritam će listu karata u ruci poredati od najskuplje karte do najjeftinije u odnosu na cijenu pozivanju. Onda će redom prolaziti kroz tu listu dodajući u drugu listu kartu ako je može pozvati te povećavajući ukupnu cijenu pozivanja svih karata te liste. Nakon prolaska kroz cijelu listu ukupna cijena pozivanja se usporedi sa najvećom cijenom pozivanja, gdje u slučaju da je ukupna cijena pozivanja veća od prijašnjeg maksimuma, ukupna cijena pozivanja postane novi maksimum. U slučaju da je ukupna cijena pozivanja jednaka raspoloživim ECTS bodovima iz petlji se odmah izlazi jer je najbolja kombinacija prema našim uvjetima pronađena.

U sljedećem koraku se iz glavne liste miču sve karte sa najvećom cijenom pozivanja i cijeli postupak se ponavlja. Tako će se ponoviti za svaku kombinaciju cijena pozivanja glavne liste.

Kada se prođe kroz sve kombinacije i nađe lista najbolje kombinacije karata za pozivanje uzima se prvi element liste, najskuplja karta, i postavlja se na borbeni dio ploče za igranje protivnika oduzimajući njezinu cijenu pozivanja od trenutnog broja ECTS bodova protivnika.

Razlog zašto pozivamo samo jednu kartu je da na barem ovaj način pokušamo smanjiti nedostatak neznanja efekata lažnog AI-a. Postoji šansa da karta koju pozove smanji cijenu pozivanja ostalih karata ili na neki način utječe na nju, bilo to na pozitivan ili negativan način. Možda karta utječe na

neku drugu kartu čiji efekt onda utječe na cijenu pozivanja protivnikovih karata. Zato pozivamo samo prvu kartu i nakon toga cijeli postupak pozivanja ponavljamo. Tako će se ovaj postupak ponavljati dok god protivnik nema karata u ruci koje može pozvati sa trenutnim brojem ECTS bodova.

```
if (cardToSummon != null)
{
    AudioControl.PlayPlayAudio();
    TurnSystem.enemyCurrMana -= cardToSummon.GetComponent<ThisCard>().cost;
    cardToSummon.GetComponent<ThisCard>().cardBack = false;
    cardToSummon.SetParent(EnemyBoard.transform);
    ShuffleHand();
}
```

Primjer koda 4.9. Pozivanje odabrane protivničke karte na borbeni dio ploče za igranje

Nakon svega toga, sljedeći korak je da preko metode *AttackPhase* odredimo kojim redoslijedom i koju kombinaciju igračevih karata će protivnikove karte napasti.

Prva stvar koja je potrebna za određivanje cijele napadačke faze je formula efikasnosti poteza. To je formula koju će svaka kombinacija koristiti kako bi odredila efikasnost poteza i našla najbolji potez protivnika. Ova formula je zbroj protivnikove prijetnje (engl. *ThreatPossesed*) i maknute igračeve prijetnje (engl. *ThreatRemoved*). Obadvije ove varijable imaju svoju formulu i svoj kod za računanje te formule.

```
ThreatRemoved = 0;
ThreatPossesed = 0;
EnemyThreat = 0;
//ThreatPossesed Calculator
foreach (int attacker in attackers)
{
    ThreatPossesed +=
    cardsThatCanAttack[attacker].GetComponent<ThisCard>().health;
    ThreatPossesed +=
    cardsThatCanAttack[attacker].GetComponent<ThisCard>().attack;
    ThreatPossesed -=
    cardsThatCanAttack[attacker].GetComponent<ThisCard>().attackMod;
    ThreatPossesed -=
    cardsThatCanAttack[attacker].GetComponent<ThisCard>().healthMod;
    ThreatPossesed++;
}
```

Primjer koda 4.10. Računanje posjedovane prijetnje protivnika

```
//EnemyThreat Calculator
foreach (int target in targets)
{
    EnemyThreat += availableTargets[target].GetComponent<ThisCard>().health;
    EnemyThreat += availableTargets[target].GetComponent<ThisCard>().attack;
    EnemyThreat -=
    availableTargets[target].GetComponent<ThisCard>().attackMod;
    EnemyThreat -=
    availableTargets[target].GetComponent<ThisCard>().healthMod;
    EnemyThreat++;
}
```

Primjer koda 4.11. Računanje posjedovane prijetnje igrača

Obadvije prijetnje se računaju kao zbroj prijetnje svake karte koji je jednak zbroju životnih bodova karte, napada karte, vanjskih modifikatora napada karte, te još dodatni bod prijetnje koji svaka karta ima zbog lakše kalkulacije. Prilikom računice svake kombinacije, za svaki životni bod koji se oduzme od bilo koje karte, prikladna prijetnja se promjeni za tu vrijednost. U slučaju da životni bodovi karte padnu ispod nule, odnosno da je karta uništena, prikladna prijetnja se mijenja i za njezine napadačke bodove, te još za dodatni bod prijetnje koji svaka karta ima kako bi se predstavilo da je karta živa, ili ovom slučaju, mrtva.

Pošto u borbenoj fazi uopće nije potrebna informacija o cijeni karte već njezina prijetnja, te još dodatni izbor koju kartu napasti, način na koji prolazimo koja karta će napasti kojim redoslijedom, te koju kartu se malo mijenja. Sada prolazimo kroz sve moguće permutacije napadača i onda sa svakom od tih permutacija kroz svaku permutaciju meti napada. Metoda koja vraća sve ove permutacije kao liste listi je prikazana kodom dolje.

```
GetPermutations<T>(IEnumerable<T> list, int length)
{
    if (length == 1)
        return list.Select(t => new T[] { t });

    return GetPermutations(list, length - 1)
        .SelectMany(t => list.Where(o => !t.Contains(o)),
            (t1, t2) => t1.Concat(new T[] { t2 }));
}
```

Primjer koda 4.12. Metoda koja vraća sve permutacije kao listu listi općeg tipa

Na kraju protivnikovog poteza, naš lažni AI poziva funkciju iz skripte *TurnSystem* za promjenu poteza i prepuštanja kontrole nazad igraču, nakon čega igrač odrađuje svoj potez, klikne dugme za završetak poteza i cijeli postupak se ponavlja dok god ili protivnik ili igrač ne padne ispod nula životnih poena nakon čega igra završava.

4.5 Scena kolekcije

Nakon programiranja borbenog dijela aplikacije prelazimo na pravljenje kolekcije karta i špilova koje koriste igrač i protivnik. U ovom dijelu programiranja i dizajniranja fokusiramo se na pravljenje sučelja preko kojega će korisnik moći proći pregledati sve karte koje se trenutno nalaze u igri i uređivati špilove.

Naravno, prije samog rada, potrebno je postaviti neka osnovna pravila i ideju kako će ovaj dio raditi išta će korisnik u njemu moći da radi. Korisnik će imati šest špilova karata kojima može mijenjati cijelu strukturu karata, bilo to dodavanjem karta ili micanjem njih iz špila. Glavna pravila špila da bude ispravan je to da on mora imati točno trideset karata inače se on neće moći koristiti u borbenom dijelu, te da u špilu ne smije biti više od tri kopije iste karte. To pravilo je često u raznim TGC i CCG aplikacija, te je uvedeno zbog raznolikosti špilova i balansiranju snage špila. Također korisnik bi tijekom pravljenja špila trebao imati mogućnost odbacivanja promjena napravljenih u špilu i vraćanje špila na zadnje spremljeno stanje, a naravno i mogućnost da spremi špil koji je napravio ili koji je u procesu pravljenja.

