UNIVERZA NA PRIMORSKEM
FAKULTETA ZA MATEMATIKO, NARAVOSLOVJE IN
INFORMACIJSKE TEHNOLOGIJE

Zaključna naloga

Realizacija namizne igre z igralnim pogonom Unity

(Board Game Realization Using Unity Engine)

Ime in priimek: Gašper Moderc
Študijski program: Računalništvo in informatika
Mentor: doc. dr. Jernej Vičič
Somentor: doc. dr. Branko Kavšek

Koper, julij 2018

Ključna dokumentacijska informacija

Ime in PRIIMEK: Gašper MODERC

Naslov zaključne naloge: Realizacija namizne igre z igralnim pogonom Unity

Kraj: Koper

Leto: 2018

Število listov: 57 Število slik: 15 Število tabel: 1

Število referenc: 11 Število prilog: 5 Število strani prilog: 5

Mentor: doc. dr. Jernej Vičič

Somentor: doc. dr. Branko Kavšek

Ključne besede: Gospodar prstanov, Namizna igra, Lotr: LCG, Unity, Realizacija namizne igre, Igralni pogoni

Izvleček:
V zaključni nalogi je opisana realizacija namizne igre z igralnim pogonom Unity. V prvem delu zaključne naloge je predstavljen problem, ki je rešen, predstavljena je tudi celotna namizna igra Lotr: LCG in igralni pogon Unity. V drugem delu so analizirane zahteve, ki jih mora končni produkt izpolnjevati. V tretjem delu je načrtovanje sistema z delitvijo na podsisteme in opis vsakega izmed njih. V četrtem delu je predstavljena še izvedba končnega produkta s primeri kod in uporabniškega vmesnika. Izvedba je razdeljena po podsistemih in opisuje vsak podsistem posebej. V zaključku je povzeta celotna zaključna naloga, prav tako so podane ideje za nadaljnje delo.

Key words documentation

Name and SURNAME: Gašper MODERC

Title of the final project paper: Board Game Realization Using Unity Engine

Place: Koper

Year: 2018

Number of pages: 57 Number of figures: 15 Number of tables: 1

Number of references: 11 Number of appendices: 5

Number of appendix pages: 5

Mentor: Assist. Prof. Jernej Vičič, PhD

Co-mentor: Assist. Prof. Branko Kavšek, PhD

Keywords: Lord of the rings, Board game, Lotr: LCG, Unity, Board game realization, Game engine

Abstract:
The thesis describes the realization of a board game using Unity game engine. In the first part of the thesis there is description of the solved problem and presentation of the whole Lotr: LCG board game and Unity game engine. The second part consists of requirements analysis, which final product most meet. The third part consists of system planning including division of the system into smaller subsystems with their description. In the fourth part of the thesis, the realization is described, with code and user interface examples. The realization is divided into the subsystems and is described for each of them. In the conclusion there is a summary of the whole thesis, with added ideas for future upgrades.

Zahvala

Zahvaljujem se mentorju, doc. dr. Jerneju Vičiču in somentorju doc. dr. Branku Kavšku za pomoč pri izdelavi in strokovnemu pregledu diplomskega dela. Zahvaljujem se tudi ostalim delavcem Fakultete za matematiko, naravoslovje in informacijske tehnologije Koper za njihovo strokovno pomoč in nasvete.

Kazalo vsebine

Kazalo tabel

Kazalo slik

Kazalo algoritmov

Kazalo prilog

Seznam kratic

Lotr: *LCG*	Lord of the rings: Living card game : Gospodar prstanov: Živa namizna igra
FFG	Fantasy flight games: Ustanoviteljsko podjetje Lotr: LCG
RAM	Random-access memory : spomin z direktnim dostopom
UML	Unified Modeling Language : poenoten vzorčni jezik
MacOS	Macintosh operating systems : Macintosh operacijski sistem
DTP	Diagram toka podatkov
CSV	Comma-separated values : Datoteka z vrednostmi, ločenimi z vejico

1 Uvod

Tudi stare namizne igre so se s pospešeno rastjo popularnosti računalniških iger začele digitalizirati. Na primer šah, igre s kartami, človek ne jezi se in podobne igre so vse zelo hitro dobile svoje digitalne oblike. Zaradi vedno večjega povpraševanja po računalniških igrah so se začeli razvijati tudi igralski pogoni, ki nam omogočajo hitro izdelavo digitalnih iger.

Tudi namizna igra Lotr: LCG je skozi čas hitro pridobivala na popularnosti. Vendar se je izkazalo, da je to igro zelo težko realizirati v digitalni obliki, saj je interakcij med kartami ogromno, pravilno razrešiti vsako izmed njih pa je zelo zahtevno. Znotraj te diplomske naloge bo predstavljena moja programska rešitev za poskus realizacije te namizne igre z igralnim pogonom Unity.

Igralski pogon Unity se je zelo hitro razvijal skozi čas in je kmalu postal eden najbolj uporabljenih orodij za razvijanje računalniških iger. Večina zaslug gre njegovi enostavni uporabi in prijaznemu vmesniku, ponaša pa se tudi z rešitvami za čisto vse platforme in omogoča tudi spletno igranje za več igralcev hkrati.

Za realizacijo namizne igre Lotr: LCG sem si izbral pogon Unity ravno zaradi tega, ker omogoča igranje prek vseh platform, z le minimalnimi popravki uporabniškega vmesnika. Za izboljšanje uporabniške izkušnje se navadno optimizira vmesnik za vsako velikost ciljne naprave posebej, torej za računalniški ekran, za mobilni telefon in za tablico. Pogon zadošča tudi ostalim funkcijskim in nefunkcijskim zahtevam programa.

Struktura dela je zasnovana na sledeči način: Najprej bosta predstavljena ozadje problema in predlagana rešitev za ta problem. Sledila bo obširnejša analiza problema, ki bo zajemala vse funkcijske in nefunkcijske zahteve, predstavljena bo tudi delitev celotnega sistema na manjše podsisteme z jasno določenimi nalogami. Sledita podroben opis načrtovanja podsistemov in podrobnejša analiza funkcij vsakega izmed njih. Opisana je tudi izvedba projekta, v kateri je predstavljena realizacija vsakega podsistema posebej, ter komunikacija med njimi. Podani so tudi primeri iz uporabniškega vmesnika in programske kode iz projekta. Na koncu sta podana zaključek in opis možnosti za nadaljnje delo na projektu.

2 Predstavitev problema

V tem razdelku je podrobneje predstavljen zastavljen problem, ki je rešen z implementacijo predstavljeno v tej diplomski nalogi. Predstavljena bosta priložnost in razlog za izbor namizne igre Lotr: LCG. Na koncu bo predstavljena še namizna igra Lotr: LCG, ki bo znotraj projekta pridobila svojo digitalno obliko.

Predstavljeno je tudi programsko okolje Unity, ki služi kot igralni pogon za izdelavo računalniških iger. Prav tako bodo predstavljene prednosti igralnega pogona Unity, ki so pripomogle k njegovemu izboru izbora za realizacijo namizne igre.

2.1 Predstavitev namena projekta

Namizna igra Lotr LCG[1] ima zelo veliko navdušencev in po kar osmih letih je še vedno zelo igrana. Ves čas se organizirajo razni turnirji, skoraj vsak drugi mesec pa se še vedno izdajajo novi scenariji in nove karte. Razlog za to je gotovo velika popularnost tematike, saj ima svet Gospodarja prstanov navdušence po celem svetu.

Nove karte so večinoma predstavljene na večjih dogodkih, kot so Comic-coni v Združenih državah Amerike ali pa različni tematični dogodki znotraj Evrope. Podjetje FFG[2] spodbuja lokalne trgovine s kartami, da tudi same organizirajo različne dogodke ter jim ponudi posebne karte v zameno, ki se jih da dobiti le z udeležbo na takih turnirjih. Zelo veliko ljudi pa igra namizno igro doma s prijatelji in znanci. Problem nastane, ko si za dlje časa ločen od prijateljev, oziroma si prvi v svojem okolišu, ki te zanima igranje in nimaš možnosti igrati z nobenim prijateljem.

V modernem času je že veliko podjetij ravno iz teh razlogov naredilo „online“ verzijo svojih namiznih iger. To pomeni, da vsak, ki je kupil osnovno igro, lahko le-to igra tudi prek interneta, tako pa lahko igra s komerkoli in ni odvisen od svoje lokacije. Podjetje FFG nekoliko zaostaja s postavitvijo svojih iger na globalni splet, kar je odprlo priložnost za realizacijo te igre v digitalni obliki. V diplomski nalogi je zato poizkušena izvedba takega projekta, ravno zaradi namena spoznavanja težavnosti vodenja projekta takšne velikosti in pridobitve pomembnih izkušenj na različnih računalniških področjih. Pri tem je bilo zagotovljeno srečanje s problematiko podatkovnih baz, izdelave enostavnega uporabniškega vmesnika in veliko programiranja različnih efektov kart. Po morebitnem uspešnem dokončanju projekta je bilo zamišljeno pokazati to izvedbo podjetju FFG, če bi jih morda zanimalo.

[1] Opis vseh trenutnih izdelkov dostopen na: https://www.fantasyflightgames.com/en/products/the-lord-of-the-rings-the-card-game/.

[2] Spletna stran podjetja dostopna na: https://www.fantasyflightgames.com/.

2.2 Predstavitev namizne igre Lotr: LCG

Igra Lord of the rings: Living card game je bila predstavljena leta 2011 pod podjetjem Fantasy flight games. Igra je namenjena 1–4 igralcem, ki morajo skupaj premagati različne scenarije, ki naključno sestavljajo vsako igro. Vsak igralec si pred igro izbere poljubne tri heroje iz svoje kolekcije in sestavi svoj igralni kupček, iz katerega med igro pridobiva karte.

Posebna lastnost te igre je, da se igra razširi vsakih nekaj mesecev. S prihodom novega dodatka k igri dobi igralec nove karte, ki jih lahko vključi v svoj kupček. Vsak dodatek pa vsebuje nov izziv v obliki scenarija. To privede do velikih težav za morebitno računalniško aplikacijo te igre. Stalno spreminjajoče se zahteve lahko z vsako novo edicijo popolnoma spremenijo potek igre, kar predstavlja nočno moro za realizacijo, saj mora biti programska oprema sposobna prenesti velike spremembe v specifikacijah vsakih nekaj mesecev (več o tem v razdelku Analiza zahtev).

Pred začetkom vsake igre si igralci izberejo poljubni scenarij, ki ga bodo poskušali premagati. Scenarij sestavlja nekaj (največkrat 2–4) kart, ki predstavljajo misije in nasprotnikov kupček, ki se igra sam po točno določenih pravilih. Najpogosteje je cilj scenarija uspešno potovanje skozi vse misije, te pa se v točno določenem zaporedju odkrivajo. Takoj ko je neka misija premagana, se zamenja z naslednjo, igra pa se pri tem nadaljuje.

Vsak krog je sestavljen iz sedmih potez, ki so vedno v enakem zaporedju. Najprej dobijo igralci možnost igranja novih kart iz roke, da si povečajo svojo moč na bojišču. Izbirajo lahko med uroki, pripomočki in novimi zavezniki. Te „kupijo“ z denarjem, ki ga heroji prislužijo vsak krog. V nadaljevanju igralci izberejo nekaj likov, s katerimi želijo oditi na misijo. Za to pa lahko izbirajo med heroji in zavezniki. Med potovanjem vstopijo v igro nove nasprotnikove karte in pri tem se preveri, ali imajo igralci dovolj močne like na potovanju, ali pa so nasprotnikove karte močnejše. Če so bili boljši igralci, naredijo nekaj napredka na misiji (ali lokaciji, če je kakšna aktivna), če pa je bil močnejši nasprotnik, se grožnja igralcev poveča (če pride grožnja nekega igralca do 50, je izgubil igro). Po tej fazi igralci lahko izberejo ali bodo potovali na novo lokacijo (ki ima lahko tudi določen efekt) ali pa ne. Sledi še poteza, kjer se določi, kateri nasprotniki bodo napadli katere igralce. Kasneje nastopi bitka, v kateri se vsak igralec spopade z nasprotniki, ki so ga napadli. V bitki lahko sodelujejo igralčevi zavezniki in heroji, ki v tej potezi niso odšli na misijo. Po tej fazi sledi še zaključna faza, ko si vsi heroji in zavezniki odpočijejo, grožnja igralcev pa se poveča.

2.3 Računalniške igre in igralni pogon Unity

V tem razdelku je predstavljen začetek računalniških iger. Kot prva računalniška igra bo šteta tista, ki je bila realizirana z elektronskim vezjem, ni pa še bila sprogramirana. V nadaljevanju je predstavljeno, kako so se skozi čas razvijali prvi igralni pogoni. Predstavljena je tudi konstrukcija igralnega pogona, torej kakšne funkcije mora vsebovati in za kaj se jih uporablja v praksi. Na koncu bo predstavljeno še razvijalno okolje Unity 3D, ki je bilo uporabljeno za realizacijo te namizne igre. Opisane bodo njegove funkcije

in razlogi za uporabo tega igralnega pogona namesto kakega drugega.

2.3.1 Začetki računalniških iger

„Računalniške“ igre so se prvič pojavile okrog leta 1950, čeprav takrat sploh še ni bilo možno govoriti o računalnikih kot o strojih, kot jih poznamo danes. Prve igre so bile narejene kar na elektronskih vezjih, vendar je bil njihov namen predvsem raziskovalni in akademski. Kot je v svoji knjigi napisal avtor Wolf [11], so se kasneje, okrog leta 1970 pojavile prve „računalniške“ igre na konzolah, ki so bile predstavljene tudi širši javnosti, njihov namen pa je postal zabavljaški in za preživljanje prostega časa. Prve izmed njih so bile recimo Computer Space in PONG.

Tudi razvoj računalniških iger se je skozi leta drastično spreminjal. Sprva je imela vsaka „računalniška“ igra čisto svoje elektronsko vezje, dve različni igri pa nista imeli skoraj nič skupnega. Ko se je neko podjetje odločilo narediti novo igrico, ki je bila precej podobna prejšnji, se je torej moralo lotiti celotnega projekta praktično od začetka. Prav tako je tukaj treba omeniti, da si lahko na nekem vezju igral kvečjemu eno računalniško igro. Ker so igre izhajale iz različnih vezij, ni mogla ena naprava ponujati možnost igranja dveh različnih iger. Za igranje druge igre si moral kupiti celotno novo napravo. Tukaj lahko že govorimo o igralnih avtomatih, na katerih se je lahko igralo zgolj eno arkadno igro.

S tem, ko so računalniške igre pridobivale na popularnosti, pa se je začela širiti tudi potreba po tem, da bi bila izdelava iger hitrejša in da bi lahko nekakšen delček že narejene igre uporabili pri izdelavi nove. S tem bi podjetja veliko privarčevala na stroških razvoja, čas izdelave igre pa bi se zmanjšal. Hkrati s tem se je tudi širila potreba po tem, da bi lahko na eni napravi igrali več različnih iger in da nebi bilo treba končnemu uporabniku za vsako novo igro kupiti celoten arkadni avtomat.

Rešitev so prinesle igralne konzole (okrog leta 1960), ki so bile sprva priključene na vektorske prikazovalnike. Omogočale so igranje različnih iger na isti konzoli, kar je razrešilo en problem, uvedle pa so tudi že možnost, da lahko isto računalniško igro igra več ljudi hkrati (ampak samo lokalno, torej da so vsi igralci fizično prisotni pri isti konzoli). Popularnost takih iger je hitro začela naraščati, pa tudi razvoj igralnih konzol je hitro začel napredovati.

2.3.2 Razvoj igralnih pogonov

Skozi čas se niso spreminjale samo naprave, na katerih so se igrale računalniške igrice, ampak tudi sam način izdelovanja iger. Ko so računalniki začeli postajati zmogljivejši, so se tudi postopki za izdelavo iger začeli izboljševati. Osnovna ideja je namreč bila, da se uporabi čim večji delež že delujoče in izdelane igre za izdelavo nove igre. Torej da lahko poljubno izberemo samo nek delček igre, ki se obdrži, ostalo pa je izdelano na novo.

