PASIJANS

Markovica, Tomislav

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIK, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Preddiplomski sveučilišni studij

PASIJANS

Završni rad

Tomislav Markovica

Osijek, 2021.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac Z1P - Obrazac za ocjenu završnog rada na preddiplomskom sveučilišnom studiju

Osijek, 17.09.2021.

Odboru za završne i diplomske ispite

Prijedlog ocjene završnog rada na preddiplomskom sveučilišnom studiju

Ime i prezime studenta:	Tomislav Markovica
Studij, smjer:	Preddiplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	R4095, 28.07.2017.
OIB studenta:	45676577963
Mentor:	izv. prof. dr.sc. Josip Job
Sumentor:	
Sumentor iz tvrtke:	
Naslov završnog rada:	Pasijans
Znanstvena grana rada:	**Informacijski sustavi (zn. polje računarstvo)**
Predložena ocjena završnog rada:	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 2 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 2 bod/boda Jasnoća pismenog izražavanja: 2 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene mentora:	17.09.2021.
Datum potvrde ocjene Odbora:	

Potpis mentora za predaju konačne verzije rada u Studentsku službu pri završetku studija:	Potpis: Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

IZJAVA O ORIGINALNOSTI RADA

Osijek, 20.09.2021.

Ime i prezime studenta:	Tomislav Markovica
Studij:	Preddiplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	R4095, 28.07.2017.
Turnitin podudaranje [%]:	2

Ovom izjavom izjavljujem da je rad pod nazivom**: Pasijans**

izrađen pod vodstvom mentora izv. prof. dr.sc. Josip Job

i sumentora

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

SADRŽAJ

1. UVOD 1

1.1. Zadatak završnog rada 2

2. PREGLED PODRUČJA TEME 3

2.1. Klondike pravila igre 3

2.2. Deterministički problem pasijansa 3

2.2.1. Računska teorija kompleknosti 4

2.2.2. Strategije računala 6

2.2.3. Zašto je klondike NP problem 8

2.3. Implementacije za različite platforme 10

3. APLIKACIJA ZA IGRANJE KARTAŠKE IGRE PASIJANS 12

3.1. Arhitektura Unity *Engine*-a 12

3.1.1. Komponente 12

3.1.2. API za izradu skripti 14

3.2. Implementacija igre 15

3.2.1. Dizajn aplikacije 15

3.2.2. Preddefinirani objekti 16

3.2.3. Kreiranje objekata scene 19

3.2.4. Kreiranje i dodjela karata 21

3.2.5. Metoda povlačenja i spuštanja 22

3.2.6. Primanje spuštene karte 24

3.2.7. Hijerarhija slaganja 25

4. ZAKLJUČAK 27

LITERATURA 28

POPIS SLIKA 29

POPIS TABLICA 30

SAŽETAK 31

ABSTRACT 32

ŽIVOTOPIS 33

1. UVOD

Pasijans (engl. s*olitaire*) je žanr kartaških igara za jednog igrača. Porijeklo igre je nepoznato, ali se vjeruje da je igra njemačkog ili skandinavskog porijekla. Cilj rada je izvedba igre u Unity Game Engine-u u C# programskom jeziku. Točnije, razvoj najpopularnije i najčešće pojavljivane verzije pasijansa, klondike verzije.

Pasijans je postao fenomen od trenutka kada je prvi put došao u paketu s Windows 3.0 operacijskim sustavom. Od tada sve inačice Windows-a imaju *Solitaire*. Ideja da se u paket operacijskog sustava uklopi ta jednostavna kartaška igra došla je zbog želje razvojnog tima u Microsoftu da upoznaju korisnike njihovog operacijskog sustava s novim „Drag and drop“ konceptom, koji je tada uveden kao nova funkcionalnost grafičkog korisničkog sučelja. Svim današnjim korisnicima računala „povuci i spusti“ metoda je jedna od temeljnih funkcionalnosti koju svi intuitivno koriste, no korisnici tada uopće nisu imali predodžbu čemu to služi. Glavna ideja razvoja pasijans igre bila je upravo ta „Drag and drop“ metoda.

Pasijans je igra za koju se može reći da je već desetljećima bijeg od dosade ili čak stresa radnog mjesta [1]. Pravila igre su obično vrlo jednostavna za savladati tako da igrač vrlo lako može zaredati nekoliko pobjeda koje su same po sebi neka vrsta nagrade koja oslobađa dopamin. Osobi se tada ukaže potencijalni privid kako može postati dobra u nečemu, novoj vještini. Zanimljivo je kako tako jednostavna igra ispadne vrlo adiktivna. Nije nužno riječ o ozbiljnoj ovisnosti, ali igra može oduzeti znatnu količinu vremena. S druge strane igra može pružiti i pozitivne utjecaje na mentalno zdravlje, može poboljšati sposobnost fokusa osobe, opušta misli.