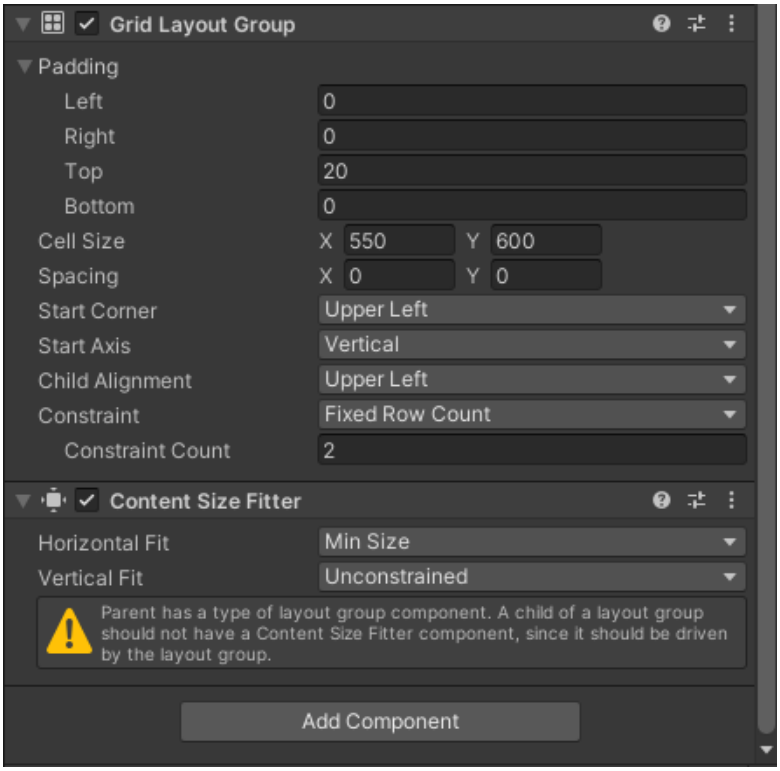
Prije programiranja prvo dolazi uređivanje i izgled preko Unity Game Engine-a. Kako bi odvojili ovaj dio aplikacije od borbenog dijela, te s time i ubrzali brzinu aplikacije zbog smanjenja objekata i učitanih skripti, ovaj kolekcijski dio ćemo odvojiti u svoju zasebnu scenu. Veoma slično kao i za borbeni dio, napraviti ćemo razne objekte i elemente teksta, slika i dugmića koji će našu praznu scenu popuniti sa privlačnim sučeljem koje je lako razumljivo i čitljivo. Naravno, sve elemente smo rasporedili po određenim vezama roditelj-dijete kako bi programerski rad sa njima bio jednostavniji i efektivniji.

Slika 4.6. Izgled scene za kolekciju karata i uređivanje špilova

4.5.1 Kolekcija karata

Sve karte, odnosno našu kolekciju, ćemo korisniku prikazati u središnjem pravokutnom prozoru naše scene vidljive na slici 4.6. Pošto u našoj igri možemo imati puno više karata nego što može stati na naš pravokutni prozor za kolekciju, odsada zvan samo kolekcija, moramo omogućiti korisniku da nekako pregleda cijelu kolekciju koliko god ona velika bila. Taj problem ćemo riješiti koristeći posebne vrste objekata sa već napravljenim skriptama. Potrebni posebni objekti koje ćemo mi koristiti su *ScrollRect*, objekt koji omogućava da se njegov sadržaj miče, *Mask*, objekt koji sakriva svaki element izvan svojeg određenog prostora, *ContentSizeFitter*, objekt koji automatski širi granice prostora kako bi u njih stali njegova djeca (engl. *Children Objects*), *GridLayoutGroup*, objekt koji svoju djecu postavlja u ćelije jednake veličine posložene u rešetku i *Slider*, pomoćni objekt *ScrollRect*-a koji služi kao alternativni način micanja kroz njegov prostor.

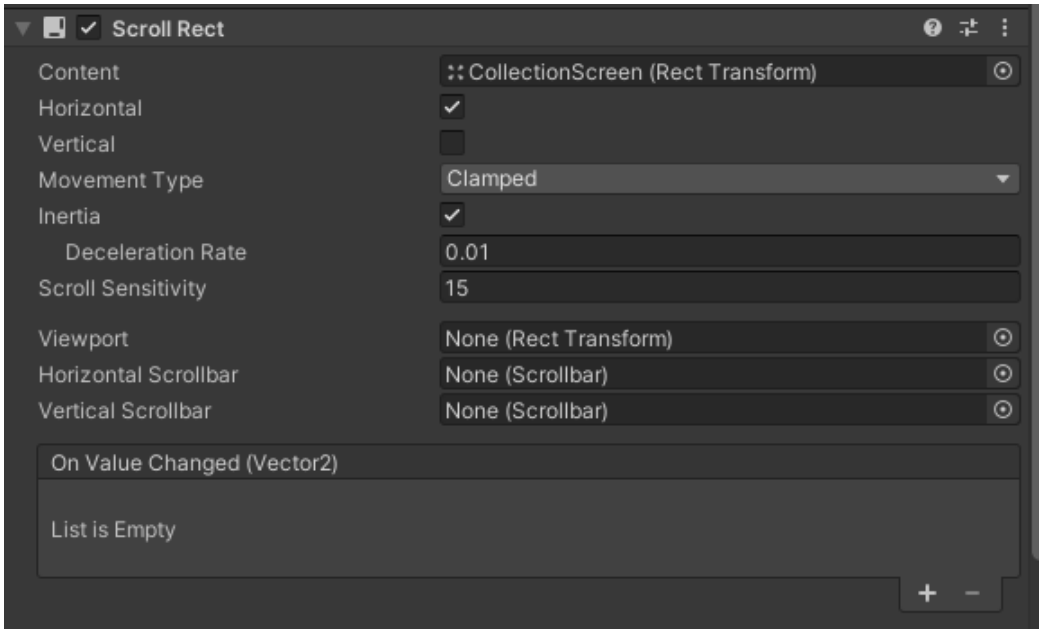
Nakon što smo definirali prostor u kojem će se naša kolekcija nalaziti, potrebno je koristeći *GridLayoutGroup* i *ContentSizeFitter* objekte odrediti pravila i ograničenja po kojima će se naše karte slagati u kolekciju. Na slici 4.7 su prikazane postavke ovih posebnih objekata. Što se tiče *ContentSizeFitter*-a, jedina stvar koju smo morali njemu reći je da se širi horizontalno, jer to je smjer u kojem će naše karte biti poredane, preko opcije MinSize.



Slika 4.7. Postavke GridLayoutGroup i ContentSizeFitter komponenti

Sa *GridLayoutGroup* komponentom morali smo odrediti veličinu jedne ćelije koju smo postavili na veličinu obične karte u pikselima, povećanu za još dodatnih 50 piksela u svakom smjeru kako bi imali razmak između karata koje se nalaze u ćelijama. Kao početni kut odakle će se djeca, koju će predstavljati karte, krenuti slagati smo postavili gornji lijevi kut prostora kolekcije. Pošto želimo da naše karte budu poslagane po cijeni pozivanja, od najmanje do najveće, a ujedno želimo i da naše karte budu postavljane od lijevo prema desno, moramo promijeniti i druge opcije ove komponente. Postavljanjem StartAxis, odnosno početne osi da bude vertikalna naše karte će prvo popuniti svaki red, a tek onda ići na popunjavanje sljedećeg stupca. Da bi to radilo, morali smo postaviti i ograničenje da ova grupa može imati samo dva reda zbog bolje preglednosti kolekcije.

Također smo postavili *Mask* komponentu na veličinu našeg *ScrollRect* dijela kako bi sakrili svaku kartu koja nije u našem željenom području. Jedina stvar koja nam je ostala je objasniti i postaviti željene postavke *ScrollRect* komponente.



Slika 4.8. Gotove postavke ScrollRect komponente

Prva stvar koju smo morali odraditi je odrediti koji zapravo objekt ćemo moći micati, te ga ubaciti u Sadržajni (engl. *Content*) dio komponente. Uobičajeno ScrollRect omogućava da se željeni objekt može micati i horizontalno i vertikalno, ali za naš slučaj potrebno nam je samo da se može micati horizontalno, što znači da ćemo isključiti mogućnost micanja vertikalno.

Tip kretnje (engl. *Movement Type*) komponente je raspodijeljen u tri vrste; *Clamped*, *Elastic* i *Unrestricted*. Svaki taj tip predstavlja šta se našem objektu dešava kada ga pokušamo povući izvan linija sadržaja. Kod bez ograničenog (engl. *Unrestricted*) tipa naš sadržaj će se moći povući koliko god želimo; kod elastičnog (engl. *Elastic*) tipa sadržaj možemo pomaknuti samo malo izvan određenog prostora unutar kojeg se sadržaj nalazi te se nakon toga vraća nazad kao da je sam prostor elastičan. Nama je potreban stegnuti (engl. *Clamped*) tip kretnje, koji ne dopušta da se sadržaj miče izvan svojih granica.