Avtor Gregory Jason je v svoji knjigi Game Engine Architecture [4] napisal o zgodovini igralnih pogonov, da je pred letom 1990 imela vsaka igra svoj igralni pogon (ang. Game Engine), ki se je razvijal skupaj z igro. Igralni pogon je omogočal programerjem upravljanje igre, dodajanje novih funkcionalnosti, vnos popravkov itd. Vsaka igra je imela svoj igralni pogon, ki je lahko upravljal izključno to igro. Dodatne težave je povzročalo to, da se je ta mogel razvijati skupaj z igro. Po letu 1990 pa so se prvič začeli pojavljati neodvisni igralni pogoni, ki niso bili več vezani izključno na eno igro, temveč so omogočali izdelavo več različnih, a podobnih iger. Prvi taki so bili recimo Hydro thunder Engine, Quake, Doom, ipd. Ti so omogočali izdelavo zelo ozkega spektra računalniških iger, torej vse igre izdelane v njih so si bile med seboj zelo podobne. Pomembno pa je to, da so prvič lahko v enem igralnem pogonu izdelovali več različnih iger.

Nekoliko kasneje pa so se začeli pojavljati prvi igralni pogoni, ki naj bi omogočali izdelavo širokega spektra računalniških iger. Prvi tak je bil Quake III, kmalu za tem pa sta se pojavila tudi Unreal Engine in Unity 3D, ki sodita v najbolj uporabljane igralne pogone še danes. Ti igralni pogoni naj bi omogočali izdelavo skoraj poljubne igralne igre (torej skoraj vsake, ki si jo lahko človek zamisli, ampak zaenkrat to še ni mogoče). Pomembno je omeniti tudi, da so to samo igralni pogoni, ki so namenjeni uporabi za širšo javnost. Torej z nakupom licence (oz. uporabljanje zastonj verzije) lahko poljubni uporabnik izdela čisto svojo igrico in to samo z osnovnim znanjem računalništva. Večja podjetja, ki proizvajajo najbolj kompleksne računalniške igre, pa ne uporabljajo teh igralnih pogonov, ampak večinoma razvijejo svoje pogone, ki jih posebej prilagodijo spektru iger, ki jih podjetja proizvajajo.

2.3.3 Predstavitev igralnih pogonov

Kot je bilo omenjeno že prej, če želi nek uporabnik narediti svojo igrico, se dandanes nikoli ne loti programiranja od čistega začetka. Veliko orodij je že vgrajenih v igralne pogone. Če si torej izbere ustrezni igralni pogon in začne igro izdelovati v njem, si lahko prihrani ogromno časa, prav tako pa tudi lahko uporabi že preizkušene in široko uporabljene algoritme, namesto da bi bil prisiljen sam sprogramirati svoje. Še vedno pa je potrebno vsa ta orodja znati pravilno uporabiti, da pride do končnega rezultata. Kot so ugotavljali avtorji v študiji History and comparative study of modern game engines [9] in v Designing a PC Game Engine [1] so najpomembnejša med temi orodji:

- pogon, ki skrbi za fizikalne zakonitosti,
- grafični pogon,
- pogon za umetno inteligenco,
- skripte,
- pogon za zvok

Pogon, ki skrbi za fizikalne zakonitosti, nadzoruje da se igra izvaja ves čas pod določenimi fizičnimi zakoni. Ta zna poskrbeti za gravitacijo delčkov glede na njihovo

maso in material ter je tudi sposoben računati premike predmetov v prostoru. Pomemben in fizikalno težek del so prav tako kolizije določenih predmetov, ko mora program biti sposoben izračunati nastalo škodo na predmetu, jo pravilno prikazati in izračunati odboj delcev vsakega v svojo smer. Če bi se vse to začelo računati od začetka za neko igro, bi bilo potrebno vložiti ogromno matematičnega znanja in predvideti vse mogoče situacije za vsako možno kolizijo. Dandanes pa dober igralni pogon vse to že vsebuje in za to poskrbi sam.

Grafični pogon poskrbi za pravilno izrisovanje predmetov v računalniškem prostoru tako, da lahko uporabnik spremlja napredek in trenutno situacijo. Celotna igra se v programu izvaja samo nekje v ozadju, kjer skupek programskih predmetov in kod opiše trenutno dogajanje v igri. Da lahko uporabnik sploh sledi kaj se dogaja v igri, mora biti vse to grafično predstavljeno in izrisano na ekranu. Dandanes je glavna funkcija grafičnega pogona, da zna optimalno izrisovati in prikazati kompleksno 3D grafiko, sestavljeno iz velik risb in 3D slik. Današnji RAM pomnilniki so sicer že zelo veliki, a še vedno zdaleč premajhni, da bi lahko v sebi vsebovali celotno grafiko neke računalniške igre. Zato je potrebno nekako imeti v hitrem pomnilniku samo tiste dele grafike, ki so trenutno na ekranu, hkrati pa pametno pričakovati, katera grafika bo kmalu prišla na vrsto za uporabo in jo pripraviti v pomnilniku. Vse to je dandanes že implementirano v boljših grafičnih pogonih in je že takoj na voljo uporabniku za uporabo.

Pogon za umetno inteligenco skrbi za obnašanje likov v računalniški igri. Glavna prednost tega pogona je, da lahko z odločitvenimi drevesi in tabelami hitro opišemo željeno obnašanje nekega lika (navadno nasprotnika) v igri. Najpogosteje namreč hočemo, da je nasprotnik odziven in pripravljen prav na vsako možno situacijo, po možnosti pa tudi strateško sodeluje z ostalimi nasprotniki. Vse to pripelje do zelo kompleksnih odločitev in obnašanja. Potreba po tem pogonu je zelo odvisna od vrste igre, saj pri nekaterih (recimo strateških) predstavlja ravno ta pogon največjo kompleksnost celotnega sistema, pri drugih pa je skoraj popolnoma odstoten.

Obstaja še veliko drugih pomembnih pogonov, recimo pogon za skripte, ki poskrbi za ustrezno dogajanje znotraj igre po vnaprej določenem scenariju. Na primer da ne more igralec zapustiti določenih omejenih področij, da se sprožijo ustrezni dogodki ob pravilnem času itd. Tudi pogon za zvok in animacije je zelo pomemben, saj igre pogosto zahtevajo, da je zvok odvisen od različnih parametrov v igri, na primer da se predvaja s spremenljivo hitrostjo, spremenljivo glasnostjo, ob različnih trenutkih ... Če bi morali posneti za vsako možno situacijo zvok s popolnoma določeno glasnostjo, hitrostjo in ozadjem, bi gotovo obupali, še preden bi lahko vse potrebne zvoke našteli. Tako pa nam pogon za to omogoči urejanje vseh teh zvokov kar prek vgrajenih nastavitev in parametrov.

2.3.4 Predstavitev razvijalnega okolja Unity

Za izdelavo projekta je bil izbran igralni pogon Unity 3D. Ta pogon je bil prvič predstavljen javnosti leta 2005. Namenjen je izdelavi in razvoju aplikacij ter iger za široko paleto različnih ciljnih platform. To pomeni, da računalniška igra, izdelana v njem, lahko delala z manjšimi popravki tako za Windows kot tudi za Mac OS, Linux, pa

tudi za Android in Apple IoS ... Deluje celo v poljubnem internetnem brskalniku, kar omogoča izjemno pokritost za ciljne uporabnike. Podpira tudi podatkovne baze v veliko različnih formatih, na primer v CSV formatu, v katerem bo realizirana podatkovna baza tega projekta.

Primarno je Unity namenjen izdelavi 3D računalniških iger, zelo enostavno pa je spremeniti vse te igre v 2D svet, če se razvijalec odloči za to možnost. Izdelati je mogoče tudi skoraj poljuben uporabniški vmesnik in ga obdelovati po potrebah. Prav tako vsebuje vse naštete funkcije igralnih pogonov, torej fizični, grafični pogon, umetno inteligenco, urejevalnik skript, zvoka ... Dovoljuje tudi sodelovanje različnih igralcev prek interneta (Networking). Uporabljati je mogoče že vgrajen pogon za to, ali pa nadgradnjo pogona za to – Photon[3].

Računalniško logiko se v Unity lahko programira s programskimi jeziki C#, JavaScript in Boo. V tej diplomski nalogi bo poudarek predvsem na jeziku C#, tudi vsi primeri kode so napisani v njem. Izbira jezika C# je povezana predvsem s tem, da je to najpogosteje uporabljen jezik v okolju Unity, saj je večina dokumentacije napisana v njem, prav tako pa se tudi večji delež drugih razvijalcev drži tega jezika. Po opisni moči so si vsi ti jeziki enakovredni, kar pomeni, da je, karkoli je možno napisati v enem izmed teh treh jezikov, možno napisati tudi v ostalih dveh.

[3]Opis pogona Photon dostopen na: https://www.photonengine.com/en/pun.

3 Analiza in definicija zahtev

V tem razdelku je tako z vidika naročnika kot tudi z vidika izvajalca projekta predstavljeno, katerim zahtevam mora program zadostovati. Pokazano je, kakšen mora biti uporabniški vmesnik ter točno katere elemente mora vsebovati. Določeni so tudi temeljni ciljni projekta v obliki nefunkcionalnih zahtev, pri vsaki pa je na kratko predstavljeno, kako je ta cilj dosežen. Na koncu so predstavljene še vse funkcije, ki jih mora v programu imeti uporabnik, in sicer s pomočjo UML diagrama primera uporabe, ter delitev sistema na podsisteme, ki je prikazana z diagramom toka podatkov.

Kot je poudarjeno v knjigah Describing Software Architecture with UML [5] in How UML is Used [3], se jezik UML zelo pogosto uporablja za grafično ponazoritev delovanja sistema, še posebej v predmetno orientiranih sistemih. S pomočjo anket so avtorji pokazali, da se razumljivost sistema drastično poveča z uporabo jezika UML v analizi, načrtovanju in v fazi implementacije. Uporaba UML jezika lahko tudi pripomore k zgodnejšem opažanju napak v sistemu in omogoča hitro popravljanje le-teh. Ravno zaradi te sistematičnosti bo v tej nalogi uporabljen jezik UML, in sicer v analizi zahtev in v načrtovanju sistema.

3.1 Analiza potreb uporabniškega vmesnika

Cilj tega projekta je zgolj realizacija igranja te namizne igre in doseg vseh zahtev za njeno nemoteno delovanje. Izdelava igralčevega kupčka, izbor herojev in podobni elementi niso vključeni. Tako bo veljala predpostavka, da si je uporabnik že izbral kupček kart in heroje, s katerimi želi igrati. V temu razdelku je opisan uporabniški vmesnik za igranje namizne igre.

V splošnem lahko igro igra poljubno število igralcev med ena in štiri. V tem projektu pa je poudarek na enem igralcu, ki igra igro proti (v našem primeru) računalniku. Elementi, ki morajo biti prikazani na zaslonu, morajo biti:

1. Igralčeve karte v roki.
2. Igralčeve mrtve karte.
3. Nasprotnikove karte v pripravljenosti.
4. Nasprotnikove mrtve karte.
5. Bojišče.
6. Nasprotnikov in igralčev kupček.
7. Ostali gumbi za vodenje igre.

Poleg tega morajo vse karte na zaslonu biti dovolj velike, da lahko igralec prebere tekst iz njih (ki je včasih nekoliko majhen), oziroma da vmesnik omogoča povečanje poljubne karte kadarkoli. Iz vmesnika mora biti razvidno tudi kateri gumbi so kadarkoli na voljo uporabniku in kateri ne (recimo ne more sprožiti napada v fazi okrevanja). Zelo priporočljivo je tudi, da je razvidno katero karto lahko trenutno igralec uporabi (plača za igranje, sproži kak efekt ...). Poleg tega je pomembno, da igralec ves čas ve, v kateri fazi se trenutno igra nahaja, torej da je to jasno razvidno iz vmesnika.

Čeprav trenutno govorimo o uporabniškem vmesniku za enega igralca, mora vmesnik omogočati tudi nadgradljivost za prihodnost, ko bo lahko več igralcev igralo isto igro vzporedno (idealno bi bilo potrebno omogočati nadgradljivost za do štiri igralce).

3.2 Analiza zahtev delovanja igre

V tem razdelku so predstavljene različne nefunkcijske zahteve, ki jih bo mogel vsebovati končni program. Ker v tem primeru nimamo dveh oseb, ki bi se pogajale o pomembnosti različnih nefunkcionalnih zahtev, je v tem razdelku opisana že rešitev za vse te nefunkcijske zahteve, ki bodo ugotovljene kot potrebne.

Določitev platforme

Za prvi prototip igre bo zadostovalo, da je igro mogoče igrati samo prek računalnika na platformi Windows. Unity že sam omogoča, da z minimalnimi popravki omogočimo delovanje igre še na drugih platformah (kasneje bo cilj tudi zagotoviti delovanje igre na platformi android za mobilne telefone, ampak to presega vsebino tega projekta). Zato so tudi v kodi že predhodno uporabljene le tiste knjižnice, ki se jih da uporabljati na poljubni platformi.

Fleksibilnost programa

Ker se igra Lotr: LCG vsaka dva meseca razširi z novimi scenariji in kartami, mora biti program zelo fleksibilno zasnovan in omogočati poljubne spremembe v kartah ter tudi čim lažje dodajanje novih kart. Vsak nov dodatek za igranje prinese s sabo nekaj novih mehanik (efektov), ti pa morajo biti vneseni tako, da ni ogrožena interakcija z ostalimi efekti, ter da se vsi efekti vedno zgodijo v smiselnem zaporedju.

Kot navaja avtor McConnell v Code Complete [7], je zelo pomembna berljivost kode, ker se večkrat program bere, kot je popravljen, zato je vitalnega pomena tudi, da je koda lepo berljiva (Self-documenting). Tak način pisanja kode zmanjšuje možnosti za napake pri popravljanju (dodajanju novih funkcionalnosti), pospeši pa tudi proces razumevanje določenega dela kode (včasih celo izboljša razumljivost). To se izkaže za zelo pomembno lastnost pri večjih projektih, ko dostopa do nekega dela kode veliko ljudi in je zelo pomembno da delček kode zelo hitro razumejo. Pri izdelavi projekta je bilo uporabljenih veliko tehnik za izboljševanje berljivosti, ki jih avtor v tej knjigi opisuje.

Skalabilnost programa

S stalnim prihodom novih kart ni problem samo v dodajanju novih efektov, ampak tudi v povečanju celotnega obsega vseh kart. Težava lahko nastane s tem, da je bila naša podatkovna baza za vsebovanje vseh kart mišljena za začetnih 200 kart, po nekaj mesecih pa mora biti baza že velika za nekaj 1000 kart, s tem pa se lahko zruši celotni sistem. Problemi lahko nastanejo tudi pri vejitvah, ker dodajanje novih kart poveča celotno število efektov, ti pa povečajo potrebo po vejitvah – sledi zmanjšanje berljivosti vejitev, povečanje časa porabljenega za iskanje pravega dela kode ... Skratka, program mora biti pripravljen na stalno rast števila kart, vseh obstoječih efektov in števila scenarijev.