Drugo poglavlje objašnjava pravila za igranje i konačni cilj klondike pasijansa. Poglavlje govori o klasama kompleksnosti i kojoj klasi pripada klondike pasijans. Još uvijek nije uspješno dokazano koja je vjerojatnost rješivosti jedne igre niti koji je najbolji način igranja. Poglavlje se bavi pitanjima kakvi su potezi igre, deterministički ili probabilistički, te kakvi algoritmi su potrebni za pronalazak rješenja igre. Koji su načini pronalaženja rješenja za tako velike probleme i kako klase kompleksnosti teoretski opisuju vrijeme izračunavanja? Što znači da je problem NP klase i NP-potpun? Na kraju je navedena je opća procedura za rješavanje igre. Poglavlje će se ukratko dotaknuti i opisati najpopularniju verziju igre.

Treće poglavlje objašnjava uloge osnovnih građevnih blokova Unity projekta, a to su komponente i skripte. Ovo poglavlje je dokumentacija, objašnjava izvedbu ovog programa u

Unity-u. Poglavlje opisuje ciljeve, određene razloge strukturiranja klasa i korištenja određenih obrazaca oblikovanja. Poglavlje sadrži slike sa UML dijagramima koji prikazuju strukturu klasa.

1.1. Zadatak završnog rada

Potrebno je proučiti i opisati pravila kartaške igre pasijans (engl. s*olitaire*) te usporediti postojeće implementacije za različite računalne platforme. U okviru završnog rada potrebno je, primjenom odgovarajućih računalnih tehnologija, izraditi vlastitu implementaciju igre.

2. PREGLED PODRUČJA TEME

Poglavlje objašnjava pravila kartaške igre, zatim objašnjava klase kompleksnosi i navodi kojoj klasi kompleksnosti pripada igra. Poglavlje se bavi pitanjima kakvi su potezi igre, deterministički ili probabilistički, te kakvi algoritmi su potrebni za pronalazak rješenja igre. Koji su načini pronalaženja rješenja za tako velike probleme i kako klase kompleksnosti teoretski opisuju vrijeme izračunavanja? Što znači da je problem NP klase i NP-potpun? Na kraju je navedena je opća procedura za rješavanje igre. Poglavlje će se ukratko dotaknuti i opisati najpopularniju verziju igre.

2.1. Klondike pravila igre

Klondike igra počinje tako da se podijeli 28 karata na 7 stogova, to se naziva tablica. Samo jedna karta je na prvom stogu, a svaki idući stog ima jednu kartu više s tim da na početku, sa svakog stoga samo je gornja karta okrenuta licem prema gore. U tablici se karte slažu silaznim redoslijedom po vrijednosti i bojama izmjenično. To znači da na crnu kartu može se staviti samo crvena karta jednog nivoa manje vrijednosti i obrnuto, na crvenu kartu može se staviti crna karta jedan nivo slabija. Primjer, na kartu 5 pik, u tablici se može staviti samo 4 srce ili 4 karo. Kada se premjeste sve licem gore okrenute karte sa jednog stoga tablice, tada se može okrenuti samo gornja karta, s vrha stoga, koja je okrenuta licem prema dolje. Na mjestu u tablici gdje više nema karata moguće je postaviti samo kraljeve kao prvu kartu na koju se dalje slaže niz. Asevi, kada postanu dostupni mogu se staviti iznad tablice, to mjesto se naziva temelj. Predviđena su četiri mjesta za temelje. Na temelje karte se slažu uzlaznim redoslijedom, gdje također se mora poštivati odijelo karte. To znači da svaki temelj prihvaća jedno odijelo i karte se slažu od najmanje prema najvećoj, počevši s asem. Karte se iz špila (talona) izvlače jedna po jedna ili u grupama po tri, ovisno o kojoj verziji klondike-a se radi. Izvučene karte se slažu prema navedenim pravilima, a igra završava kada se slože sve karte na samo 4 stoga u tablici ili 4 stoga na temelju.

2.2. Deterministički problem pasijansa

Klondike je kartaška igra, no može se nazvati i slagalicom. Igra sama po sebi nije kompetitivna jer je samo za jednog igrača, ali moguće je u njoj itekako pronaći izazov. Najveći izazov u ovoj igri su pronašli matematičari. Još uvijek nije pronađen odgovor na pitanje koliki postotak klondike pasijans igara je moguće završiti. Kako bi se pronašao potpuno točan odgovor na ovo

pitanje potrebno bi bilo analizirati svih 52 faktorijela različitih mogućih igara, što je u praksi nemoguće.