Sitni detalji kao što je inercija (engl. *Inertia*) i osjetljivost micanja (engl. *Scroll Sensitivity*) upravljaju sa brzinom stajanja i kretnje micanja sadržaja, te nisu toliko bitni, pa smo ih ostavili na njihovim početnim vrijednostima.

4.5.2 Uređivanje špila

Sada kada je kolekcija gotova i karte u njoj popunjene, trebamo napraviti dio za pravljenje i uređivanje špila. Taj dio smo napravili na desnoj strani kolekcijske scene kao što se može vidjeti na slikama 4.6 i 4.9. Tu smo vertikalno poredali dugmiće koji predstavljaju svaki od šest špilova koje igrač posjeduje. Kada korisnik klikne na jedan od špilova dugmiće špilova sakrijemo i aktiviramo objekt koji predstavlja izbornik za uređivanje špila na istom mjestu gdje su se nalazila dugmad špilova.

Slika 4.9. Promjena objekta izbora špila u objekt uređivanja špila

Kada je otvoren izbornik za uređivanje špila, on se popuni kartama koje su bile u odabranom špilu, ako ih je uopće bilo. Korisnik bi trebao moći klikom na kartu iz kolekciju tu istu kartu dodati u špil. Pošto su karte prevelike da bi se trideset njih stavilo u špil, napraviti ćemo još jedan Unity Prefab objekt *DeckCard* koji će predstavljati kartu u špilu i imati će sve potrebne informacije o karti bez da prenapučuje naš izbornik za uređivanje špila.

Način na koji ćemo napraviti ovaj Prefab objekt je sličan kao i način na koji smo napravili *ThisCard* objekt iz drugog poglavlja. Prvo ćemo u samom kodu napraviti klasu koja će predstavljati objekt karte u špilu sa svim potrebnim informacijama.

```
public class DeckCard : MonoBehaviour, IPointerEnterHandler,
                        IPointerExitHandler
{
    public int cost;
    public TextMeshProUGUI manaText;
    public TextMeshProUGUI nameText;
    public TextMeshProUGUI amountOfInDeckText;
    public int amountInDeck;
    public int id;
    public Transform hoverCardPosition;
    public GameObject AmountImage;
    public Image cardImage;
    public GameObject hoverCard;

    ...
}
```

Primjer koda 4.13. Klasa koja predstavlja kartu u špilu

Za *DeckCard* klasu koja predstavlja objekt karte u špilu smo napravili i grafički izgled koristeći elemente grafičkog sučelja. U samoj klasi smo povezali grafički i programerski dio karte u špilu tako što smo odmah u klasi definirali tekstualne objekte koji će predstavljati cijenu karte, naziv karte i samu kartu iz kolekcije. Same minimalne informacije o karti koje naš Prefab objekt ima, cijena, naziv i slika, nisu dovoljne da kažu korisniku sve informacije o karti.

Kao programerski dizajneri ne možemo očekivati od korisnika da se sjeća svih karata i svakog efekta svih karata samo prema nazivu, slici i cijeni karte. Zbog tog razloga u našu *DeckCard* klasu dodajemo i potpunu kartu iz kolekcije sa svim svojim informacijama i izgledom. Poziciju te karte postavljamo na istoj visini kao i objekt karte u špilu, ali sa desne karte kako ne bi smetala ostatku karata iz špila. Također tu kartu činimo nevidljivom dok god korisnik ne prijeđe mišom preko nje i isto tako je čini nevidljivom kada korisnik prestane držati miš iznad nje.

Te interakcije su nam omogućene preko specifičnih sučelja *IPointerEnterHandler* i *IPointerExitHandler* koje naša klasa nasljeđuje. Ta sučelja nam daju dvije metode koje se aktiviraju kada miš prijeđe preko našeg objekta i kada miš prestane biti iznad našeg objekta i njihov rad je prikazan kodom 4.14.

```
public void OnPointerEnter(PointerEventData eventData)
{
    hoverCard.SetActive(true);
}
public void OnPointerExit(PointerEventData eventData)
{
    hoverCard.SetActive(false);
}
```

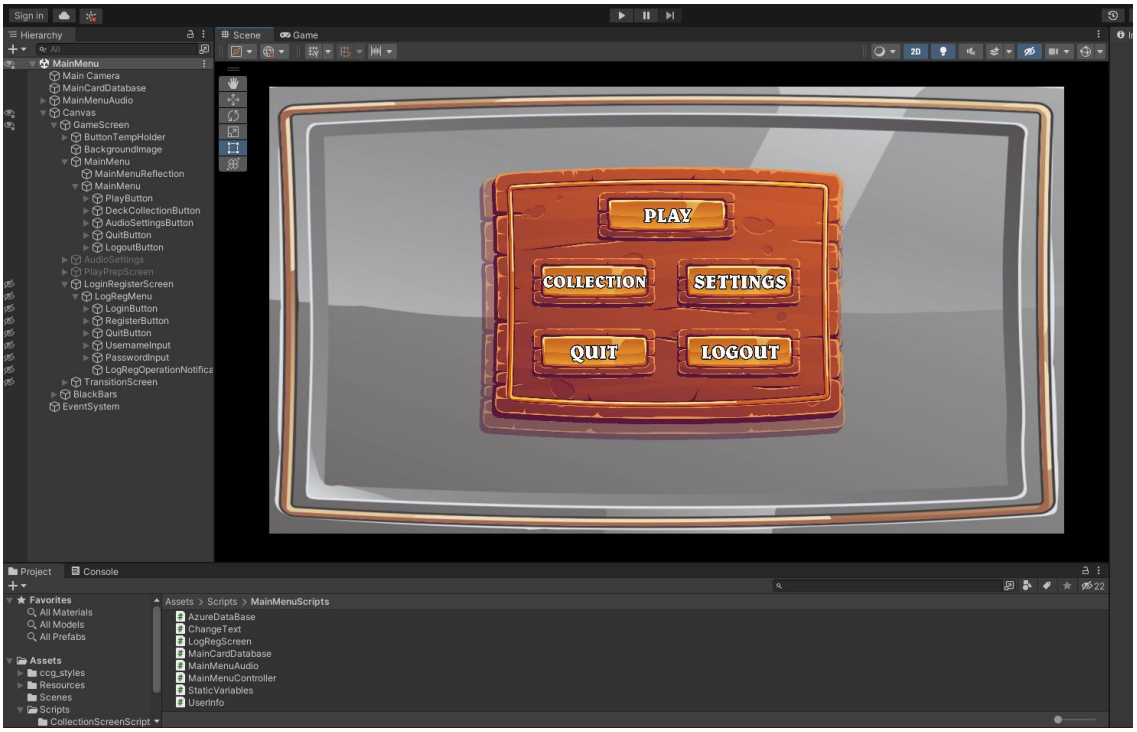
Primjer koda 4.14. Naslijeđene metode za pokazivanje karte iz kolekcije

4.6 Glavni izbornik

Naše scene kolekcije i borbenog dijela aplikacije su gotove, ali nisu povezane jedna sa drugom. To ćemo napraviti preko često rješenja ovih problema, koristeći glavni izbornik, neko centralno sučelje koje će dočekati korisnika prilikom paljenja igre, te preko kojega će korisnik moći da odabere točno što želi da uradi nakon što je pokrenuo igru, bilo to odmah ići u borbeni dio, otići na kolekciju da napravi špil ili samo pogleda karte, promjeni nivo glasnoće zvuka u aplikaciji ili samo ugasi aplikaciju.

Kao i za prijašnja dva potpoglavlja, odnosno dijelove aplikacije, i ovaj dio aplikacije će biti napravljen u svojoj odvojenoj sceni koja će ujedno i objediniti ostale scene. Korisniku se mora moći pružiti opcija odlaska na borbeni dio aplikacije gdje prije samog ulaska u borbeni dio korisniku se ponudi koje špilove želi koristiti, opcija odlaska na kolekcijski dio aplikacije, opcija otvaranja pomoćnog prozora za kontrolu glasnoće zvuka, opcija da korisnik izađe iz igre, i na kraju opcija da se korisnik odjavi čija će funkcionalnost biti određena u sljedećeg poglavlju kada se uvede veza između baze podataka i unutrašnjeg dijela aplikacije.