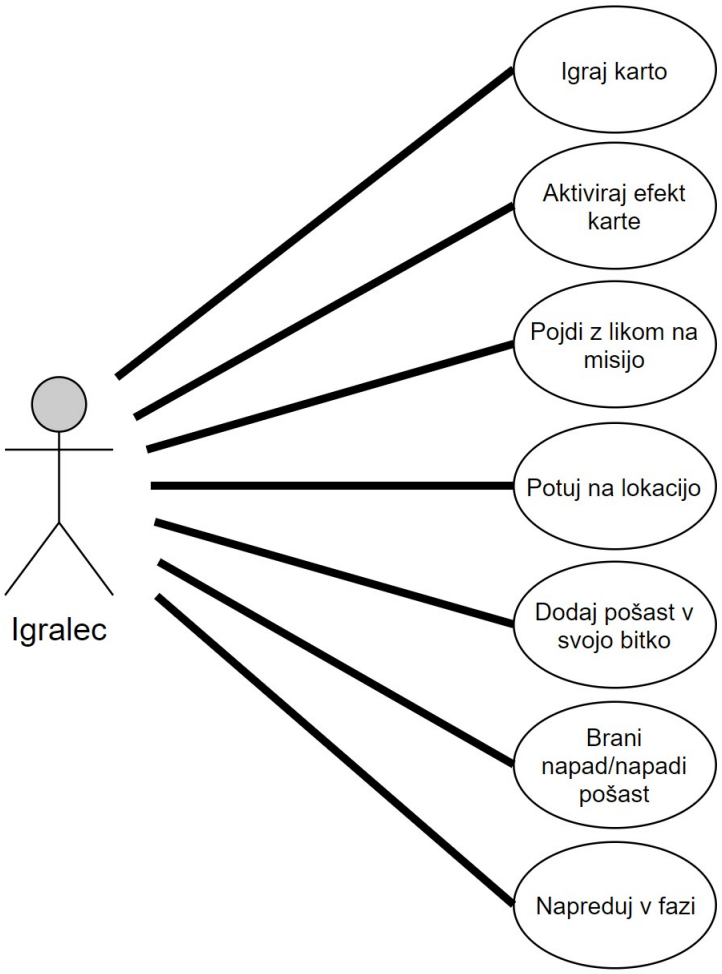
Omogočanje testiranja programa

Pri programih, v katerih se pričakujejo velike in/ali pogoste spremembe v njihovem delovanju, je nujno potrebno vedeti, ali po določenih popravkih in vnosu novih funkcionalnosti še vedno delujejo vsi deli enako kot prej. Potrebno je namreč imeti neko serijo testnih primerov, ki so avtomatizirani in omogočajo vpogled v to, ali program še vedno pri nekem vnosu poda enak rezultat, kot ga je podal pred popravki.

Avtor John Sonmez v knjigi The Complete Software Developer's Career Guide [10] močno poudarja pomembnost tako imenovanih „unit testov“ v projektih. To pomeni, da se majhni segmenti kode preverjajo pri vsakem novem popravku. Ti testi se izdelujejo še preden se napiše celotna koda, ki jo morejo preveriti, z namenom preverjanja pravilnega pisanja kode. Tako ti testi služijo dvema namenoma: najprej preverijo, ali je vsak segment kode pravilno napisan, nato še pri vsakem popravku programa preverijo, ali vsi prej napisani segmenti še vedno pravilno delujejo. Tako z rastjo programa hitro narašča tudi število Unit testov, ki lahko hitro dosežejo več tisoč testnih primerov.

3.3 Opis primera uporabe

Na sliki 1 je prikazan UML diagram primera uporabe. Ta diagram prikazuje, katere funkcije mora imeti igralec znotraj igre. Funkcije so čisto osnovne, te pa ima na razpolago znotraj določenih faz in pod zelo različnimi pogoji. Cilj programa je, da bo igralcu omogočal vse potrebne funkcije ob pravilnem času. Pomembno je, da so vse funkcije sprožene natanko tedaj, kot je to zapisano v uradnih pravilih in da bodo proizvedle točno tak rezultat, kot ga igralec pričakuje.

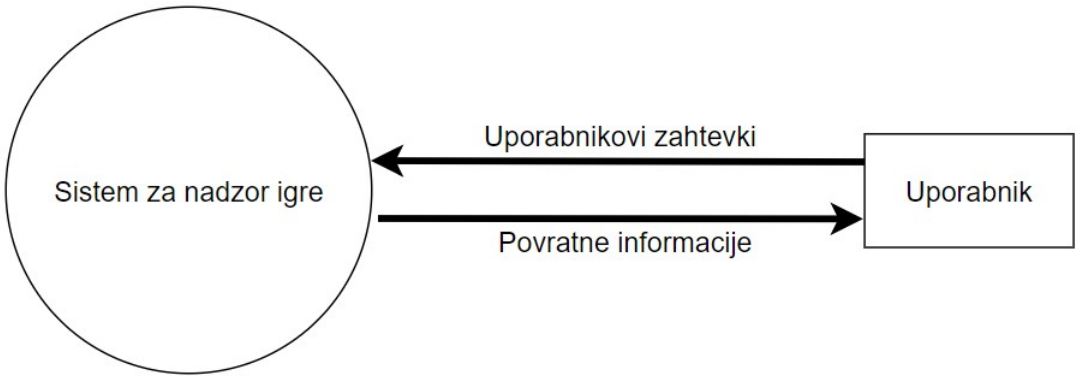


Slika 1: UML Diagram primera uporabe za sistem. Naštete so vse funkcije, ki jih mora imeti uporabnik znotraj igre, vsaka v svojem oblačku

V primeru prve funkcije (Igraj karto): Igralec lahko igra karto samo v svoji fazi načrtovanja, ki poteka čisto na začetku vsakega kroga. Če pa je karta urok, potem lahko to karto igra kadarkoli. Za igranje karte mora plačati nekaj zlatnikov, torej mora sistem preveriti ali igralec sploh lahko plača določeno karto. Na koncu, ko je karta igrana, mora program preveriti kakršnekoli morebitne interakcije z vstopom karte na bojišče. Ostale funkcije bodo predstavljene v nadaljevanju.

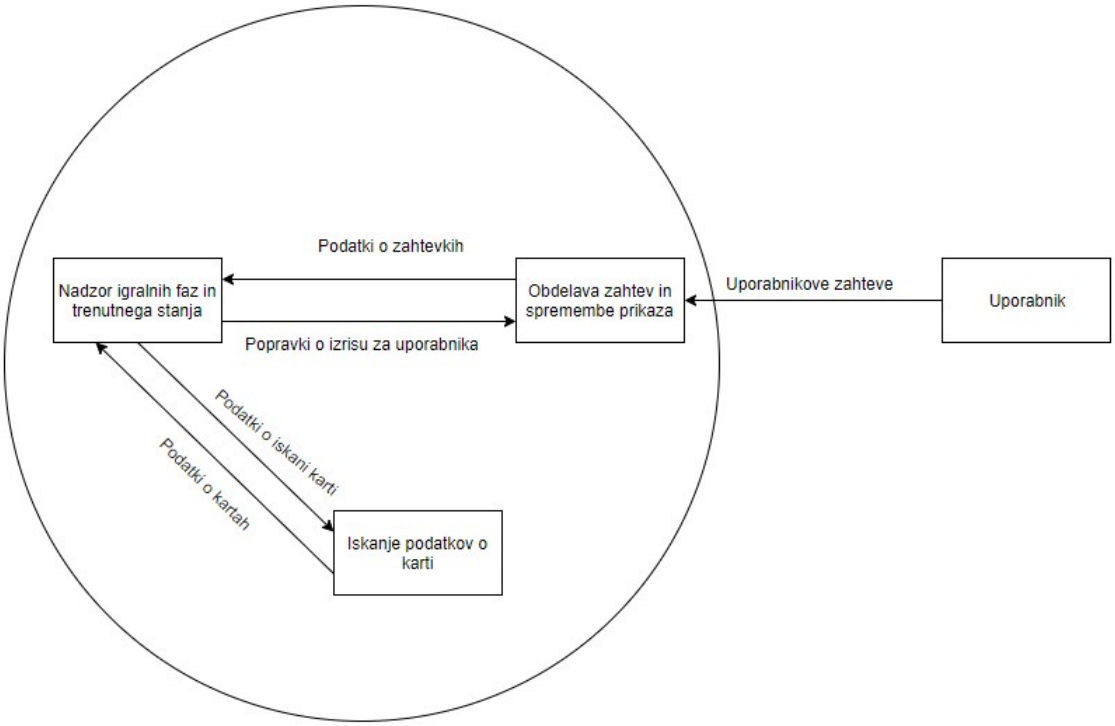
3.4 Diagram toka podatkov

Za prihodnjo razdelitev sistema na podsisteme je potrebno najprej imeti načrt, kako se bo program sploh delil v manjše podsisteme, v tem primeru na procese. To se najlažje razbere iz diagrama toka podatkov. Na sliki 2 je predstavljen DTP nivoja 0, torej najvišjega nivoja.



Slika 2: Diagram toka podatkov, nivo 0. Prikazuje najbolj splošen pogled na sistem, torej interakcijo med zunanjim uporabnikom in sistemom

Slika prikazuje najbolj enostavni pogled na sistem. Sistem vsebuje samo enega zunanjega uporabnika, ki hoče komunicirati s sistemom in prejemati neke povratne informacije glede na svoje vedenje. Komunikacija je torej dvosmerna, saj uporabnik stalno posreduje nove zahtevke s kliki na različne gumbe, sistem pa mora vse te zahtevke obdelati, ter jih na uporabniku prijazen način prikazati.



Slika 3: Diagram toka podatkov, nivo 1. Ta nivo prikazuje celotni sistem, ki je razdeljen v tri podsisteme, in prehajanje podatkov med posameznimi podsistemi ter interakcijo vsakega izmed njih z zunanjim uporabnikom

Na sliki 3 je prikazan diagram toka podatkov nivoja 1. Ta diagram že prikazuje,

kako se bo naš sistem razdelil v podsisteme, ki so v tem primeru procesi. Vsak proces dobi svojo dobro definirano nalogo, ki jo mora opravljati tekom delovanja programa. Za dobro programsko arhitekturo je potrebno čimbolj zmanjšati povezljivost med procesi in povečati kohezijo znotraj njih. To pomeni, da je naloga vsakega procesa zelo specifična in jo mora proces znati opraviti s čim manj komunikacije z drugimi procesi. Torej komunikacija med procesi mora biti minimalna, saj to olajšuje kasnejše popravke v sistemu in olajšuje dodajanje novih funkcionalnosti.

Sistem ima tako tri glavne procese, ki skrbijo za pravilni potek programa. Ti trije procesi so:

1. Nadzor igralnih faz in trenutnega stanja.
2. Obdelava zahtev in spremembe prikaza.
3. Iskanje podatkov o karti.

Ti procesi so zgrajeni hierarhično, kar pomeni, da se tudi sami delijo na manjše podprocese po metodi „Deli in vladaj“. Ta metoda je večkrat uporabljena v računalništvu, saj lahko zelo olajša razumevanje problema, ali v našem primeru, procesa. Za nek širok problem je včasih zelo zahtevno poiskati rešitev, zato se pogosto tak problem porazdeli na manjše in enostavnejše podprobleme. Ideja je v tem, da če pametno razdelimo problem na podprobleme, lahko potem s pomočjo rešitve podproblemov sestavimo rešitev celotnega problema. Manjše probleme je namreč veliko lažje reševati kot velike. Pri razumevanju tega problema je tudi zahtevno razumeti, kaj točne mora celoten sistem početi. Če razdelimo sistem na te procese, je že veliko lažje načrtovati končno rešitev. Ravno tako, ko bomo recimo sistem za nadzor igralnih faz in trenutnega stanja razdelili na podprocese, bo veliko jasneje kaj točno mora ta podproces početi.

4 Načrtovanje sistema

Avtorji knjige Systems Analysis and Design [2] so pojasnili, da jedro vsakega uspešnega projekta predstavlja faza načrtovanja sistema. Tu se namreč določi zgradbo sistema, njegovo delitev na podsisteme in načrtovanje le-teh. Faza načrtovanja lahko poteka strogo pred fazo izvedbe, lahko se pa tudi prepleta z njo, odvisno od izbire načrtovalnega modela. Najbolj uporaben model za sekvenčno izdelavo programske opreme je „waterfall model“, ki predvidi razvoj s strogim prehodom iz faze v fazo. Za ta projekt je bil uporabljen dinamični model, in sicer iterativni model načrtovanja. Ta predvidi več ciklov izdelave, vse od načrtovanja do izvedbe in na koncu do testiranja, po potrebi pa se še enkrat ponovi vse faze (ali večkrat). Zgodi se namreč lahko, da je pri testiranju ugotovljena nekakšna napaka in jo je treba pred dokončanjem projekta tudi odpraviti. Zato se je mogoče vrniti v fazo načrtovanja sistema in dopolniti načrt s predlagano rešitvijo. To pa se lahko nato vnese v program, kjer se izvede potrebne teste za preveriti, ali je bila težava odpravljena. Če ni bila, se lahko ta postopek ponovi.

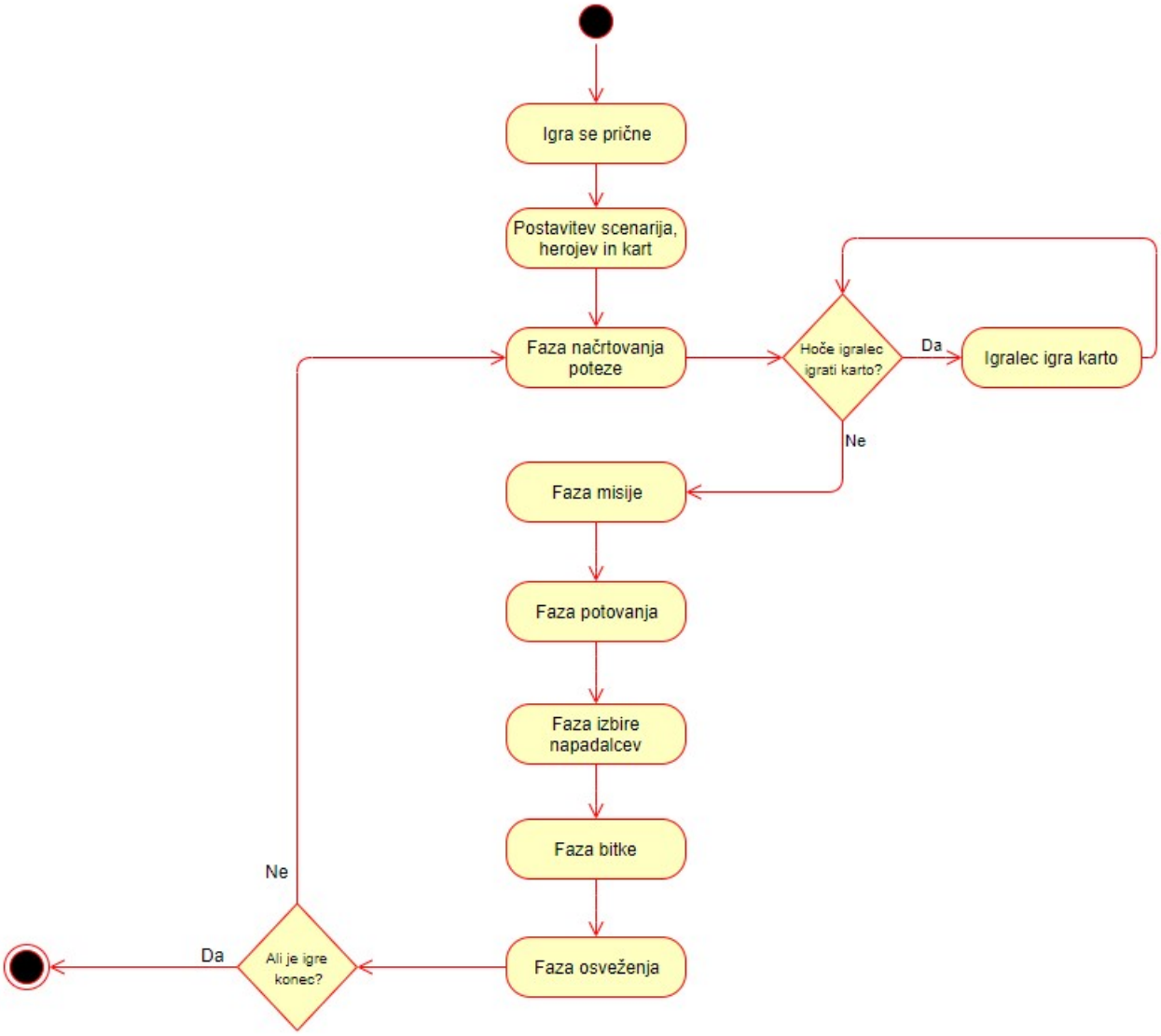
Raziskave, ki so bile poudarjene v knjigi Iterative and incremental developments. a brief history[6], so potrdile pomembnost iterativnih modelov proti osnovnemu linearnemu modelu. Po tem, ko so postali iterativni modeli širše poznani in se je razvila njihova uradna različica, se je zadovoljstvo strank s končnim produktom drastično izboljšala, tudi stroški popravkov so padli, čeprav se je čas razvoja nekoliko povečal.