Ovaj izbornik ne treba izgledati previše komplicirano zato jer je to prva stvar koju će korisnik vidjeti kada upali aplikaciju. Ali samo zato jer naša izbornik ne treba izgledati komplicirano, ne znači da on ne mora imati komplicirane mogućnosti. Dodati ćemo posebnu opciju da kada korisnik klikne na dugme za igru (engl. *Play Button*), otvori se novi izbornik gdje korisnik ima opciju izabrati svoj špil koji želi koristiti u borbenom dijelu igre, te i protivnikov špil koji će protivnik koristiti tijekom borbenog dijela igre.



Slika 4.10. Sučelje glavnog izbornika

Koristeći grafičke elemente, omeđene objekte i interaktivne elemente kao što su dugmad (engl. *Interactive Button Element*) napravili smo izgled našeg glavnog izbornika kao što je prikazano na slici 4.10. koji je lako razumljiv.

Kao što je rečeno ranije, potrebno je implementirati dodatni izbornik za odabir špila korisnika i protivnika koji se otvori pritiskom na *Play* dugme. Zato smo napravili izgled izbornika za biranje špilova i deaktivirali ga kako bi on bio skriven od korisnika dok god on ne klikne na *Play* dugme.

```
public void OpenPlayPrepMenu()
{
    PlayPrepMenu.SetActive(true);
    if (StaticVariables.deck1.Count == 30)
    {
        deck1Button.transform.SetParent(PlayerDeckSelect.transform);
        EnemyDeck1Button.transform.SetParent(EnemyDeckSelect.transform);
    }
    if (StaticVariables.deck2.Count == 30) {...}
    if (StaticVariables.deck3.Count == 30) {...}
    if (StaticVariables.deck4.Count == 30) {...}
    if (StaticVariables.deck5.Count == 30) {...}
    if (StaticVariables.deck6.Count == 30) {...}
}
```

Primjer koda 4.15. Radnja Play dugmeta

Problem u vezi korištenja scena je to što se skripte, a ujedno time i varijable, scena koje nisu učitane deaktiviraju, pa varijable skripti iz učitane scene ne možemo direktno poslati u drugu ne inicijaliziranu scenu. Za rješavanje tog problema pravimo jedinstvenu statičnu skriptu *StaticVariables* koja se nalazi u naši resursima, ali ne u nekoj sceni. Ta skripta se instancira samo kod pokretanja aplikacije a ostane učitana dok god se aplikacija ne ugasi. Zbog toga, naše varijable koje želimo prenijeti u skripte iz drugih scena šaljemo koristeći *StaticVariables* skriptu kao medijatora, posrednika.

Za svaki od šest špilova koje korisnik može napraviti imamo posebno dugme klikom na koje se za borbeni dio igre odabere odgovarajući špil. Ta dugmad su skrivena dok god njihovi odgovarajući špilovi ne budu spremni za borbeni dio, odnosno, dok god ti špilovi ne budu imali točno trideset karti.

Prijelaz iz jedne scene u drugu je omogućen zbog posebne Unity metode koji učitava novu scenu, a deaktivira sadašnju. Scena koje se učita je odabrana preko imena scene ili preko indeksa scene koji se može namjestiti u samom Unity uređivaču.

Kako bi se aplikacija dodatno uljepšala, a ujedno i kako bi se sakrilo učitavanje svih elemenata scene, na svaku scenu je dodan prozor izmjene (engl. *Transition Screen*) koji se sastoji od jedne ploče podijeljene na 4 jednako velika dijela. Prilikom učitavanja scene taj prozor izmjene je postavljen kao jedina stvar koja se vidi na toj sceni, te nakon što se svi objekti i skripte scene učitaju, taj prozor izmjene se ovisno o sceni otvori vertikalno ili horizontalno. To je napravljeno na način da svaki dio prozora za izmjenu ima svoje funkcije pomaka koje ga ili postave nazad na početnu poziciju ili odmaknu od nje izvan prostora koji korisnik može vidjeti.

```
IEnumerator MoveLeftOpen(GameObject part)
{
    float moveBy = (positionLeftTop.position.x -
    part.transform.position.x)  / 50f;
    for(int i=0; i < 50; i++)
    {
        part.transform.position = new Vector2(part.transform.position.x +
        moveBy, part.transform.position.y);
        yield return new WaitForSecondsRealtime(0.01f);
    }
}
```

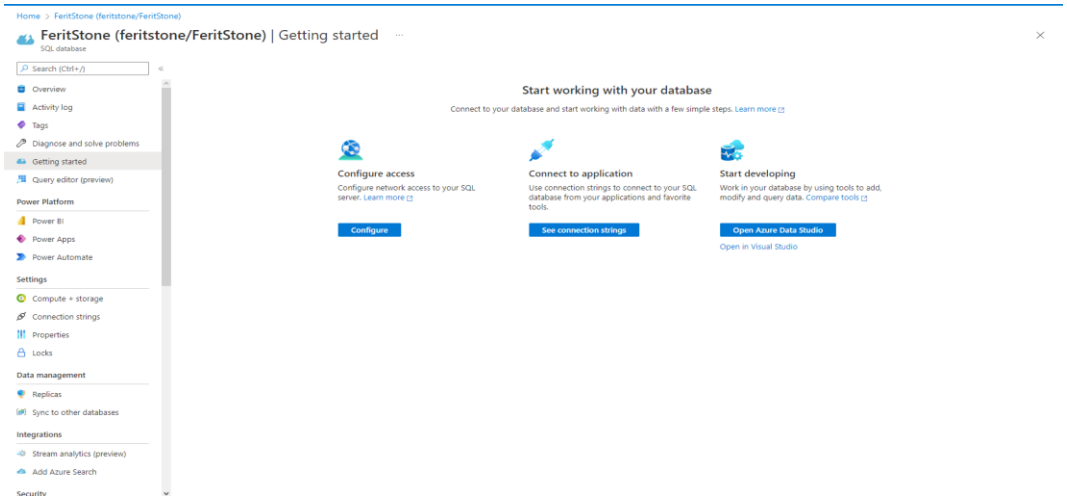
Primjer koda 4.16. Primjer jedne od funkcija za pomak prozora izmjene

5. SPAJANJE BAZE PODATAKA

Iako je aplikacija sada funkcionalna što se tiče koda, ona se ne može koristiti za puno stvari iz razloga što u njoj nema nikakvih karata. Kako bi dodali karte u igru možemo ih napraviti preko koda unutar Visual Studia i to bi bilo to. Problem sa tim načinom rada je to to, ako želimo dodati nove karte mi bi svaki put morali ponovno izmjenjivati unutarnji dio koda što nije poželjno kod TGC i CCG aplikacija.

Da bi odvojili kod od karata koristiti ćemo vanjsku bazu podataka održanu na oblaku. Na ovaj način ćemo sve podatke o našim kartama držati na toj bazi podataka, gdje lagano možemo dodavati još karata, mijenjati njihove efekte, dodavati ili oduzimati efekte, te mijenjati vrijednosti karata po želji da bi igru napravili ispravnijom. Također uvođenjem baze podataka možemo u aplikaciju ubaciti i mogućnosti prijave i registracije koje će davati korisnicima da naprave račune (engl. *Accounts*) uz pomoć kojih ćemo pamtiti i u aplikaciju učitavati razine glasnoće zvuka koje korisnik postavi i spremljene špilove karata koje korisnik napravi ili uredi.

Za poslužitelja baze podataka koristiti ćemo Azure servere zbog njihove jednostavne implementacije u Visual Studio, te zbog besplatnog korištenja usluga Azura baza podataka u periodu od godinu dana. Na Azure web stranici napravimo račun, te onda kliknemo na dugme *Create a resource* odakle nam se novi izbornik pojavi odakle kao resurs odaberemo *SQL Database*. Nakon toga naša baza podataka se napravi održana na serveru lociranom najbliže našoj geografskoj lokaciji. Kada taj postupak završi Azure nas odvede na izbornik naše baze podataka gdje su nam prikazane glavne stvari za početak korištenja baze podataka.



Slika 5.1. Azure izbornik za početak korištenja baze podataka

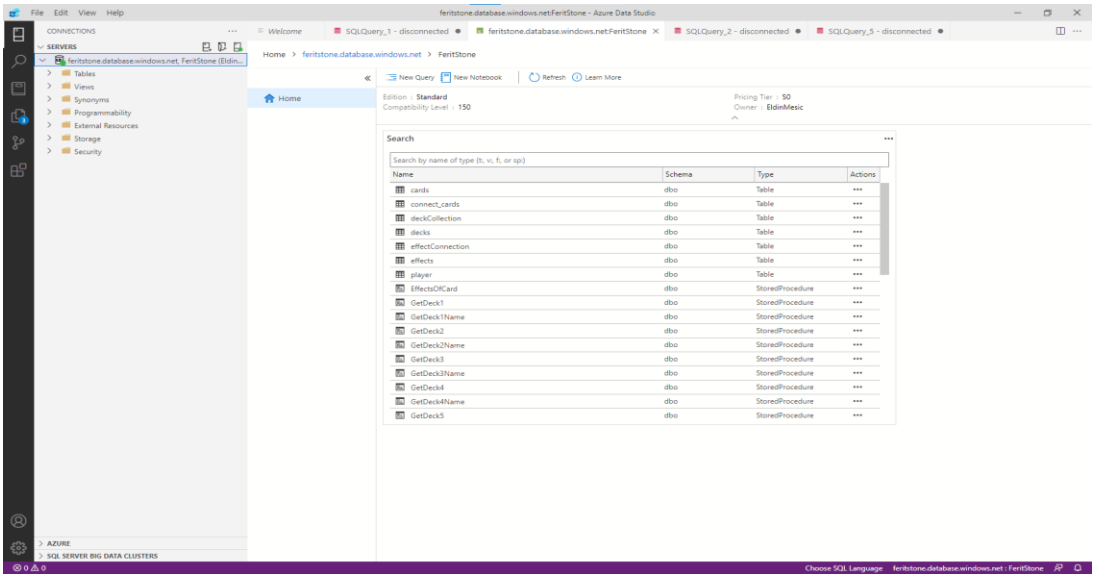
Klikom na konfigurirajte pristup (engl. *Configure Access*) dugme, odvedeni smo na novi izbornik gdje upravljamo tko sve može pristupiti podatcima baze podataka, odnosno, koje IP adrese mogu pristupiti ovim podatcima. Pošto je naša aplikacija napravljena da razni korisnici od bilo gdje u svijetu koriste njezine podatke, stavili smo da je raspon dopuštenih IP adresa od 0.0.0.0 do 255.255.255.255, odnosno, sve adrese mogu pristupiti našoj bazi podataka jer iz nje moraju čitati podatke, te u nju moraju spremiti određene podatke o korisniku.

Klikom na dugme spoji se na aplikaciju (engl. *Connect to application*) odvedeni smo na dio baze podataka koji prikazuje kako i na koji način se baza podataka spaja sa našom aplikacijom, u kojeg god ona jeziku ili razvojnom okruženju bila napravljena. Taj dio ćemo koristiti na samom kraju razvoja baze podataka i u zadnjem potpoglavlju ovog poglavlja.

Klikom na dugme započni sa razvojem (engl. *Start Developing*) stranica nam preuzme povezanu aplikaciju Azure Data Studio u kojem ćemo praviti sve tablice, procedure, ograničenja i povezanosti.

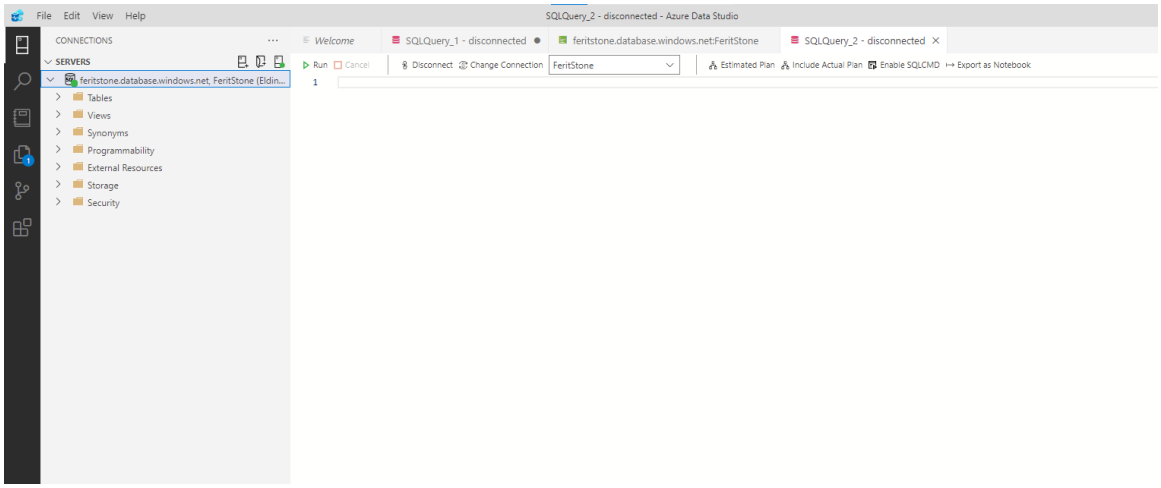
5.1 Stvaranje tablica baze podataka

Trenutno se nalazimo unutar sučelja Azure Data Studia. Povezivanjem našeg računa sa aplikacijom dobijemo izbor koju bazu podataka želimo uređivati, u našem slučaju, to je naša FeritStone baza podataka.



Slika 5.2. Glavno sučelje Azure Data Studia

U glavom sučelju Azure Data Studia imamo pregled svih naših tablica, procedura, funkcija i okidača koje smo napravili u našoj bazi podataka. Sa lijeve strane imamo malo opširniji ali i teži za koristiti pregled istih komponenti sa nekim dodatnim opcijama koje nećemo koristiti. Glavna stvar nama je opcija novi upit (engl. *New Query*) koja otvori novi prozor u kojem preko SQL jezika možemo raditi sa našom bazom podataka šta god želimo.



Slika 5.3. Prozor novog upita

Najvažnija stvar u aplikaciji zbog koje uopće koristimo bazu podataka su karte a time i njihovi efekti, pa prema tome, prve tablice koje moramo napraviti su tablica karti i tablica efekata.

```
CREATE TABLE cards (
    card_id INT NOT NULL IDENTITY PRIMARY KEY,
    card_name VARCHAR(100) NOT NULL,
    card_cost INT,
    card_power INT,
    card_health INT,
    is_spell BIT,
    card_image VARCHAR(100),
    CONSTRAINT Chk_stats
        CHECK (card_cost>=0 AND card_power>=0 AND card_health>=0),
    CONSTRAINT Chk_bit
        CHECK (is_spell BETWEEN 0 AND 1)
);
```

Primjer koda 5.1. Stvaranje tablice *cards*

U tablici *cards* stavili smo sve potrebne informacije za kartu isto kao i kod naše klase *Card* iz Visual Studio koda. Jedina stvar koje nema je polje efekata koje ćemo implementirati preko dodatne

povezne tablice. Također je važno napomenuti da smo postavili i dva ograničenja (engl. *Constraints*), jedan od kojih provjerava ispravnost unosa osnovnih vrijednosti karte; cijene pozivanja, snage karte, životnih poena karte. Drugo ograničenje služi da ograniči unos naše is_spell varijable kao vrijednost 0 ili 1. To radimo zato jer u SQL jeziku ne postoji bool varijabla, pa moramo koristiti dvije ograničene lako razumljive i logične vrijednosti koje ćemo provjeravati u našem kodu kasnije.

```
CREATE TABLE effects (
    effect_id INT NOT NULL IDENTITY PRIMARY KEY,
    card_id INT FOREIGN KEY REFERENCES cards(card_id),
    effect_type VARCHAR(100) NOT NULL,
    effect_description VARCHAR(200) NOT NULL,
    effect_costMod INT NOT NULL,
    effect_powerMod INT NOT NULL,
    effect_healthMod INT NOT NULL,
    CONSTRAINT Chk_type
        CHECK (effect_type IN ('Battlecry', 'Deathrattle', 'Continuous',
                               'OnDamage', 'Spell', 'Attack'))
);
```

Primjer koda 5.2. Stvaranje tablice *effects*

Tablica *effects* sa primjera koda 5.2. isto kao i tablica cards iznad nje ima varijable identične kao svoja pripadna klasa iz Visual Studio koda. Dodano je i jedno ograničenje koje sprječava unos efekta kojem tip efekta nije jedna od zadanih vrijednosti. Ovo ograničenje je važno jer prilikom unosa efekata u bazu podataka u slučaju da se jedan od ovih tipova pogrešno napiše taj efekt nikada ne bi bio iskorišten, te bi nalaženje greške bilo otežano.