Kot je bilo že prikazano na sliki 3, je celotni sistem razdeljen na tri procese, ki bodo predstavljali osnovni gradnik programa. V tem razdelku bo vsak od teh treh procesov podrobno obrazložen in tudi načrtovan. Kot je bilo že poudarjeno, je glavni cilj čim manjša komunikacija med procesi, torej da so procesi čim bolj neodvisni drug od drugega. Zato bo pri načrtovanju poseben poudarek na komunikaciji med procesi. V tem razdelku so predstavljeni vsi trije podprocesi, za boljšo razumljivost, pa so dodani UML diagrami, ki prikazujejo delovanje posameznega podprocesa.

4.1 Nadzor igralnih faz in trenutnega stanja

Proces za nadzor igralnih faz in trenutnega stanja skrbi za vso računalniško logiko, ki se mora neki trenutek izvajati. Vsaka računalniška igra vsebuje tako imenovan „Main loop“, ki predstavlja neskončno zanko igre, katera se izvaja v neskončnost. Ta zanka stalno preverja, ali je uporabnik izdal nov ukaz in/ali je čas, da se nekaj v igri spremeni. Če je vsaj eden od teh dveh pogojev resničen, mora zanka poskrbeti za obdelavo zahtev in njihovo izvedbo, kasneje pa tudi za izris na uporabnikov ekran. Proces za nadzor igralnih faz in trenutnega stanja simulira izvajanje te neskončne zanke ter preverja uporabnikove zahtevke, če se lahko trenutno sploh izvedejo in če se, jih mora tudi izvesti.

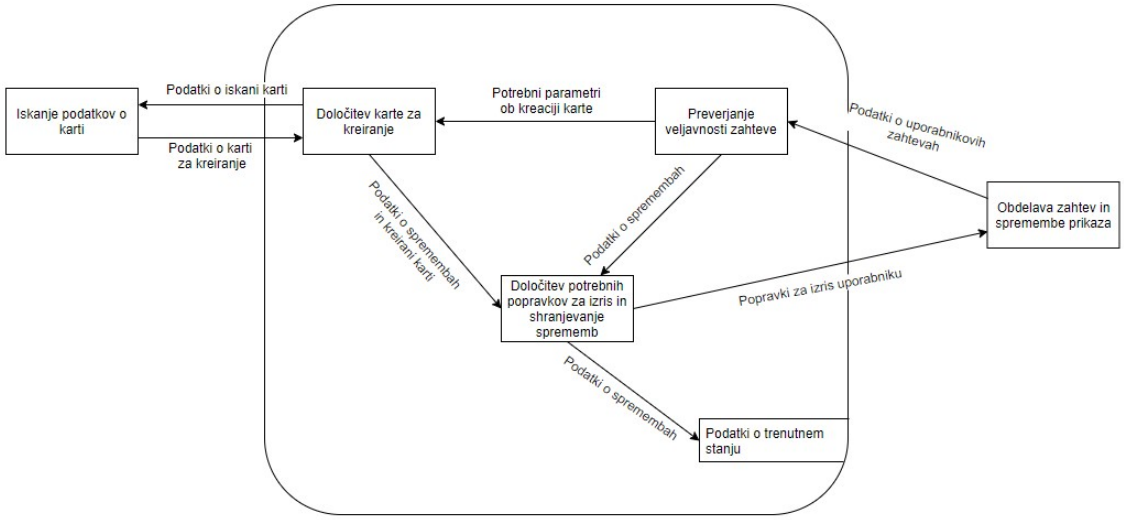
Proces je torej v grobem sestavljen iz dveh sestavnih delov. Eden izmed teh je shranjevanje trenutnega stanja, torej v poljubnem trenutku mora biti razvidno, v kateri fazi se program nahaja, katera karta je v kateri igralni coni, podatki o kartah, itd. Drugi sestavni del pa je nadzor nad igralnimi fazami, kar pomeni, da se mora preklapljanje med fazami izvajati v pravilnem zaporedju ter da program ve, kaj je v vsaki fazi mogoče početi in kaj je prepovedano.



Slika 4: UML aktivnostni diagram prikazuje potek faz znotraj poteze. Prikazano je prehajanje med fazami in osnovne uporabnikove možnosti znotraj faze

Diagram poteka na sliki 4 prikazuje, kako izgleda posplošena neskončna zanka igre. Igra se prične s svojo inicializacijo, ko se uredijo vsi potrebni začetni parametri. Nato se postavi bojišče, pripravijo se izbrani heroji, scenarij in izbrani igralčev kupček. Ko je vse to narejeno, se začne prva faza, v kateri ima igralec možnost nadzora. V fazi načrtovanja poteze lahko namreč na bojišče poda poljubno število kart, ki jih ima v

roki. Ko zaigra vse izbrane karte, gre igra v naslednjo fazo, torej v fazo misije. Ko je le-ta končana, ji sledi faza potovanja in tako naprej, dokler ne pride do faze osvežitve, po kateri se lahko igra ali zaključi (če imajo igralci preveč grožnje) ali pa nadaljuje z naslednjo potezo (v splošnem se lahko igra zaključi tudi v poljubni drugi fazi, slika je poenostavljena). V primeru, da se igra nadaljuje, se spet prične faza načrtovanja poteze, celotni cikel pa se ponavlja, dokler se igra ne konča.



Slika 5: Diagram toka podatkov, nivo 2 – nadzor faz. Diagram prikazuje podsistem za nadzor igralnih faz in trenutnega stanja ter njegovo delitev na manjše dele podsistema

Na sliki 5 je prikazan drugi nivo diagrama toka podatkov, in sicer za proces nadzora igralnih faz in trenutnega stanja. Proces kot celota dobi enake informacije kot v nivoju 1 (slika 3). Po metodi „Deli in vladaj“ je bil v tem koraku proces razdeljen na tri podprocese, in sicer:

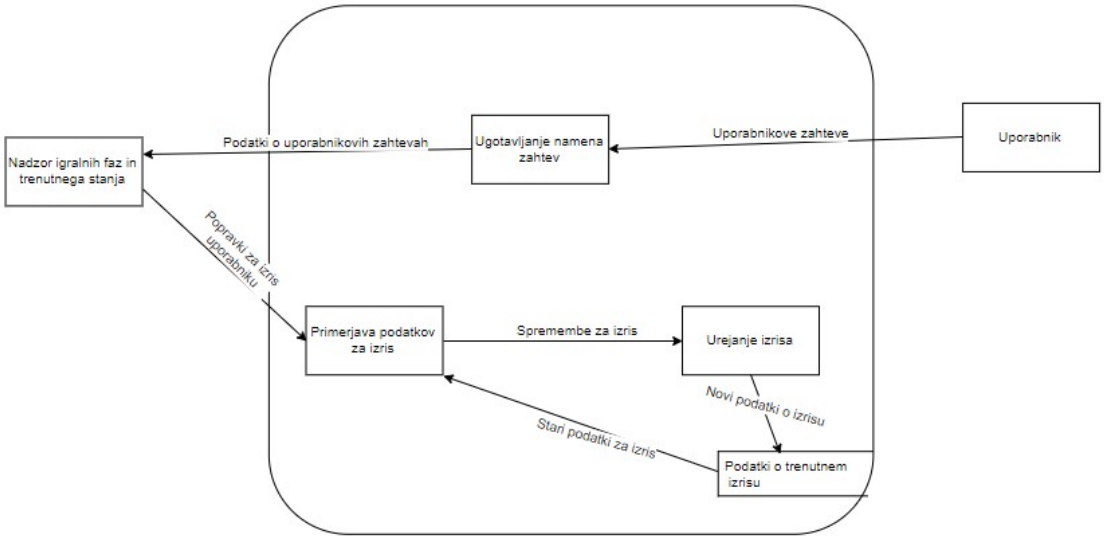
1. Preverjanje veljavnosti zahteve.
2. Določitev karte za kreiranje.
3. Določitev potrebnih popravkov za izris in shranjevanje sprememb.

Poleg tega pa ta proces shranjuje podatke tudi v podatkovno bazo, in sicer podatke o trenutnemu stanju. Celotni proces se začne, ko dobi podatke o uporabnikovih zahtevah iz procesa za obdelavo zahtev in spremembe prikaza. Te podatke obdela podproces za preverjanje veljavnosti zahteve. Ta podproces bodisi ugotovi, da je zahteva veljavna (da se v trenutnem stanju lahko zgodi), ali pa da ni veljavna in zahtevo zavrne. Poleg tega ta podproces ugotovi, ali je za izpolnjevanje uporabnikove zahteve potrebna kreacija nove karte. Če je kreacija potrebna, poda potrebne parametre za kreacijo podprocesu za določitev karte za kreiranje, če pa kreacija ni potrebna, pa podatke o novih spremembah samo sporoči podprocesu za določitev potrebnih popravkov za izris in shranjevanje sprememb. Podproces za določitev karte za kreiranje ustvari novo karto

s pomočjo procesa za iskanje podatkov o karti. Na novo ustvarjeno karto poda procesu za določitev potrebnih popravkov za izris in shranjevanje sprememb, ta pa shrani potrebne spremembe v podatkovno bazo za hranjenje podatkov o trenutnem stanju, ter javi potrebne popravke za izris nazaj procesu za obdelavo zahtev in spremembe prikaza.

4.2 Obdelava zahtev in spremembe prikaza

Proces za obdelavo zahtev in spremembe prikaza simulira delovanje uporabniškega vmesnika v sistemu. Ta proces ima dve funkciji. Prva je, da ugotovi, od kod prihaja uporabniška zahteva (iz katerega gumba), in da ugotovi, kaj hoče uporabnik s svojo zahtevo narediti, ter da vse to sporoči naslednjemu procesu. Druga funkcija pa je, da poskrbi za izris sprememb, ki mu jih drugi proces naroči. S tem dosežemo skladnost uporabnika z dogajanjem, torej da je uporabniku vedno na voljo učinkovit prikaz trenutnega dogajanja v sistemu.



Slika 6: Diagram toka podatkov, nivo 2 – obdelava zahtev. Diagram prikazuje podsistem za obdelavo zahtev in spremembe prikaza. Prikazana je tudi delitev na manjše dele podsistema in njegova interakcija z uporabnikom in podatkovno bazo.

Na zunanji strani sistema je uporabnik, ki hoče z neko zahtevo spremeniti trenutno stanje v igri. Ta uporabniška zahteva gre najprej do podprocesa za ugotavljanje namena zahtev, ki ugotovi, točno od kod je ta zahteva prišla in kam jo je treba posredovati, da se bo primerno obdelala, ter katere podatke še potrebuje sistem za obdelavo zahteve. Vse to sporoči naprej sistemu za nadzor faz in trenutnega stanja, ki bo preveril, ali je zahteva trenutno izvedljiva, ali ne. V primeru da se zahteva izvede, se lahko pričakuje povratno sporočilo, ki vsebuje informacijo o tem, kaj mora biti trenutno izrisano za uporabnika. Te podatke najprej sprejme podproces za primerjavo podatkov za izris. Ta proces najprej prebere iz baze trenutne podatke na izrisu in jih primerja s podatki, ki bi morali trenutno biti izrisani. Če so potrebne spremembe, jih javi podprocesu za

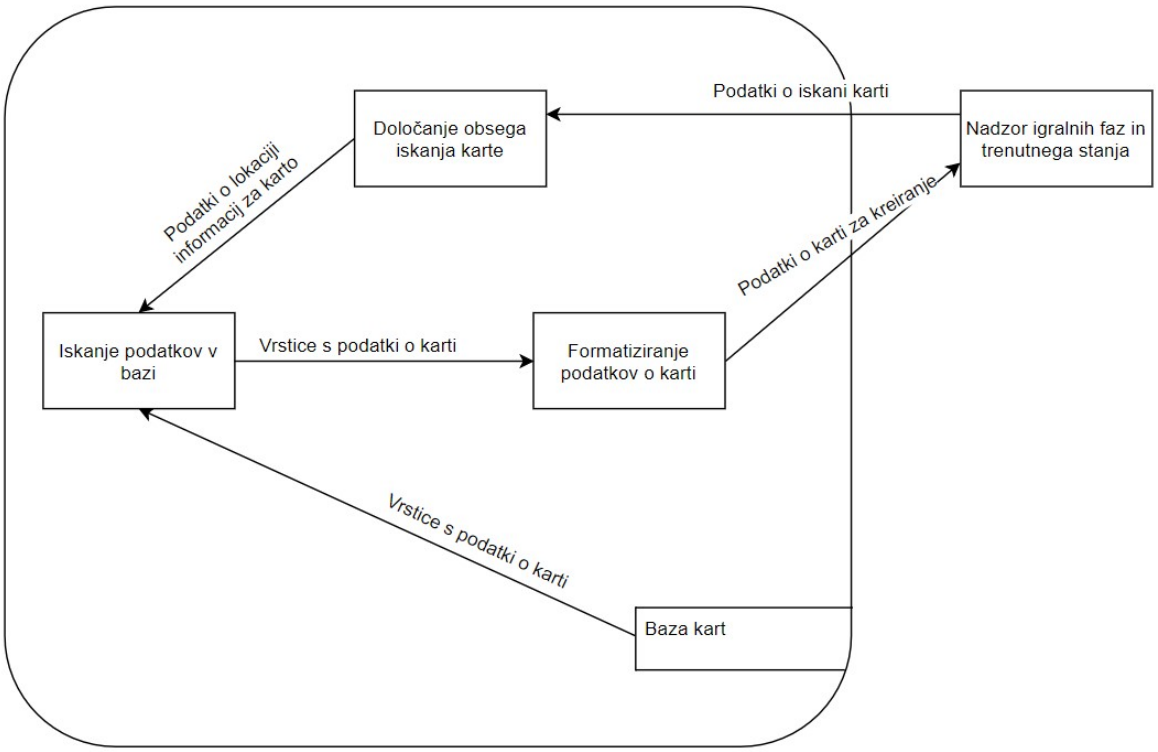
urejanje izrisa. Ta podproces samo zamenja trenutne informacije izrisa s spremenjenimi.

Podproces za obdelavo zahtev in spremembe prikaza je sprogramiran v kodi, poslužuje pa se že definiranih funkcij igralnega pogona Unity za uporabniški vmesnik. Torej sam uporabniški vmesnik ni sprogramiran, temveč je le postavljen iz že ponujenih elementov. Ti elementi so lahko statični, torej da služijo zgolj prikazu trenutnega stanja, lahko pa tudi ponujajo uporabniku neke vrste dostop do sprememb igralnega poteka. Tak je recimo gumb za končanje trenutne faze, ki je postavljen v vmesnik kot ponujen element igralnega pogona Unity, ampak se ob kliku nanj sproži neka funkcija iz tega podprocesa, ki je opisana v kodi. Tako komunikacija med definiranimi elementi iz Unity in projektno kodo poteka neprestano v uporabniškem vmesniku.

Podatkovna baza je tudi realizirana kar v vmesniku igralnega pogona Unity, namesto v uporabniški kodi ali v zunanji podatkovni shrambi. Razlog za to je, da je nadzor nad spremembami prikaza realiziran kar pri popravljanju trenutnih predmetov v prikazu. Recimo, če je potrebno neko karto obarvati, potrebujemo le referenco te karte, preko te reference pa spremenimo njene lastnosti. Podobno je za lokacijo kart in ostalih predmetov. Trenutne informacije o kartah so večinoma spravljene kar v predmetu, ki predstavlja karto v obliki privatnih spremenljivk. S tem postane pregled nad stanjem trivialen, saj lahko prek spremenljivk enostavno dostopamo do trenutnega stanja v prikazu, popravljanje pa je nekoliko težje, ker moramo najprej popraviti spremenljivko, ki opisuje trenutno stanje, nato pa še popraviti dejansko stanje karte. Ta princip ima torej določeno stopnjo redundantnosti, ampak le-to zmanjša možnost napake in pospeši dostop do trenutnega stanja (pričakuje se lahko, da bo program večkrat samo pogledal trenutno stanje, kot ga dejansko popravljal). Zaradi lažjega popravljanja in branja kode, bodo te privatne spremenljivke enkapsulirane, kar pomeni da bo dostop do njih omejen prek posebej definiranih funkcij za ta namen. Vsakič, ko se bo bralo stanje spremenljivke, bo enkapsulacija prevezala na dejansko stanje te karte, ki je shranjeno v vmesniku. Ko pa se bo spremenljivko popravljalo, bo funkcija namenjena popravljanju istočasno še ustrezno popravila dejanski izgled v vmesniku.