Razlog zašto smo odvojili karte od efekata u potpunosti je to što karte i efekti imaju odnos N:M, naše karte mogu imati koliko god želimo efekata, ali to ne znači da je svaki od tih efekata ograničen samo jednoj karti, neke karte mogu imati iste efekte, te bi ih mogli spojiti samo sa jednim efektom, umjesto da pravimo potpuno novi efekt.

Kako bi omogućili ovaj M:N odnos pravimo pomoćnu tablicu koja će služiti samo za povezivanje karte i efekta. Ta tablica će imati samo strani ključ koji predstavlja kartu i strani ključ koji predstavlja efekt, te dodatni primarni ključ u slučaju da želimo da neka karta ima dva identična efekta.

```
CREATE TABLE effectConnection (
    connect_id INT NOT NULL IDENTITY PRIMARY KEY,
    card_id INT NOT NULL FOREIGN KEY REFERENCES cards(card_id),
    effect_id INT NOT NULL FOREIGN KEY REFERENCES effects(effect_id)
);
```

Primjer koda 5.3. Stvaranje tablice povezivanja efekta i karti

Moramo napraviti i tablice vezane za korisnike u kojima će se spremiti podatci o nazivu korisnika, njegova šifra spremljena i zaštićena koristeći hash algoritam i njegove razine glasnoće zvuka.

```
CREATE TABLE player (
    player_id INT NOT NULL IDENTITY PRIMARY KEY,
    player_name VARCHAR(50) NOT NULL,
    player_password VARCHAR(100) NOT NULL,
    general_volume INT NOT NULL,
    music_volume INT NOT NULL,
    effect_volume INT NOT NULL,
    CONSTRAINT Chk_Volume
        CHECK (general_volume>=0 AND music_volume>=0 AND effect_volume>=0)
);
```

Primjer koda 5.4. Tablica koja predstavlja korisnika, odnosno igrača

Naravno svaki korisnik ima svoju kolekciju špilova koja se sastoji od šest špilova, da bi to implementirali u našu bazu podataka, moramo prvo napraviti tablicu koja predstavlja naš špil, te dodatnu tablicu koja će spojiti karte u naš špil jer naše karte ne bi trebale direktno biti povezane sa špilom. Razlog tome je to što naše karte su unikatne, te ako ih povežemo sa špilom bez pomoćne povezne tablice, morali bi praviti nove karte za svaki špil. U primjerima kodova 5.5 i 5.6 imamo prikaz tablica koje čine vezu između špilova i karti.

```
CREATE TABLE decks (
    deck_id INT NOT NULL IDENTITY PRIMARY KEY,
    deck_name VARCHAR(50) NOT NULL
);
```

Primjer koda 5.5. Tablica koja predstavlja glavni objekt špila

```
CREATE TABLE connect_cards (
    deck_id INT FOREIGN KEY REFERENCES decks(deck_id),
    card_id INT FOREIGN KEY REFERENCES cards(card_id)
);
```

Primjer koda 5.6. Tablica koja povezuje špilove sa kartama

Sada kada imamo objekt špila u našoj bazi podataka možemo taj špil preko ograničenja stranog ključa povezati sa korisnikovom kolekcijom preko još jedne tablice koja predstavlja cijeli kolekcijski dio korisnika.

```
CREATE TABLE deckCollection (
    collection_id INT NOT NULL IDENTITY PRIMARY KEY,
    player_id INT FOREIGN KEY REFERENCES player(player_id),
    deck_1_id INT FOREIGN KEY REFERENCES decks(deck_id),
    deck_2_id INT FOREIGN KEY REFERENCES decks(deck_id),
    deck_3_id INT FOREIGN KEY REFERENCES decks(deck_id),
    deck_4_id INT FOREIGN KEY REFERENCES decks(deck_id),
    deck_5_id INT FOREIGN KEY REFERENCES decks(deck_id),Ž
    deck_6_id INT FOREIGN KEY REFERENCES decks(deck_id),
);
```

Primjer koda 5.7. Kreiranje tablice za kolekciju korisnika

5.2 Stvaranje procedura baze podataka

Sa ovime smo napravili sve potrebne tablice za našu aplikaciju, ostalo je samo da napravimo procedure koje će olakšati ubacivanje i čitanje naše baze podataka kako bi sam kod bio pregledniji.

Prva procedura koju ćemo napraviti je procedura za lakše ubacivanje i povezivanje efekta sa kartama. Plan je da se ovoj proceduri predaju sve informacije za pravljenje novog retka efekta, a da ona uz pomoć njih ubaci novi efekt u tablicu *effects* i ujedno da taj efekt poveže sa zadnjom ubačenom kartom tako što napravi novi redak u tablici povezivanja karti i efekata.

Ova procedura će iskoristiti predane vrijednosti i napraviti novi efekt uz pomoć njih koristeći INSERT INTO izraz. Nakon toga će napraviti pomoćnu varijablu u koju će spremiti vrijednost card_id od zadnje karte ubačene u listu. Dobili smo zadnju kartu ubačenu u listu preko SELECT izraza koji uzima samo prvu vrijednost zbog TOP 1 pomoćnog izraza. Da bi dobili zadnju kartu tu prvu vrijednost smo uzeli od liste karta koja je sortirana od najnovije karte do najstarije.

Na taj identičan način smo dobili i zadnji efekt ubačen u tablicu efekata, koji bi ujedno trebao biti i efekt koji je ovaj procedura ubacila, te njegovu vrijednost effect_id sprema u još jednu pomoćnu varijablu.

Nakon svega toga uz te dvije pomoćne vrijednost napravi novi redak u tablici povezivanja karata i efekata, te nam na taj način stvori povezanu vezu između njih.

```
CREATE PROCEDURE InsertEffectLastCard (@name    VARCHAR(100),
                  @desc VARCHAR(100),  @type    VARCHAR(100),
                  @costM   INT,  @powerM  INT,  @healthM INT )
AS  BEGIN
    INSERT INTO dbo.effects (effect_name,  effect_description,
                             effect_type,  effect_costMod,
                             effect_powerMod,  effect_healthMod )
    VALUES  (
        @name,  @desc,  @type,  @costM,  @powerM,  @healthM )

    DECLARE @card_id INT;
    SELECT @card_id = [card_id]
                      FROM cards
                      WHERE card_id = (SELECT TOP 1 card_id
                                       FROM cards
                                       ORDER BY card_id DESC);

    DECLARE @effect_id INT;
    SELECT @effect_id = [effect_id]
                        FROM effects
                        WHERE effect_id = (SELECT TOP 1 effect_id
                                           FROM effects
                                           ORDER BY effect_id DESC);

    INSERT INTO effectConnection (card_id,
                                  effect_id )
    VALUES(@card_id,
           @effect_id )

END
```

Primjer koda 5.8. Procedura povezivanja efekta i zadnje unesene karte

Napravljena je i glavna procedura za dodatak igrača koja prima ime igrača i njegu šifru i sa njima unosi novi redak u tablicu koja predstavlja igrača, odnosno korisnika. Procedura na isti način kao i u primjeru koda 5.8. spremi podatak od zadnje unesenog id-a igrača u pomoćnu varijablu, i onda iskoristi taj id kako bi unijela novi redak u tablici igračeve kolekcije i spremila id svakog špila na osnovnu vrijednost, u ovom slučaju 1.

Procedura onda pravi šest špilova, unosi svakome predodređeno ime i vrijednost deck_id svakoga pamti u dodatnim pomoćnim varijablama.

Na samom kraju procedura preko UPDATE izraza mijenja id vrijednosti špilova unutar igračeve kolekcije na zapamćene vrijednosti unutar pomoćnih varijabli.

```
CREATE PROCEDURE InsertPlayer (@name  VARCHAR(100), @password  VARCHAR(100) )
AS  BEGIN
    INSERT INTO dbo.player (player_name, player_password,
                            general_volume, music_volume, effect_volume )
    VALUES  ( @name, @password, 5, 5, 5 );

    DECLARE @playerId INT;
    SELECT @playerId = [player_id]
                       FROM player
                       WHERE player_id = (SELECT TOP 1 player_id
                                          FROM player
                                          ORDER BY player_id DESC);

    INSERT INTO dbo.deckCollection (player_id, deck_1_id, deck_2_id,
                                    deck_3_id, deck_4_id, deck_5_id,
                                    deck_6_id )
    VALUES (@playerId, 1, 1, 1, 1, 1, 1 )



    [ovaj dio ponoviti za svaki špil]

    INSERT INTO dbo.decks ( deck_name )
    VALUES ( 'Deck 1' )
    DECLARE @deckId INT;
    SELECT @deckId = [deck_id]
                     FROM decks
                     WHERE deck_id = (SELECT TOP 1 deck_id
                                      FROM decks
                                      ORDER BY deck_id DESC);

    UPDATE dbo.deckCollection
    SET deck_1_id = @deckId
    WHERE deckCollection.player_id = @playerId;

END
```

Primjer koda 5.9. Procedura unosa korisnika/igrača

Na slične načine su napravljene procedure za dobivanje svih karata specifičnog špila, za dobivanje svih karata općenito, za dobivanje efekata određene karte, za unos karte u špil, unosa i dobivanja imena određenog špila, te unosa i dobivanja informacija o razini glasnoće zvuka.

```
EXEC InsertCard @name = 'Showcase Card', @cost = 1, @power = 3, @health = 2,
                @spell = 0, @image = 'testCard';
EXEC InsertEffectLastCard @name = 'ModifyFriendlyTarget',
                          @desc = Give an ally Minion +2 Health',
                          @type = 'OnDamage',
                          @costM = 0, @powerM = 0, @healthM = 2;
```

Primjer koda 5.10. Primjer unosa karte sa jednim efektom

5.3 Povezivanje baze podataka sa aplikacijom

Sljedeći zadatak je vratiti se u Visual Studio i dodati dijelove koda koji će spojiti bazu podataka zajedno sa aplikacijom. Postupak kojim se kod spaja sa bazom podataka je prikazan jednostavnim primjerom na kodu 5.11. ispod.

```
string connStr = "Server=tcp:feritstone.database.windows.net,1433;Initial
Catalog=FeritStone;Persist Security Info=False;User
ID=EldinMesic;Password=EMesic88;MultipleActiveResultSets=False;Encrypt=True;T
rustServerCertificate=False;Connection Timeout=30;";
using (SqlConnection connection = new SqlConnection(connStr))
{
    connection.Open();

    string sql = "UPDATE player
                  SET generalVolume = {generalVolume}
                  WHERE player_name = '{playerName}'";
    using (SqlCommand command = new SqlCommand(sql, connection))
    {
        using (SqlDataReader reader = command.ExecuteReader())
        {
            while (reader.Read())
            {
                ...
            }
        }
    }
    connection.Close();
}
```

Primjer koda 5.11. Primjer spajanja koda sa bazom podataka koristeći tekst za povezivanje

Na ovaj način iz naše baze podataka može čitati podatke koje želimo, unositi podatke koje želimo i izmjenjivati podatke koje želimo. Tako ćemo napraviti novu skriptu koja će predstavljati i učitavati karte naše baze podataka zajedno sa njihovim efektima.

U skripti će se otvoriti konekcija sa bazom podataka i onda koristeći proceduru za dohvat svih karata naprave se karte bez efekata i dodaju u glavnu listu a njihova imena se spreme u odvojenu listu. Nakon toga, za svako ime karte iz naše druge liste napravimo novi upit koji zove proceduru kojoj preko danog imena karte dobijemo sve efekte te karte. Te efekte spremimo u listu efekata i damo prikladnoj karti. Na taj način u našu glavnu listu učitamo sve karte i efekte sa baze podataka.

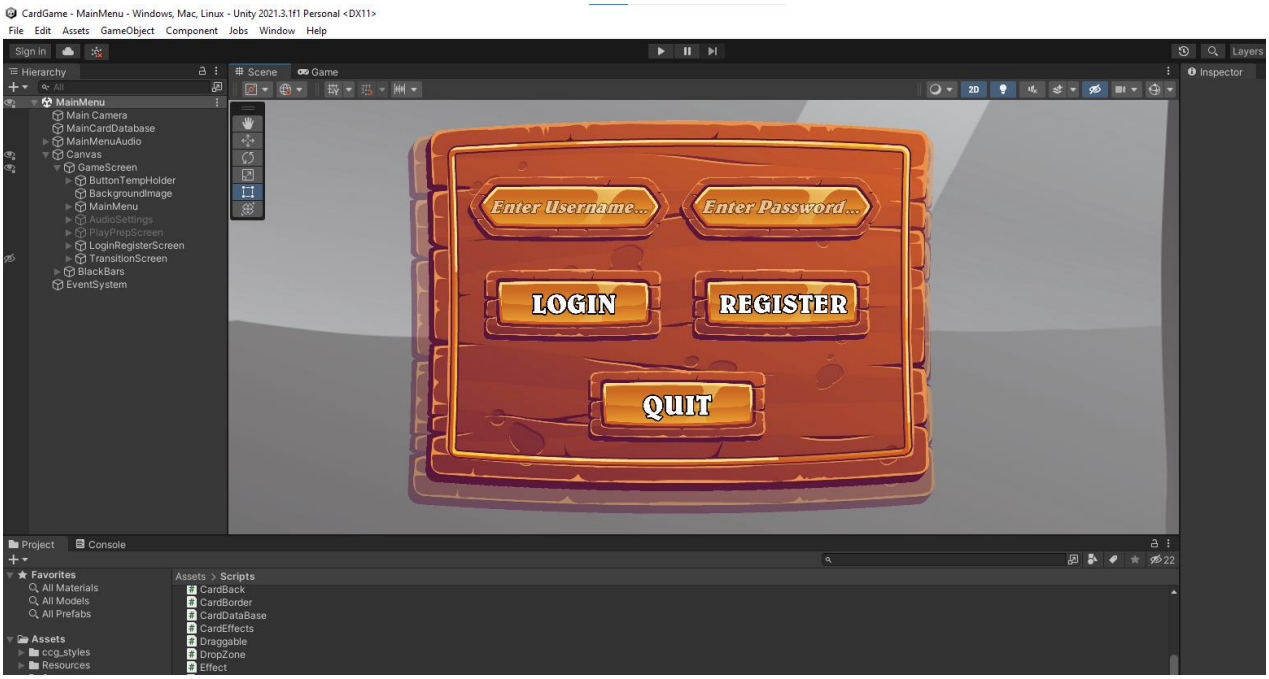
```
using (SqlConnection connection = new SqlConnection(connStr))
{
    connection.Open();
    string sql = "SELECT * FROM cards";
    using (SqlCommand command = new SqlCommand(sql, connection))
    {
        using (SqlDataReader reader = command.ExecuteReader())
        {
            while (reader.Read())
            {
                string imageName = reader["card_image"].ToString();
                StaticVariables.MainCardList.Add(new Card(
                    Convert.ToInt32(reader.GetValue(0)),
                    reader["card_name"].ToString(),
                    Convert.ToInt32(reader.GetValue(2)),
                    Convert.ToInt32(reader.GetValue(3)),
                    Convert.ToInt32(reader.GetValue(4)),
                    "",
                    Resources.Load<Sprite>($@"CardImages\{imageName}"),
                    Convert.ToInt32(reader.GetValue(5)) == 1 ) );

                    cardNames.Add(reader["card_name"].ToString());
            }
        }
    }

}
```

Primjer koda 5.12. Učitavanje svih karti iz baze podataka u glavnu listu

Na isti način je odrađeno učitavanje karti unutar špilova korisnika, učitavanja karti iz novouređenih špilova u bazu podataka i učitavanje i uređivanje razine glasnoće zvuka aplikacije. Jedina stvar koja je preostala da ova aplikacija bude u potpunosti spremna za korištenje je srediti prijavu i registraciju korisnika.