4.3 Iskanje podatkov o karti

Podproces za iskanje podatkov o karti je namenjen komunikaciji z bazo podatkov kart. Vhod v sistem je podatek o karti, ki jo hoče prejšnji proces kreirati. Podproces pa mu mora vrniti v izhodu vse potrebne podatke za kreacijo nove karte. Potrebujemo torej podatkovno bazo, ki bo imela podatke o več tisoč kartah, urejen mora biti dostop do vseh podatkov, poleg tega pa mora biti ta dostop dovolj hiter. Dodatno mora imeti baza omogočeno urejevanje podatkov o kartah, saj se včasih zgodi, da se kakšno karto tudi popravi. Prav tako mora ločeno podpirati različne tipe kart, saj ima vsaka karta drugega tipa drugačne lastnosti (recimo zavezniki imajo določeno število točk napada, medtem ko lokacije tega sploh nimajo).

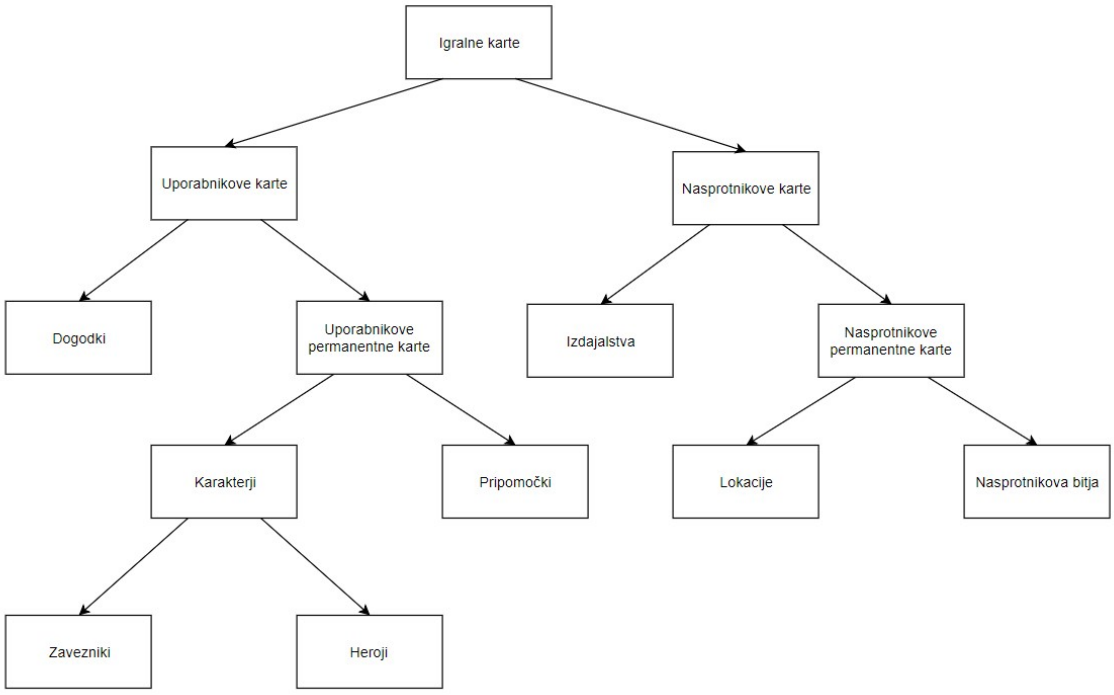


Slika 7: Diagram toka podatkov, nivo 2 – iskanje podatkov o karti. Diagram prikazuje podsistem za iskanje podatkov o karti. Prikazuje njegovo delitev na manjše dele podsistema in interakcijo med njimi, podatkovno shrambo ter sistemom za nadzor igralnih faz in trenutnega stanja

Na sliki 7 je prikazano, kako izgleda drugi nivo diagrama toka podatkov za iskanje podatkov o karti. V tem podprocesu je najpomembnejši element podatkovna baza, ki vsebuje vse podatke o kartah. Podproces v svojem vhodu dobi podatke o iskani karti, s pomočjo le-teh pa določi, kje v bazi bo določeno karto sploh iskal. Karte so v bazi namreč lahko ločene in je njeno lokacijo potrebno včasih tudi izračunati. Ko je lokacija izračunana, se posreduje podprocesu za iskanje podatkov v bazi, ki je omenjen posebej ravno zaradi tega, ker mora bit sprogramiran posebej, saj ga Unity ne omogoča. Podatki so v bazi shranjeni v vrsticah, kjer vsaka vrstica opiše eno karto. Tako proces s pomočjo določene lokacije zahteva vrstico na tej lokaciji in to vrstico posreduje podprocesu za formatiranje podatkov o karti. Ta podproces mora iz vrstice razbrati, kaj določen parameter v njej pomeni, ter ga pravilno umestiti v predmet, ki bo predstavljal karto. Ko je ta predmet kreiran, se njegovi podatki pošljejo nazaj procesu za nadzor igralnih faz in trenutnega stanja, kjer se bo kreirala tudi dejanska karta, ki se bo izrisala na zaslonu.

Podatkovna baza je realizirana kar v vmesniku igralnega pogona Unity, namesto v uporabniški kodi ali v zunanji podatkovni shrambi. Razlog za to je, da je nadzor nad spremembami prikaza realiziran kar v popravljanju trenutnih predmetov v prikazu. Recimo, če je potrebno neko karto obarvati, potrebujemo le referenco te karte, preko te reference pa spremenimo njene lastnosti. Podobno je za lokacijo kart in ostalih predmetov. Trenutne informacije o kartah so večinoma spravljene kar v predmetu, ki

predstavlja karto, v obliki privatnih spremenljivk. S tem postane pregled nad stanjem trivialen, saj lahko prek spremenljivk enostavno dostopamo do trenutnega stanja v prikazu, popravljanje pa je nekoliko težje, ker moramo najprej popraviti spremenljivko, ki opisuje trenutno stanje, nato pa še popraviti dejansko stanje karte. Ta princip ima torej določeno stopnjo redundantnosti, ampak le-to zmanjša možnost napake in pospeši dostop do trenutnega stanja (pričakuje se lahko, da bo program večkrat samo pogledal trenutno stanje, kot ga dejansko popravljal). Zaradi lažjega popravljanja in branja kode, bodo te privatne spremenljivke enkapsulirane, kar pomeni da bo dostop do njih omejen preko posebej definiranih funkcij za ta namen. Vsakič, ko se bo bralo stanje spremenljivke, bo enkapsulacija prevezala na dejansko stanje te karte, ki je shranjeno v vmesniku. Ko pa se bo spremenljivko popravljalo, bo funkcija namenjena popravljanju istočasno še ustrezno popravila dejanski izgled v vmesniku.



Slika 8: Hierarhija igralnih kart prikazuje dedovanje lastnosti posameznih abstraktnih tipov kart. Na najvišjemu položaju je najbolj splošen tip karte, z nižanjem nivojev so pa tipi vedno bolj specifični

Na sliki 8 je prikazana celotna hierarhija vseh igralnih kart. V listih drevesa so vse dejanske karte, ki obstajajo v igri, v vmesnih točkah pa so abstraktni predmeti. Slika prikazuje, kako so različni tipi kart povezani med sabo, na podlagi skupnih lastnosti, ki jih imajo. Kot je na primer razvidno iz slike, so si zavezniki in heroji zelo podobni, saj imata njuna predmeta skupnega starša. S tem, ko razdalja med predmeti narašča, so si tudi predmeti med sabo vse manj podobni. Tako na primer zavezniki in lokacije nimajo skupnega skoraj nič. V programu bo ta struktura predstavljena z dedovanjem, torej ko imajo predmeti nekaj skupnih lastnosti, navadno tistih, ki jih določajo vmesni abstraktni primeri.

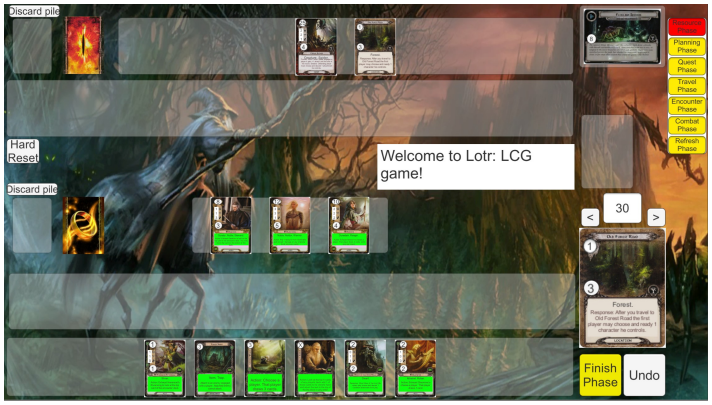
Uporaba abstraktnih predmetov pomaga na dva načina. Prvi je, da omogoča lažje dedovanje in s tem ne pride do redundantnosti kode. Recimo zavezniki in heroji imajo skupno metodo, ki preveri koliko napada imajo. Če nebi bil uporabljen abstraktni predmet lika, ki vsebuje metodo za preverjanje napada, bi bilo potrebno to kodo definirati tako za zaveznike kot za heroje (čeprav delujeta identično). Tudi če bi bilo potrebno to metodo kasneje spremeniti oziroma popraviti, bi jo bilo potrebno popraviti na vseh mestih, kjer se nahaja (lahko tudi v vseh sedmih razredih). Druga uporabna lastnost abstraktnih predmetov pa je, da se nekatere karte, oziroma njihovi efekti, sklicujejo na like (ki so predstavljeni ali kot zavezniki ali kot heroji), čeprav točno določen tip lik na karti ne obstaja. Z uvedbo abstraktnega razreda se lahko doseže direktno preverjanje, če je nek predmet lik glede na njegov podtip.

C# je predmetno orientiran jezik, zato vsebuje zahtevnejše programske rešitve, kot so na primer polimorfizem in dedovanje. Kot predstavljata tudi avtorja v Fundamentals of Computer Programming with C# [8], omogoča jezik C# zelo enostavno primerjanje podtipov med seboj. V praksi to pomeni, da če iz istega razreda (v tem primeru iz razreda Karta) dedujeta tip zaveznik in lik, lahko njegove splošne lastnosti enostavno primerjamo med seboj, kot da če bi bila tipa karta (ali poljuben drugi starš, ki jim je skupen). Obstaja tudi direktno primerjanje tipov, ki pove, ali neka karta deduje iz drugega tipa ali ne. Poleg tega se lahko tudi bolj specifični objekt obnaša tako, kot da bi bil bolj splošen predmet (na primer predmet tipa heroj se obnaša kot predmet tipa lik).

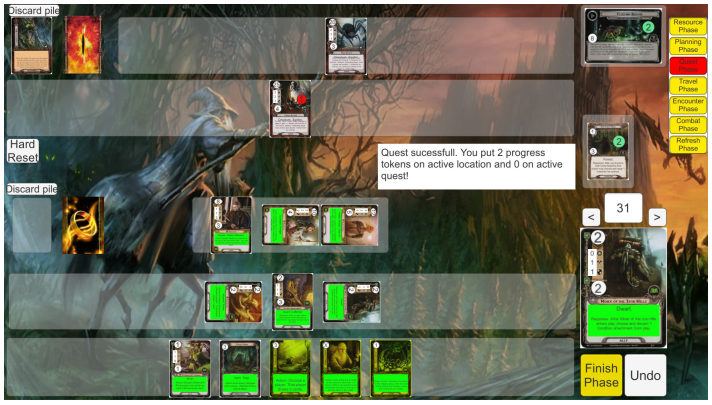
5 Izvedba projekta

Ta razdelek je namenjen opisu izvedbe projekta s pomočjo razvijalnega okolja Unity. Poudarek je na izvedbi glavnih treh podsistemov, ki so bili opisani v razdelku številka 4, omenjene pa so še nekatere druge bolj zanimive funkcije in metode v sistemu. Opisana je tudi podatkovna baza, ki je pokazana v programu Microsoft Excel. Prav tako je opisan postopek za iskanje po bazi, torej tako imenovane poizvedbe v bazi.

Vsi primeri kode so v programskemu jeziku C#, predstavljeni so tudi primeri iz vmesnika Unity za lažje razumevanje kode. Razdelek bo razdeljen na tri dele, po eden za vsak podsistem.



(a) Začetna postavitev igre, z nekaj nasprotniki v pripravljalnem prostoru, s tremi heroji in šestimi kartami v igralčevi roki



(b) Kompleksnejša postavitev po nekaj potezah igre, s poškodovanim nasprotnikom, napredkom na aktivni lokaciji in scenariju ter z nekaj utrujenimi heroji

Slika 9: Slika začetne postavitve in kompleksnejše scene v igri, ki prikazujeta razliko med enostavno in zahtevnejšo situacijo znotraj igre, ter prikaz te situacije

Na sliki 9 sta prikazana dva primera scen znotraj igre. Slika 9a prikazuje začetno postavitev, torej ko so heroji na bojišču, igralčeve karte v roki, nasprotnik pa ima samo začetne karte v pripravljalnem območju. Na sliki 9b pa je prikazana že nekoliko kompleksnejša scena, in sicer v potezi raziskovanja. Nekaj herojev je odšlo na raziskovanje trenutne lokacije, eno nasprotnikovo bitje že ima nekaj škode, igralec je tudi zaigral že tri zaveznike na bojišče in je ravnokar že uspešno raziskal lokacijo ter dal nanjo dve točki napredka. V nadaljevanju razdelka bo prikazano, kako je vsak izmed podsistemov, ki upravljajo s scenami, realiziran.

5.1 Izvedba podsistema za nadzor faz in trenutnega stanja

Podsistem za nadzor igralnih faz in trenutnega stanja ima znotraj tega projekta tri glavne funkcije, ki so:

1. nadzor igralnih faz,
2. nadzor trenutnega stanja,
3. kreacija karte.

Nadzor nad igralnimi fazami ima funkcija, ki jo sproži gumb za menjavo trenutne faze. Ob vsakem pritisku gumba mora biti igra v pravilnem stanju, da se lahko faza res zamenja. Ta funkcija mora tako hkrati primerjati, ali so izpolnjeni vsi pogoji za menjavo trenutne faze, in tudi uskladiti vse potrebne informacije ob nastopu zamenjave.

Nadzor trenutnega stanja je znotraj programa Unity decentraliziran. To pomeni, da ima vsak predmet svoje trenutno stanje, hkrati pa nosi informacije od vseh njegovih „otrok“, torej predmetov vsebovanih znotraj le-tega. Stanje predmetov večinoma preverja in spreminja ravno funkcija za nadzor igralnih faz, vendar imajo dostop do podatkov tudi druge funkcije.

Kreacija karte je proces, ki mora za svojo izvedbo imeti vse podatke o kreaciji karte iz podatkovne baze. Prikazano bo, kako sistem pošlje zahtevek za te podatke in kako s temi podatki kreira karto ter jo vstavi na potrebno igralno cono.

5.1.1 Nadzor igralnih faz

Nadzor nad igralnimi fazami se večinoma izvaja iz funkcije „ChangeTurn“, ki bo predstavljena v nadaljevanju. Glavni podatek, ki mora biti znotraj igre ves čas znan, je ta, v kateri fazi se trenutno nahaja igra. Najprej je potrebno definirati podatkovno strukturo, ki bo lahko zavzemala vse vrednosti trenutnih možnih faz.

Algoritem 1 Podatkovna struktura možnih faz

```
Variables
    public enum PossiblePhases {ResourcePhase, PlanningPhase, QuestPhase,
      TravelPhase, EncounterPhase, CombatPhase, RefreshPhase}
    public PossiblePhases currentPhase = PossiblePhases.ResourcePhase
end Variables
```

S kodo, ki je prikazana v algoritmu 1, smo definirali podatkovno strukturo „PossiblePhases“, ki lahko nosi vrednosti „ResourcePhase“, „PlanningPhase“ itd. Zavzema lahko vse možne faze in to po eno naenkrat. Nato smo še deklarirali javno spremenljivko „currentPhase“, ki opisuje trenutno potezo, v kateri se nahajamo. Za začetek smo jo nastavili na „ResoucePhase“, saj se igra v tej fazi prične. Ko bo potrebno znotraj igre prebrati, v kateri fazi se trenutno igra nahaja, je potrebno zgolj pogledati katero vrednost trenutno zavzema spremenljivka „currentPhase“. Ravno tako je potrebno za spremembo trenutne faze le zamenjati vrednost te spremenljivke.

Algoritem 2 Poenostavljena funkcija za zamenjavo poteze

```
procedure ChangeTurn
    if isMakingDecision == true then
        return
    else
        if isclickable == true then
            if currentPhase == PossiblePhases.ResourcePhase then
                currentPhase = PossiblePhases.PlanningPhase
            end if
            if currentPhase == PossiblePhases.PlanningPhase then
                currentPhase = PossiblePhases.QuestPhase
            end if
            if currentPhase == PossiblePhases.QuestPhase then
                currentPhase = PossiblePhases.TravelPhase
            end if
            if currentPhase == PossiblePhases.TravelPhase then
                currentPhase = PossiblePhases.EncounterPhase
            end if
            if currentPhase == PossiblePhases.EncounterPhase then
                currentPhase = PossiblePhases.CombatPhase
            end if
            if currentPhase == PossiblePhases.CombatPhase then
                currentPhase = PossiblePhases.RefreshPhase
            end if
            if currentPhase == PossiblePhases.RefreshPhase then
                currentPhase = PossiblePhases.ResourcePhase
            end if
        end if
    end if
end procedure
```

Funkcija „ChangeTurn“ v algoritmu 2 je poenostavljena koda za zamenjavo faze, v kateri se trenutno nahaja igra. Funkcija je znotraj sistema klicana samo prek gumba, ki je namenjen uporabniku za poskus končanja trenutne faze in začetka naslednje. Najprej mora funkcija preveriti, ali se trenutno izvaja kakšna odločitev, ki jo mora igralec sprejeti, in če se ta res izvaja, se faza ne sme zamenjati, dokler ni odločitev sprejeta. V tem primeru se s klikom na gumb ne zgodi nič. Če odločitve ni, se preveri, ali je trenutno mogoče klikniti na gumb, torej če ustrezajo vsi ostali pogoji oz. če se je končal potek dogodkov, ki se mora znotraj neke faze zgoditi. Če je temu tudi tako, se faza zamenja. Naslednje vrstice preverijo, v kateri fazi se trenutno igra nahaja in glede na to preklopi v naslednjo fazo. Ker je to poenostavljena različica, se ne zgodi nič drugega, kot da se preklopi faza. V realni kodi se mora glede na to, katera faza sledi, zgoditi določeno zaporedje dogodkov.

Algoritem 3 Funkcija za zamenjavo poteze v Resource phase

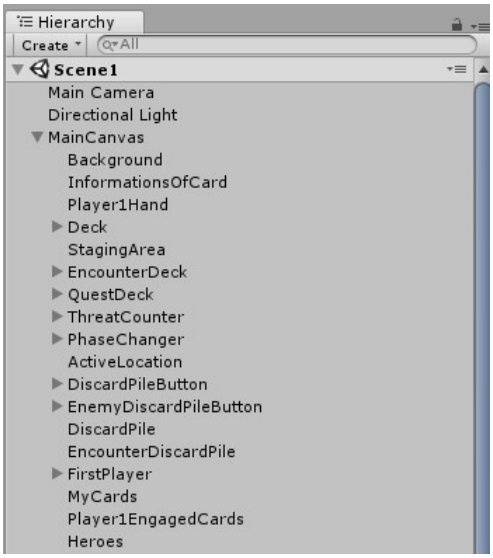
```
procedure ChangeTurn
    if isMakingDecision == true then
        return
    else
        if isclickable == true then
            if currentPhase == PossiblePhases.ResourcePhase then
                CheckForEndRoundTriggers()
                infoWindow.transform.Find("Text").ChangeText(ResourcePhase)
                hand.MakeEventsGlow(true)
                for all GameObject hero in heroes do
                    if hero != null then
                        hero.Coins += 1
                    end if
                end for
                deck.DrawCard()
                checkForEndPhaseTriggers()
            end if
        end if
    end if
end procedure
```

Del kode funkcije „ChangeTurn“ v algoritmu 3 prikazuje, kako v končnem produktu izgleda delček funkcije „ChangeTurn“, in sicer za prehod v fazo „Resource phase“. Začetna dva pogoja sta enaka kot v algoritmu 2, pojasnjena pa sta bila že v prejšnjem odstavku. Ko program zagotovo ve, da je potrebno zamenjati fazi v „Resource phase“, se zgodi naslednje zaporedje dogodkov. Najprej se zgodijo vsi dogodki kart, ki se sprožijo na koncu vsake poteze igralcev. Igra preveri vsako karto na bojišču, če ima kak tak efekt in če ga ima, se le-ta izvede. Ko so vsi efekti izvedeni, se na informativnem okencu spremeni tekst, ki uporabniku pove, da se trenutno nahajamo v fazi „Resource phase“, ter opiše, kaj se bo še zgodilo v tej fazi. Naslednji ukaz v izvedbi poišče skripto, ki se ukvarja z vsemi kartami v igralčevi roki, in naroči, naj se vse karte tipa „Event“ pobarvajo, kar pomeni, da jih lahko igralec trenutno zaigra. Več o tem bo prikazano

v podsistemu Obdelava zahtev in spremembe prikaza. Naslednji dogodek doda vsem herojem en kovanec, saj na začetku igralne poteze vsak heroj prejme en kovanec, ki ga lahko kasneje zapravi. To se zgodi samo, če je heroj še vedno živ, torej ni enak null. Tudi igralec v tej potezi še vleče karto (ta funkcija bo natančneje opisana v delu Kreacija karte). Na koncu vsake faze še preverimo, če se zgodijo kakšni dogodki na koncu te faze in jih izvedemo.

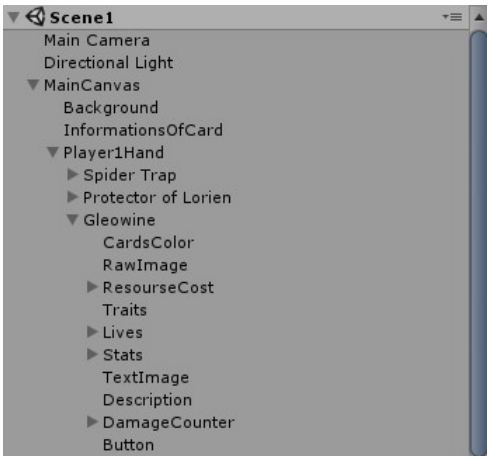
5.1.2 Nadzor trenutnega stanja

Kot je bilo že omenjeno, je celoten nadzor nad igralnim stanjem decentraliziran. Pogon Unity deluje na predmetnem sistemu, kar pomeni, da je vsaka stvar znotraj sistema neke vrste predmet. Nekemu predmetu lahko priredimo določene lastnosti (ki so lahko privatne ali pa javne) in tudi skripte obnašanja. V vsaki skripti mu lahko spet definiramo določene lastnosti in metode. Vsak predmet je tudi nekje vsebovan v celotni hierarhiji, ki vsebuje vse predmete znotraj neke scene. Dani predmet je gotovo otrok nekega drugega predmeta (ali otrok scene same), vsebuje pa lahko 0 ali več svojih otrok.



Slika 10: Hierarhija predmetov v glavni sceni prikazuje vse predmete, ki so na začetku igre prisotni. Vsak izmed njih ima v sceni svoj grafični prikaz in vsebuje določene lastnosti

Na sliki 10 je prikazana hierarhija predmetov pred začetkom igranja igre. Seveda se znotraj igre celotna hierarhija tudi spreminja, saj se v neko sceno lahko dodajajo novi predmeti (recimo karta, ki jo igralec povleče iz svojega kupčka), lahko pa se tudi predmet izbriše iz scene (npr. karta ni preživela bitke). Na sliki je razvidno, da ima trenutna scena „Scene1“ tri otroke, in sicer: „Main Camera“, „Directional Light“ in „MainCanvas“. Skoraj vsi ostali predmeti v sceni so otroci predmeta „MainCanvas“.



Slika 11: Hierarhija karte prikazuje predmete, ki jih vsebuje določena karta v igralčevi roki. Vsak predmet je prisoten zaradi grafičnega prikaza ali za vsebovanje določenih lastnosti

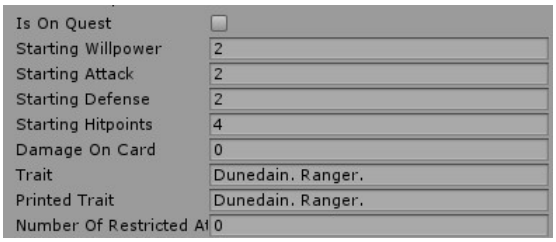
Slika 11 prikazuje podroben izgled igralčeve roke v nekemu trenutku. V tem primeru ima igralec trenutno v roki tri karte. Za primer je karta „Gleowine“ še bolj podrobno prikazana, razvidno pa je, da vsebuje veliko drugih predmetov, ki opisujejo njeno trenutno stanje. Vsebuje na primer predmet zadolžen za barvo karte (ki se lahko spreminja skozi čas), za slikico na karti, ceno karte, ... Ti predmeti skupaj tvorijo celotno karto in pomagajo hraniti določene informacije za nadzor nad trenutnim stanjem.

Algoritem 4 Primer lastnosti iz skripte zaveznika

Variables
public bool $IsOnQuest$ = False
public int $startingWillpower = 0$
private int $currentWillpower = 0$
public int $startingAttack = 0$
private int $currentAttack = 0$
public int $startingDefense= 0$
private int $currentDefense = 0$
public int $startingHitpoints = 0$
private int $currentMaxHitpoints = 0$
public int $damageOnCard = 0$
public string $trait$ = null
public string $printedTrait$ = null
public int $numberOfRestrictedAttachments = 0$
public int $currentPhase$ = PossiblePhases.ResourcePhase
end Variables

Znotraj predmetov so skripte, ki so napisane v kodi C#. V algoritmu 4 je primer lastnosti iz skripte zaveznika, ki pomagajo opisati njegovo trenutno stanje. Vidi se, da so nekatere spremenljivke javne, nekatere pa privatne. Dostop do javnih in njihovo spreminjanje je omogočeno vsem drugim predmetom, dostop do privatnih pa je dovoljen le znotraj predmeta. Za vsako lastnost zaveznika se hranita dve kopiji, in sicer trenutna ter začetna. To je pomembno zaradi tega, ker določeni efekti nastavijo trenutno stanje

na začetno, zato mora biti tudi drugo hranjeno. Ko pa se dogajajo spremembe v sistemu, se spreminja zgolj trenutno stanje.



Slika 12: Primer lastnosti iz predmeta zaveznika prikazuje primer, kako lahko nek predmet hrani določene lastnosti znotraj igre

Če je bila skripta pravilno napisana in prevedena, se vsaka javna lastnost iz prejšnjega primera prikaže tudi znotraj hierarhije predmeta. To omogoči, da se te lastnosti lahko spremlja in spreminja tudi ročno, torej znotraj igre in ne prek drugih skript. To je nadvse pomembno pri testiranju in iskanju napak. Slika 12 prikazuje pogled na javne lastnosti iz prejšnjega primera.

5.1.3 Kreacija karte

V tem podrazdelku bo predstavljen postopek kreacije karte znotraj igre. Karte se neprestano kreirajo, saj se mora ta vsakič, ko mora igralec iz kupčka potegniti karto (oz. ko se karta prikaže z vrha nasprotnikovega kupčka), kreirati z določenimi parametri. Ti so odvisni od karte, ki jo predstavljajo, včasih pa tudi od trenutnega stanja igre.

Algoritem 5 Realizacija igralčevega kupčka in vlečenja karte

```
Variables
    public List(Int) PlayersDeck = new List(Int)
end Variables
procedure DrawACard
    if PlayersDeck.Count > 0 then
        int rez = deck[0]
        deck.RemoveAt(0)
        MakeCard(rez, hand)
    end if
end procedure
```

Kot je prikazano v algoritmu 5, je igralčev kupček znotraj igre predstavljen kot seznam števil. Vsako število znotraj tega seznama je zbirateljska številka posamezne karte, ki enolično določa neko karto. Torej, ko mora igralec povleči karto, najprej preveri sistem, če je v kupčku vsaj še ena karta in če je ni, konča postopek. V nasprotnem primeru si najprej program zapomni karto na prvem mestu, nato iz seznama odstrani prvo karto in pokliče funkcijo za kreacijo karte.

Algoritem 6 Postopek kreiranja karte

```
procedure MakeCard(int cardNumber, GameObject destination)
    string colectorsNumber = cardNumber.ToString()
    string attack = dataBase.Find_Id (colectorsNumber).Attack
    string cost = dataBase.Find_Id (colectorsNumber).Cost
    string willpower = dataBase.Find_Id (colectorsNumber).Willpower
    string defence = dataBase.Find_Id (colectorsNumber).Defense
    string healthPoints = dataBase.Find_Id (colectorsNumber).HealthPoints
    string name = dataBase.Find_Id (colectorsNumber).Name
    string traits = dataBase.Find_Id (colectorsNumber).Traits
    string text = dataBase.Find_Id (colectorsNumber).Text
    string colour = dataBase.Find_Id (colectorsNumber).Color
    string typeCard = dataBase.Find_Id (colectorsNumber).Type
    string effect = dataBase.Find_Id(colectorsNumber).Effect
    string target = dataBase.Find_Id(colectorsNumber).PossibleTarget
    string isUnique = dataBase.Find_Id(colectorsNumber).IsUnique
    if TypeCard == "Ally" then
        GameObject card = Instantiate (allyPrefab, destination)
        card.MakeCard (name, attack, defence, cost, colectorsNumber, willpower,
          healthPoints, traits, text, colour, target, isUnique)
        card.transform.Find("RawImage").texture = Resources.Load(
          collectorNumber)
        InsertEffectsIntoCard(card, effect)
        card.informationsOfCard = informationsOfCard
        return card
    end if
end procedure
```

Funkcija „MakeCard“, ki je prikazana v algoritmu 6, sprejme dva parametra, in sicer zbirateljsko številko karte, ki mora bit kreirana, in destinacijo, kjer se more ta karta na koncu pojaviti (na primer igralčeva roka, pokopališče, bojišče itd). Ko funkcija preneha z izvajanjem, vrne referenco na predmet, ki ga je kreirala, oz. referenco na null, če je med postopkom kreacije prišlo do napake. Na sliki je delček kode, ki skrbi za kreacijo kart zaveznikov.

Funkcija najprej vzpostavi povezavo z bazo kart, od kod bo vse te podatke prebrala (več o zgradbi baze in delovanju v naslednjem razdelku). Ko je to storjeno, shrani referenco na bazo v lokalni spremenljivki, ki je pripravljena za nadaljnjo uporabo. Naslednjih nekaj vrstic kode služi za prebiranje vseh potrebnih podatkov iz baze in jih shranjuje v lokalne spremenljivke, saj bodo bile vrednosti v nadaljevanju posredovane drugim funkcijam za njihovo obdelavo.

Sledi koda, ki je namenjena izključno kreaciji kart zaveznikov. Tip karte je bil že nekaj vrstic nazaj prebran iz baze, sedaj pa se pogleda, ali je tip karte zaveznik in če je, se sproži naslednji del kode. Prva vrstica kreira čisto nov predmet v sceni, ki ga postavi v hierarhijo. Ta predmet je sprva prazen in mora biti še napolnjen s podatki. Naslednji dve vrstici posredujeta vse podatke o karti novi skripti, ki služi

za shranjevanje podatkov zavezniških kart, oz. njeni metodi „MakeCard“. Ta metoda tudi izriše vse podatke o karti na karto, da jih lahko uporabnik stalno spremlja. Četrta vrstica znotraj tega „if“ stavka naloži sliko karte kot teksturo in to prikaže na karti. Naslednja vrstica poskrbi še za vse ostale efekte, ki so odvisni od trenutnega stanja igre, in jih shrani v karto. Nato se v karto shrani še referenca na informacije o karti, ki služi za hitrejši dostop do informacij, na koncu pa funkcija vrne kar celotno karto, oz. referenco na karto, če bo klicatelj te funkcije to slučajno potreboval.