Na sceni glavnog izbornika smo dodali novu grupu Unity objekata koji predstavljaju sučelje za prijavu i registraciju. Ovi objekti su postavljeni iznad svih drugih što znači da je to sučelje prva stvar koju korisnici vide nakon što se aplikacija pokrene, a ujedno i jedina stvar koju će vidjeti dok god se ne ulogiraju u aplikaciju.

Sučelje za prijavu i registraciju ne treba biti previše posebno, dovoljno je da ubacimo dva elementa unosnih polja preko kojih će korisnik unijeti svoje ime i šifru, te tri dugmeta, jedno za prijavu sa unesenim podatcima, drugo za registraciju sa unesenim podatcima i treće za izlaz iz igre.



Slika 5.4. Izgled sučelja za prijavu/registraciju

Još je samo ostalo napraviti kod funkcionalnosti *login* i *register* dugmadi. Pozabaviti ćemo se prvo sa funkcijom registracije. Prije nego zapravo krenemo pisati funkciju registracije i prijave, potrebno je u našoj novoj skripti, koja sadržava funkcionalnost *login/register* sučelja, dodati učitavanje svih imena i kriptiranih šifri, te ih spremiti u posebnu listu preko koje ćemo provjeravati podudaranje i spriječiti ponavljanje imena.

```
public void Register()
{
    string userName = UserNameField.text;
    string password = PasswordField.text;
    string hashPass = HashString(password);
    if(!CheckIfValid(userName))
    {
        notificationText.text = "Error Text";
    }
    else
    {
        notificationText.text = "<color=green>SUCCESS:</color>\nAccount Has
                                 Been Successfully Created";
        AddUserToDatabase(userName, hashPass);
    }
}
```

Primjer koda 5.13. Metoda koja registrira korisnika u bazu podataka

Na isti ovaj način je napravljena i funkcija za prijave u igru, jedina razlika je to što umjesto da se pojavi tekst za uspješnu registraciju prozor sučelja se deaktivira i korisniku se dadne pristup ostatku aplikacije.

Također se prilikom prijave preko već napravljenih SQL procedura i prethodno poznatog koraka spajanja sa bazom podataka i izvršavanja upita učitavaju korisnikovi špilovi, postavke glasnoće zvuka i imena špilova.

6. ZAKLJUČAK

Ovim radom proširili smo naše znanje o nekim od najčešćih programa korištenim u kompleksnom svijetu dizajniranja desktop aplikacija. Također smo pridodali tom svijetu sa našom ne toliko jednostavnom aplikacijom. Tijekom pravljenja ove igre, suočili smo se sa mnogim problemima i poteškoćama, ali ujedno i sa određenim tehnikama koje se koriste i u daleko razvijenijim tvrtkama koje pomažu u potpunom micanju ovih problema ili barem smanjenju njihove učestalosti. Igra koju smo napravili ne bi trebala biti podcijenjena. Iako nije na razini raznih TCG i CCG aplikacija čije smo mehanike i mnoge ideje posudili i uspjeli emulirati u našoj igri. Naučili smo raditi sa tehnikom popunjivih objekata, korištenjem Unity programa za razvoj 2D i 3D aplikacija, te stvaranjem, popunjavanjem i povezivanjem baze podataka. U Unity Game Engine-u smo prošli kroz razne načine kreiranja 2D dizajna, korištenja nekoliko komponenti kao što su ScrollRect i Prefab objekti. U Unity-u smo prošli i kroz izmjenu i kreiranje novih scena, te kontrola tih scena preko koda skripti ugrađenih u pojedine scene. Naš rad u Visual Studiu je pokazao da je C# programski jezik adekvatan i veoma efektivan što se tiče programiranja ovakvih aplikacija. Mnoge tehnike i načela objektno orijentiranih jezika su pridonijele našem kodu i njegovoj čistini i efikasnosti. Azure Data Studio i same Azura baze podataka su bile savršen uvod u korištenje pravih baza podataka održanim na vanjskim serverima preko internetskih poslužitelja. Korisničko sučelje Azure Data Studia se pokazalo veoma jednostavnim za koristiti, te je bio dobar oslonac za našu bazu podataka. Preko njega smo poboljšati znanje našeg SQL jezika, te naučili izmjenjivati više tablica baze podataka preko procedura.

LITERATURA

[1] M. Dealessandri, *What is the best game engine: is Unity right for you?*, Internet članak ,Gamer Network, 16. Siječanj, 2020.

[2] J. Brodkin, *How Unity3D Became a Game-Developing Beast*, Dice Insights, 19. Listopad, 2018.

[3] T. Slunjski, *Izrada 2D igre kroz Unity3D programski alat*, Završni rad, Sveučilište Sjever, Koprivnica, 2017. Dostupno na: https://urn.nsk.hr/urn:nbn:hr:122:432063

[4] I. Stojaković, *Izrada video igre u Unity razvojnom okruženju*, Završni rad, Sveučilište Sjever, Koprivnica, 2015. Dostupno na: https://urn.nsk.hr/urn:nbn:hr:122:136213

[5] M. Vijay, *Extending Visual Studio 2019*, Internet članak, CodeGuru, 17. Ožujak, 2021.

[6] A. Yaseen, *Starting your journey with Azure Dana Studio*, Internet članak, SQLShack, 19. Srpanj, 2020.

[7] C. Rabeler, *Azure SQL Database Learns & Adapts*, Gantec Publishing Solutions, 13. Studeni, 2016.

SAŽETAK

U ovom radu pokazan je rad u više razvojnih okruženja kao što su Visual Studio, Azure Data Studio, Unity razvojno okruženje i Photoshop u svrhu razvijanja TGC i CCG desktop aplikacije nalik sličnim, uspješnijim aplikacijama od puno većih kompanija. Napravljena igra je napisana u C# programskom jeziku te sadrži mogućnosti spremanja podataka korisnika preko sustava za registraciju i prijavu, mogućnost pravljenja špila od velike kolekcije karti koje je moguće preko povezane baze podataka izmjenjivati na koji god način je potrebno bez mijenjanja unutarnje strukture koda. Svaka karta ima mogućnost posjedovati više efekata koji rade različite radnje. Glavni dio aplikacije je borbeni bio gdje se korisnik sa svojim špilom bori protiv AI protivnika. Sve promjene špilova pojedinog korisnika i sve karte općenito se nalaze na odvojenoj bazi podataka odakle se sve njihove informacije mogu mijenjati.

Ključne riječi: AI protivnik, Azure, baza podataka, CCG, C#, desktop aplikacija, kolekcija karti, razvojno okruženje, registracija i prijava, TCG, Unity

ABSTRACT

Unity combat card game application

This paper presents work in several development environments such as Visual Studio, Azure Data Studio, Unity Game Engine and Photoshop for the purpose of developing a TGC and CCG desktop application similar to more successful application from much larger companies. The created game contains the possibility of saving user data through the registration and login system, the possibility of creating a deck from a large card collection that can be changed via the connected database in any way necessary without changing the internal code structure. Each card has the ability to have multiple effects that do different actions. The main part of the application is the battle segment, where the user fights with his deck against an AI opponent. All changes to an individual user's decks and all cards in general are on a seperate database from where all their information can be changed.

Keywords: AI opponent, Azure, card collection, CCG, C#, database, desktop application, game engine, registration and login, TGC, Unity

ŽIVOTOPIS

Eldin Mešić rođen je 05.02.2000. godine u Vinkovcima, Republika Hrvatska. Završio je Osnovnu Školu Orašje u Orašju. Srednju školu je završio u Orašju, smjer elektrotehničar. 2019. godine upisao se na preddiplomski sveučilišni studij elektrotehnike na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku, a 2020. godine prebacio na preddiplomski sveučilišni studij računarstvo na Fakultetu eleketrotehnike, računarstva i informacijskih tehnologija u Osijeku.