5.2 Izvedba podsistema za obdelavo zahtev in spremembe prikaza

V tem razdelku bo prikazanih nekaj primerov iz kode, ki obdelujejo zahteve in/ali spreminjajo prikaz igre. Celoten podsistem je tudi v tem primeru decentraliziran, saj nekako vsak predmet poskrbi le za zahteve, ki se nanašajo nanj. Prednost tega je, da se da (če se pojavi napaka pri neki zahtevi) zelo enostavno poiskati izvir napake, saj je celotna koda za ta predmet zbrana na določenem mestu. Tudi za spremembo prikaza skrbi vsak predmet zase, s pomočjo tehnike enkapsulacije. Ta tehnika bo podrobneje predstavljena v nadaljevanju, in sicer ob podanih primerih iz kode.

5.2.1 Nadzor pripravljenosti lika

Vsak lik (zaveznik ali heroj) je lahko v nekem danem trenutku na bojišču ali pripravljen ali utrujen (ang. Ready/Exhausted). V pravilniku za igranje je napisano, da se to predstavi z zasukom karte za 90 stopinj v desno. To pomeni, da je dana karta lahko v nekem trenutku postavljena ali vertikalno (je pripravljena) ali horizontalno (je utrujena). V tem razdelku bo predstavljen primer kode, ki skrbi za pravilno postavitev karte.

(a) Vertikalna postavitev pripravljenega heroja

(b) Horizontalna postavitev utrujenega heroja

Slika 13: Slika horizontalne in vertikalne pozicije karte na bojišču, glede na njeno pripravljenost. Prikazuje razliko v grafičnem prikazu, za lažji pregled situacije na bojišču

Na sliki 13 je prikazano, kako izgleda karta po tem, ko se zasuka v desno za 90 stopinj. S tem obratom omogočamo veliko boljšo preglednost stanja na bojišču. V nadaljevanju je predstavljeno, kako je to rešeno programsko. Rešitev mora vsekakor

biti znotraj skripte, ki upravlja z vedenjem karte, da lahko poljubna druga funkcija sproži to spremembo v karti.

Algoritem 7 Funkcija za utruditev lika

```
procedure MakeCard(int cardNumber, GameObject destination)
    if isExhausted == false then
        transform.Rotate (0, 0, 270)
        for all Transform transformTmp in this.gameObject.transform do
            if transformTmp.GetComponent(AttachmentScript) != null then
                transformTmp.Rotate(0, 0, 90)
                Vector3 oldVector = transformTmp.localPosition
                float oldXComponent = transformTmp.localPosition.x
                float oldYComponent = transformTmp.localPosition.y
                float oldZComponent = transformTmp.localPosition.z
                Vector3 newVector = new Vector3(-oldXComponent, oldYComponent,
                  oldZComponent)
                transformTmp.localPosition = newVector
            end if
        end for
    end if
end procedure
```

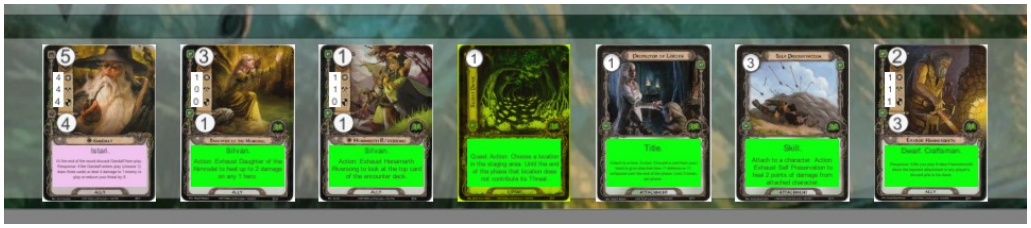
Funkcija v algoritmu 7 prikazuje kodo znotraj karte, ki služi utrujanju lika. Funkcija najprej preveri, ali je lik že utrujen in če je, ne stori nič. V primeru, da je pripravljen in ga mora utruditi, se zgodi naslednja koda. Najprej zasuka celotni predmet karte v levo za 270 stopinj, kar naredi efekt obračanja v desno za 90 stopinj (funkcija „Rotate“ je definirana za levo stran), nato pa nastavi vrednost utrujenosti na pozitivno.

Problem nastane, ko ima karta na sebi tudi kak pripomoček, recimo meč ali ščit. S tem, ko se karta obrne v desno za 90 stopinj, morajo vsi pripomočki obdržati svojo pozicijo. Ampak funkcija „Rotate“ zavrti celotno karto skupaj z vsemi pripomočki na njej. Zato je potrebno vse pripomočke obrniti spet na prejšnjo pozicijo (ker se tudi pripomočki „utrudijo“ neodvisno od karte, morajo obdržati njihovo „utrujenost“). Funkcija „Rotate“ za nasprotni kot uredi vse potrebno za pravilno rotacijo pripomočkov, vendar se spremeni tudi pozicija pripomočkov v odvisnosti od karte, zato je potrebno vse pripomočke geometrijsko gledano še preslikati čez abscisno os. To se stori tako, da se ustvari nov tridimenzionalni vektor, ki je kopija prejšnjega vektorja pozicije, le da ima nasprotno vrednost na x-osi. Ko je ta vektor ustvarjen, se z njegovo vrednostjo prepiše prejšnji vektor pozicije in s tem uredi pravilno orientacijo karte. Funkcija za pripravo heroja pa deluje podobno, vendar v nasprotni smeri.

5.2.2 Upravljanje sijaja kart

Karte, ki so trenutno v igralčevi roki, so lahko v enem izmed dveh stanj. Igralec ima lahko možnost igrati določeno karto v natanko tem trenutku, lahko pa se zgodi da te možnosti nima. Prikaz možnosti igranja karte je zelo priročen, saj igralcu pokaže, katere možnosti ima trenutno na voljo, s tem pa mu olajša odločitve. Znotraj igre torej

želimo, da vsaka karta zasije, ko jo ima igralec možnost igrati, ter da izgubi sijaj, ko igralec izgubi možnost igranja te karte. V tem razdelku bo predstavljena realizacija te zahteve.



(a) Primer nesijočih kart

(b) Primer sijočih kart

Slika 14: Slika sijočih in nesijočih kart v igralčevi roki, za prikaz razlike med kartami, ki jih igralec lahko trenutno uporabi in tistimi, ki jih trenutno ne more uporabiti

Na sliki 14 je prikazana razlika v izgledu sijočih in nesijočih kart v igralčevi roki. Na sliki (a) je četrta karta z leve pobarvana, saj je „dogodek“ in se jo lahko igra skoraj kadarkoli, zato je vedno sijoča. Ostale karte zasijejo le v fazi planiranja, saj jih lahko igralec igra iz roke le takrat. Na sliki (b) pa se igralec nahaja znotraj faze planiranja in zaradi tega lahko igra poljubno karto iz svoje roke.

Algoritem 8 Funkcija za vstop v fazo planiranja

```
if currentPhase == PossiblePhases.PlanningPhase then
    infoWindow.transform.Find("Text").text = "Planning phase. You may play
      allies/attachments from your hand"
    phaseLegend.ChangeTurnInLegend(1)
    hand.MakeAllyAttachemntGlow(true)
    hand.MakeEventsGlow(true)
    checkForEndPhaseTriggers()
end if
```

V algoritmu 8 je prikazan delček kode iz že prej omenjene funkcije „ChangeTurn“, ki spremeni trenutno fazo, v kateri se igra nahaja. Delček kode s slike predstavlja tisti del kode, ki spremeni igro v fazo planiranja. Najprej se spremeni besedilo, ki je prikazano uporabniku, da ta ve, v kateri fazi se trenutno nahaja. Med drugim se tudi pokličeta funkciji za narediti vse karte v igralčevi roki sijoče. Metodo za to opravilo nosi predmet, ki predstavlja uporabnikovo roko, saj se vse relevantne karte nahajajo tam.

Algoritem 9 Funkcija za osvetlitev karte v igralčevi roki

```
procedure MakeAllyAttachmentGlow(bool glow)
    if glow == true then
        for all Transform transformTmp in transform do
            if transformTMP.GetComponent(AllyScript) != null OR
                transformTMP.GetComponent(AttachmentScript) != null then
                transformTMP.Find("RawImage").material = glowingEmission
            end if
        end for
    else
        for all Transform transformTmp in transform do
            if transformTMP.GetComponent(AllyScript) != null OR
                transformTMP.GetComponent(AttachmentScript) != null then
                transformTMP.Find("RawImage").material = null
            end if
        end for
    end if
end procedure
```

Predmet, ki predstavlja igralčevo roko, vsebuje funkcijo opisano v algoritmu 9. Ta funkcija je precej enostavna, saj vsebuje le en parameter, in sicer vrednost, ali je treba karte obsijati ali ne. V primeru, da je ta vrednost negativna, se bodo karte prenehale svetiti. Funkcija takoj pride do vejitve, in če gre v prvo vejitev, preveri vse karte v igralčevi roki, ter če je katera od njih zaveznik ali pripomoček, se prične svetiti. Treba je vedeti, da predmet igralčeve roke vsebuje veliko število otrok v svoji hierarhiji, zato je vedno treba preveriti, če je otrok sploh karta v roki, ali če je element za kaj drugega. To se najlažje preveri tako, da se pogleda, če vsebuje skripto enega izmed tipa kart (v tem primeru zaveznika ali pripomočka). Druga vejitev naredi podobno, le da kartam odvzame element, ki povzroča njihov sijaj.

5.2.3 Enkapsulacija lastnosti

Zelo močna programerska tehnika za preprečevanje napak in povečanja preglednosti kode je enkapsulacija lastnosti znotraj nekega razreda. To pomeni, da določenim lastnostim dodelimo privatni status in s tem preprečimo, da bi lahko drugi deli programa dostopali do vrednosti in jih spreminjali. Namesto normalnega dostopa pa omogočimo dostop do lastnosti prek posebej definiranih metod v razredu. Te metode previdno spremljajo, kakšen vhod dobijo, in preverijo možnosti napak, preden naredijo popravke v lastnostih. Prav tako lahko v njih naredimo še dodatne popravke, ki se morajo izvesti ob vsaki spremembi lastnosti.

Algoritem 10 Primer enkapsulacije trenutnega napada

```
procedure CurrentAttack
    if Get then
        return currentAttack
    end if
    if Set then
        actualAttack += value
        if actualAttack < 0 then
            currentAttack = 0
        else
            currentAttack = actualAttack
        end if
    end if
    this.transform.Find("Stats").transform.Find("Attack").text =
      currentAttack.ToString()
end procedure
```

V algoritmu 10 je prikazan primer enkapsulacije trenutnega napada nekega lika. Enkapsulacija je vedno sestavljena iz dveh delov, in sicer iz „Get“ in „Set“ metod. „Get“ metoda določa, kaj se zgodi, ko hoče neka funkcija izven tega razreda dostopati do dane lastnosti z namenom ogleda vrednosti lastnosti. Ker pri trenutnem napadu ne more iti nič narobe, ga samo pokažemo. V primeru, ko pa hoče funkcija popraviti lastnost trenutnega napada, pa se sproži metoda „Set“. V tem primeru je potrebno pogledati, če je funkcija želela spremeniti napad pod vrednost 0. Posebej za ta namen se hrani tudi lastnost „actualAttack“, ki predstavlja dejanski trenutni napad. Ta je lahko negativen, in če je, se pokaže na karti vrednost, kot da bi bil napad enak 0 (znotraj igre se ne sme zgoditi, da bi bil napad na karti manjši od 0). Vrednost dejanskega trenutnega napada se torej hrani zgolj za izračun, če gre napad enkrat pod 0, kasneje pa se spet poveča. Pomembno pa je tudi, da se ob vsakem popravku napada zapiše tudi nova vrednost v predmet, ki predstavlja karto. Za to pa poskrbi zadnja vrstica kode.

Algoritem 11 Primer enkapsulacije prejemanja škode

```
damageOnCard += damage
procedure TakeDamage(int amountOfDamageTaken)
    if damageOnCard ≥ currentMaxHitpoints AND isDead == false then
        isDead = true
        for all Transform transformTMP in transform do
            if transformTMP.GetComponent(AttachmentScript) != null then
                PlayerHand.CardDiscardedFromPlay(transformTMP.gameObject)
                DiscardPileImage.addCardToDiscPile( transformTMP.collNum)
            end if
        end for
        DiscardPileImage.addCardToDiscPile(GetComponent(CardScript).collNum)
        PlayerHand.CardDiscardedFromPlay(this.gameObject)
        Destroy(this.gameObject)
    else if damage > 0 then
        CheckSufferDamageResponse(damage)
        DamageCounter.transform.Find("Text").text = damageOnCard.ToString()
        DamageCounter.GetComponent(Image).enabled = true
        DamageCounter.transform.Find("Text").enabled = true
    end if
end procedure
```

Bolj zahteven primer enkapsulacije je prikazan v algoritmu 11. Tukaj se je celotni tok iz „Set“ metode za popravljanje števila življenjskih točk lika preusmeril v dano funkcijo za prejemanje škode. Kot parameter v tej funkciji dobimo število škode, ki je bila prejeta, in na karti povečamo število škode za toliko. Preveriti moremo, če ima karta na sebi več škode, kot jo lahko prejme, in če lik ni še umrl, umre sedaj. Iz njega se morejo odstraniti vsi pripomočki, ki so nato dodani v igralčevo pokopališče. Ko so vsi pripomočki odstranjeni, dodamo še ta lik v pokopališče (natančneje njegovo kopijo dodamo v pokopališče, njegov predmet na bojišču pa uničimo). Če pa lik ni umrl, ko je prejel dodatno škodo, pa le sprožimo vse efekte, ki se zgodijo ob prejetju škode, ter na njegovem predmetu popravimo število prikazane škode, da je prikaz skladen s trenutnim dogajanjem.

5.3 Izvedba podsistema za iskanje podatkov o karti

Za prikaz delovanja podsistema za iskanje podatkov o karti je potrebno najprej razumeti, kako je sestavljena baza, v kateri se podatki o karti nahajajo. V temu razdelku bo najprej prikazano, na kakšnemu principu je zgrajena celotna podatkovna baza s pomočjo programa Excel, sledil bo še prikaz podatkovne baze v končni izvedbi. Za tem bosta sledila prikaz poizvedb v bazo z namenom iskanja podatkov o karti in interpretacija tako pridobljenega rezultata.

5.3.1 Opis podatkovne baze z informacijami o kartah

Podatkovna baza z informacijami o karti je v tem sistemu predstavljena kot ena velika tabela s podatki o vseh kartah. To je narejeno z namenom lokalizacije podatkov in zmanjševanjem kompleksnosti pri iskanju po bazi. Vseeno je končna tabela v tretji normalizacijski obliki, da se lahko izogne morebitnim anomalijam pri branju in posodabljanju podatkov znotraj baze. Da je tabela v tretji normalizacijski obliki, mora zadoščati naslednjim pogojem:

1. V vsakem stolpcu mora biti vrednost natanko enega, točno določenega tipa.
2. Na križišču vsakega stolpca in vsake vrstice mora biti natanko ena vrednost.
3. Tabela mora vsebovati identifikator, ki enolično označuje vsako vrstico.
4. Vse lastnosti morajo biti enolično določene s tem identifikatorjem.
5. Med nobenimi tremi lastnostmi ne sme veljati tranzitivna relacija.

S pomočjo teh pravil se olajša iskanje v bazi, z njimi odstranimo morebitne redundantnosti iz baze. Za identifikator je bila izbrana kar zbirateljska številka karte, saj je enolična za vsako karto. Od nje so nato odvisne vse ostale lastnosti, saj jih zbirateljska številka enolično določa (ne moreta dve karti imeti enake zbirateljske številke).

Tabela 1: Poenostavljen primer podatkovne baze

Id	Cost	Attack	Willpower	Defense	HealthPoints	Name	Traits
1	2	3	3	1	10	Gimli	Dwarf
2	4	5	4	2	8	Aragorn	Human
3	6	3	2	3	7	Silverlode Archer	Silvan
4	2	0	0	0	0	Steward of Gondor	Noble

Tabela 1 prikazuje poenostavljen primer tabele za iskanje informacij o karti. Ko bo hotela neka funkcija dostopati do podatkov o neki karti, bo dovolj, da ve samo zbirateljsko številko te karte (oz. njen identifikator), nato pa bo lahko prebrala celotno vrstico in imela vse podatke o zahtevani karti. Vse lastnosti so ločene po stolpcih, da je vstavljanje novih kart enostavno, tipi lastnosti po stolpcih pa so vnaprej definirani. Poleg zgornjih lastnosti, mora uporabljena tabela vsebovati še tekst vsake karte, njene efekte, tip karte itd.

Id	Cost	Attack	Willpower	Defense	HealthPoints	Name	Traits	Text	Color	Type	Effect	PossibleT	IsUnique
1	12	3	2	2	5	Aragorn	Dunedain. Noble. Ranger.	Sentinel.Response: Afte	Purple	Hero	1.QuestResponse.Pay	None	TRUE
2	8	2	1	1	4	Theodred	Noble. Rohan. Warrior.	Response: After Theodre	Purple	Hero	1.QuestResponse.Nor	None	TRUE
3	9	2	2	1	4	Gloin	Dwarf. Noble.	Response: After Gloin su	Purple	Hero	1.OnSufferDamageRe	None	TRUE
4	11	2	2	2	5	Gimli	Dwarf. Noble. Warrior.	Gimli gets +1 Attack for e	Red	Hero	1.OnSufferDamageRe	None	TRUE
5	9	3	1	1	4	Legolas	Noble. Silvan. Warrior.	Ranged. Response: After	Red	Hero	1.OnKilledEnemy.Nor	None	TRUE
6	9	2	1	2	4	Thalin	Dwarf. Warrior.	While Thalin is committe	Red	Hero	1.WhenRevealedResp	None	TRUE
7	9	1	4	1	3	Eowyn	Noble. Rohan.	Action: Discard 1 card fro	Blue	Hero	1.Action.DiscardACarc	Player	TRUE
8	7	1	1	2	3	Eleanor	Gondor. Noble.	Response: Exhaust Elean	Blue	Hero	1.WhenRevealedResp	None	TRUE

Slika 15: Primer iz podatkovne baze prikazuje prvih nekaj vrstic iz končne verzije podatkovne baze. Pogled je iz programa Microsoft Excel 2016

Podatkovna baza je v končni različici v formatu CSV, na sliki 15 pa je prikazana znotraj programa Microsoft Excel za lažjo predstavo. CSV format pomeni, da so vse

vrstice stisnjene skupaj in ločene z vejico. Ker stolpca „Text“ in „Effect“ (na sliki označena s sivo barvo) vsebujeta zelo dolge lastnosti, sta na sliki skrčena. Razvidno je, da so podatki v približno enaki obliki kot v tabeli 1. Vsi ti podatki morajo biti prebrani iz baze, nato pa vstavljeni v kreirano karto, tako da jih lahko igralec razbere iz grafičnega prikaza in tudi, da se podatki shranijo znotraj programa. Atribut „Effect“ pa mora biti prepoznan s posebno funkcijo, ki karti priredi potrebni efekt. Zaradi ponavljanja efektov z različnimi parametri, a drugačnim tekstom, je ta stolpec izpolnjen ročno, ločeno od lastnosti „Text“, iz katerega bi sicer lahko razbrali efekt karte.

5.3.2 Opis poteka poizvedb v podatkovno bazo

Kot je bilo omenjeno že v razdelku za kreacijo karte, imajo poizvedbe za podatkovno bazo dva parametra, in sicer: zbirateljska številka karte in lastnost, ki je zahtevana. V nadaljevanju bo prikazano, kako je sestavljen razred, ki skrbi za neposredni dostop do podatkovne baze, in kako upravlja s poizvedbami podanega tipa.

Algoritem 12 Razred za upravljanje s podatkovno bazo

```
Variables
    public TextAsset File
end Variables
procedure START
    Load(File)
end procedure
PUBLIC CLASS ROW:
Variables
    public string Id
    public string Cost
    public string Attack
    public string Willpower
    public string Defense
    public string HealthPoints
    public string Name
    public string Traits
    public string Text
    public string Color
    public string Type
    public string Effect
    public string PossibleTarget
    public string IsUnique
end Variables
```

V algoritmu 12 je prikazan razred za upravljanje s podatkovno bazo. Razred najprej potrebuje datoteko, do katere bomo sploh dostopali, referenco nanjo pa nastavimo kar v uporabniškem vmesniku Unity. Ob zagonu celotnega programa pa mora biti datoteka naložena v spomin računalnika, za kar poskrbi funkcija „Load(File)“. Za upravljanje s podatkovno bazo je zelo priročno tudi, če imamo razred „Row“, ki predstavlja dano vrstico v bazi. Ko preberemo vrednosti neke vrstice, samo shranimo posamezne vrednosti

iz stolpcev v njiho lastnost, da jih lahko nato poljubna druga funkcija prebere.

Algoritem 13 Prebiranje celotne podatkovne baze

```
Variables
    List(row) rowList = new List(row)
end Variables
procedure Load(TextAsset csv)
    rowList.Clear()
    string[][] grid = CsvParser2.Parse(csv.text)
    for int i = 1; i < grid.Length; i++ do
        Row row = new Row()
        row.Id = grid[i][0]
        row.Cost = grid[i][1]
        row.Attack = grid[i][2]
        row.Willpower = grid[i][3]
        row.Defense = grid[i][4]
        row.HealthPoints = grid[i][5]
        row.Name = grid[i][6]
        row.Traits = grid[i][7]
        row.Text = grid[i][8]
        row.Color = grid[i][9]
        row.Type = grid[i][10]
        row.Effect = grid[i][11]
        row.PossibleTarget = grid[i][12]
        row.IsUnique = grid[i][13]
        rowList.Add(row)
    end for
    isLoaded = true
end procedure
```

Prebiranje iz podatkovne baze je predstavljeno v algoritmu 13. Najprej se ustvari seznam tipa vrstice, kjer bodo shranjene vse vrstice iz baze, da ne bo potrebno vsakič znova brati podatke iz baze, temveč se bo do njih dostopalo kar preko tega seznama. Funkcija „Load“, ki je klicana ob zagonu programa, prebere celotno podatkovno bazo in shrani vse vrstice lokalno. Najprej sprazni prejšnji seznam, če še ni prazen, nato pa ustvari matriko iz celotnega teksta CSV datoteke. Sledi sprehod po celotni matriki, vrstica za vrstico. Znotraj vsake vrstice matrike ustvarimo nov predmet tipa vrstica, vanj pa shranimo vse prebrane vrednosti. Ta predmet nato shranimo v seznam vrstic, ki je bil predhodno ustvarjen. Ko je to končano, še potrdimo, da je celoten postopek prebiranja končan.

Algoritem 14 Iskanje po identifikatorju v bazi

```
procedure Row_Id(string find)
    return rowList.Find(x => x.Id == find)
end procedure
```

Ko imamo narejen seznam vrstic celotne baze, je postopek iskanja po njej trivi-

alen. Koda v algoritmu 14 poišče dano vrstico znotraj seznama vrstic po id-ju in jo vrne. Podobno bi lahko definirali tudi iskanje po poljubni drugi lastnosti. Če pa hočemo dostopati do specifične lastnosti znotraj neke vrstice, uporabimo izraz: „Find_ID(ZbirateljskaStevilka).Attack“ za primer iskanja napada določene karte.

Predhodno prebiranje baze v pomnilnik je uspešno zaradi tega, ker je znano, da se podatkovna baza skozi potek programa ne bo spreminjala. Če bi bili možni tudi popravki med potekom, bi bilo potrebno začasno zapreti možnost dostopa do podatkovne baze, jo nato posodobiti in na koncu spet prebrati celotno bazo v pomnilnik. Pri pogostih popravkih bi bilo to delo zamudno in bi lahko bila boljša možnost kar sprotno prebiranje iz baze.

6 Zaključek in nadaljnje delo

Končni rezultat projekta je bil popolnoma delujoč program, ki omogoča na poljubni platformi igranje namizne igre Gospodarja prstanov. Sledilo je tudi obširno testiranje in preučevanje različnih interakcij med kartami, ki bi lahko ogrozile želen potek igre. Ker se tudi sama igra spreminja skozi čas ter vedno znova prihajajo novi scenariji in karte za igralce, je potrebno programsko opremo stalno vzdrževati in dodajati nove karte. To lahko pripelje do potrebe po popravkih v že delujočem delu programa, zato ne bo program nikoli čisto končan, vsaj dokler bo igra še živa.

V prihodnje bi se lahko igro razširilo za več igralcev, da bi se lahko povezali na nek strežnik in v igri sodelovali. Potrebna razširitev je tudi kolekcija kart, ki jih igralec lahko uporablja in tiste, ki jih mora še nekako odkleniti (sedaj ima vsak igralec na voljo čisto vse obstoječe karte). Prav tako bi lahko bila realizirana tudi optimizacija za ostale naprave, ki niso računalnik. Igro je seveda možno na njih igrati, vendar je razporeditev elementov v sceni prilagojena za računalnik. V primeru manjšega ekrana, kot pa ga ima na primer telefon, bi bilo potrebno elemente drugače razporediti.

Literatura in viri

[1] L. Bishop, D. Eberly, T. Whitted, M. Finch, and M. Shantz. Designing a pc game engine. *IEEE Comput. Graph. Appl.*, 18(1):46–53, Jan. 1998.

[2] A. Dennis. *Systems Analysis and Design*. Wiley Publishing, 5th edition, 2012.

[3] B. Dobing and J. Parsons. How uml is used. *Commun. ACM*, 49(5):109–113, May 2006.

[4] J. Gregory. *Game Engine Architecture*. A. K. Peters, Ltd., USA, 2nd edition, 2014.

[5] C. Hofmeister, R. L. Nord, and D. Soni. Describing software architecture with uml. In *Proceedings of the First Working IFIP Conference on Software Architecture*, pages 145–160. Kluwer Academic Publishers, 1999.

[6] C. Larman and V. R. Basili. Iterative and incremental developments. a brief history. *Computer*, 36(6):47–56, June 2003.

[7] S. Mcconnell. *Code Complete, 2nd Edition*. Wiley India Pvt. Limited, 2004.

[8] S. Nakov. *Fundamentals of Computer Programming with C#: The Bulgarian C# Book*. Free books in computer programming, software development and C#. Svetlin Nakov, 2013.

[9] P. Paul, S. Goon, and A. Bhattacharya. History and comparative study of modern game engines. May 2012.

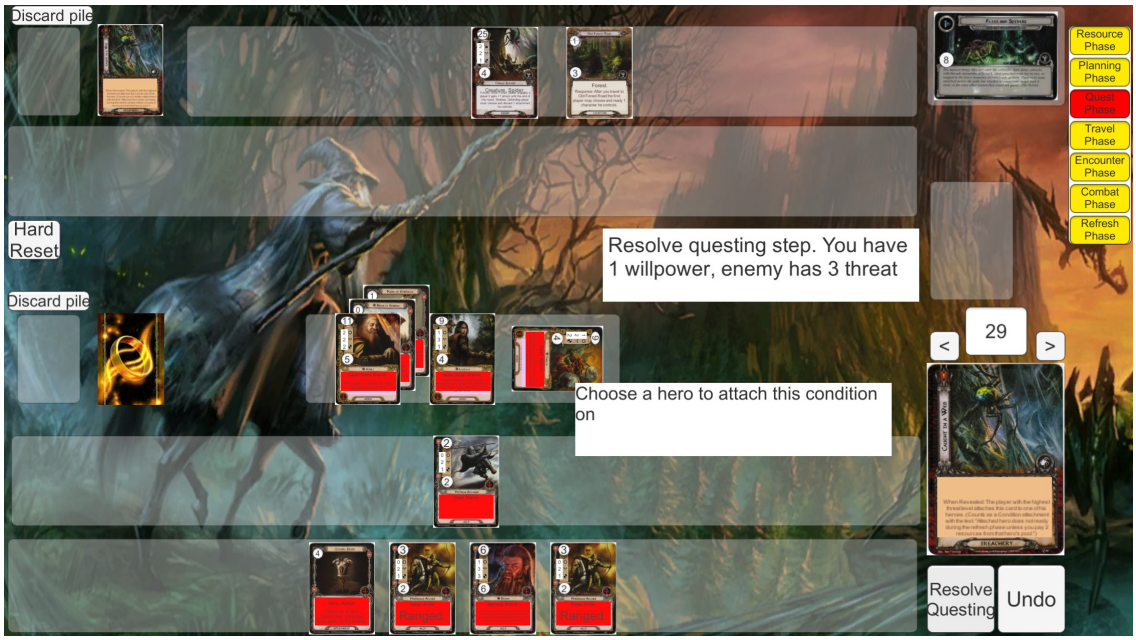
[10] J. Sonmez. *The Complete Software Developer's Career Guide: How to Learn Your Next Programming Language, Ace Your Programming Interview, and Land the Coding Job of Your Dreams*. Simple Programmer, LLC, 2017.

[11] M. Wolf. *The Video Game Explosion: A History from PONG to Playstation and Beyond*. Greenwood Press, 2008.

Priloge

A Primer položaja z modrim kupčkom

B Primer položaja z rdečim kupčkom



C Primer položaja z vijoličnim kupčkom

D Primer kode za borbo med zaveznikom in nasprotnikom

```
procedure ResolveCharacterAttack
    int currentAttack = (attackingCharacter.GetComponent(CharacterScript).
    CurrentAttack + CheckForExtraDamage(attackingCharacter, defendingEnemy))
    for all GameObject attacker in additionalAttackers do
        currentAttack += (attacker.GetComponent()CharacterScript).CurrentAttack
        + CheckForExtraDamage(attacker, defendingEnemy))
    end for
    int enemyDefence = defendingEnemy.GetComponentEnemyScript).currentDefense
    if currentAttack > enemyDefence then
        int currDmg = defendingEnemy.GetComponent(EnemyScript).damageOnCard
        int currMaxHP = defendingEnemy.GetComponent¡EnemyScript¿().currentMaxHP
        if currDmg + (currentAttack − enemyDefence) ≥ currMaxHP then
            CheckForKilledEnemyResponse()
        end if
        defendingEnemy.GetComponent(EnemyScript).takeDamage (
        currentAttack - enemyDefence)
    end if
    additionalAttackers = new List(GameObject)
    waitingForAdditionalAttackers = false
    isAttackResolving = false
    attackingCharacter = null
    defendingEnemy = null
    numberOfAttackingCharacters = 0
    firstPlayer.Find("Button").GetComponent(Image).color = Color.yellow
    firstPlayer.Find("Button").Find("Text").text = Štop attacking"
end procedure
```

E Primer kode za uspešnost misije

```
procedure ResolveQuesting
    int totalWillpower = CalculateTotalWillpower()
    int totalThreat = CalculateTotalThreat()
    if (totalWillpower − totalThreat) ≥ 0 then
        int difference = totalWillpower - totalThreat
        LocationScript ActiveLocation = GameObject.Find("ActiveLocation").
        GetComponent(LocationScript)
        if ActiveLocation! = null then
            int remaining = ActiveLocation.currentQuestPoints -
            ActiveLocation.LocationProgress
            if difference > remaining then
                ActiveLocation.LocationProgress = remaining
                currentScenario.TokensOnQuest = difference - remaining
                lastProgressOnLocation = remaining
                lastProgressOnQuest = difference - remaining
                lastThreatRise = 0
            else
                ActiveLocation.LocationProgress = difference
                lastProgressOnLocation = difference
                lastProgressOnQuest = 0
                lastThreatRise = 0
            end if
        else
            currentScenario.TokensOnQuest = difference
            lastProgressOnLocation = 0
            lastProgressOnQuest = difference
            lastThreatRise = 0
        end if
    else
        threatCounter.CurrentThreat += (totalThreat - totalWillpower)
        lastProgressOnLocation = 0
        lastProgressOnQuest = 0
        lastThreatRise = totalThreat - totalWillpower
    end if
end procedure
```