2.2.1. Računska teorija kompleknosti

Igranje obične klondike verzije uključuje određenu dozu rizika zbog još neotkrivenih karata. Planiranje igranja se temelji na vjerojatnosti jer ishod otkrivanja karte je jedna vjerojatnost iz skupa svih još neotkrivenih karata. Ipak, većina poteza u igri je deterministička, jer tek nakon preseljenjenja karte, tek kada se oslobodi neotkrivena karta saznaje se ishod. Za potrebe detaljnijeg sagledavanja igre pasijansa matematičari su uveli novu verziju igre zvani thoughtfoul solitaire za potrebe proučavanja kombinatorike igre. Ova verzija igra se po standardnim klondike pravilima, ali su sve karte vidljive. Sa svim unaprijed poznatim kartama igraču je ostavljeno mnogo više prostora za taktiziranje. Kada su sve karte otkrivene to znači da u igri nema više probabilističkog planiranja već su svi daljni potezi deterministički. Deterministički znači da računalo generira izlaz na temelju unosa i koraka koji prate predviđenu logiku. Nedeterminističko računanje može uključivati unos korisnika i dodatno mjesto za pohranu informacija pa čak i izvođenje više zadataka odjednom. Nedeterministički način je zato mnogo učinkovitiji i koristi manje memorije, ali ne nužno i manje vremena. Ovaj način ima u cilju skratiti vrijeme računanja preskakanjem određenih koraka, dok deterministički način zahtjeva pamćenje svih prijašnje izvršenih koraka.

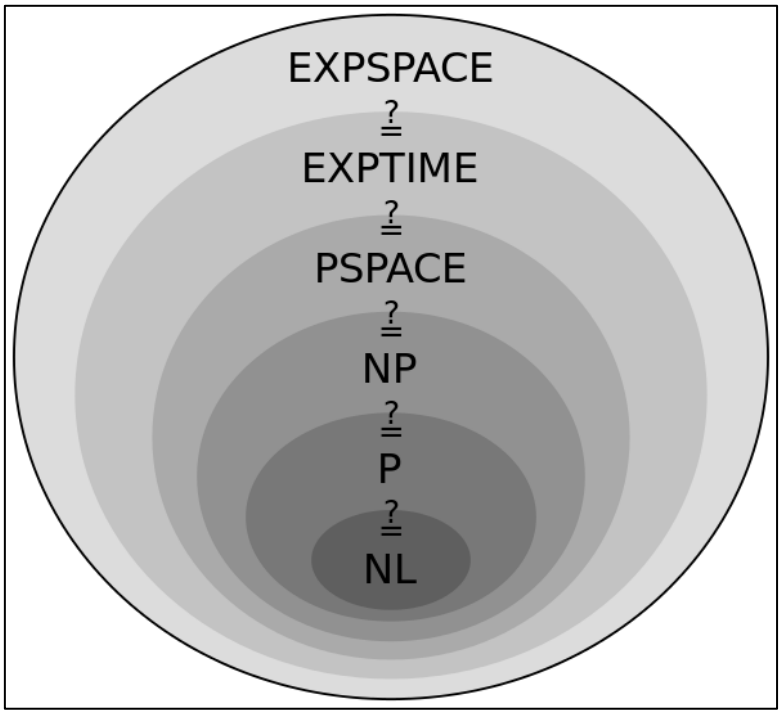
U računskoj teoriji kompleksnoti, prema [2,3], želja je sve probleme uspjeti izračunati u polinomijalnom vremenu jer je učinkovitije. Polinomijalno vrijeme znači da vrijeme potrebno računalu da prođe određeni broj koraka je ograničeno gornjom granicom kao veličina polinomske funkcije. Broj koraka algoritma je ograničen polinomskom funkcijom $f(n)$ gdje je n duljina ulaza za promatrani problem.

Neka je $f(n) = c_k n^k + c_{k-1} n^{k-1} + \ldots + c_1 n + c$ polinomska funkcija, a c_k konstanta. Funkcija $f(n)$ je stupnja k, što znači da je k najveća potencija od n. Kompleksnost se zapisuje kao $f(n) = O(n^k)$.

Neka su f i g dvije funkcije iz $\mathbb{N} \to \mathbb{N}$. $f(n) = O(g(n))$ i postoje $c, m \in \mathbb{N}\ \forall\ n \geq m$ tako da vrijedi $f(n) \leq c \cdot g(n)$, tj. funkcija f raste jednako brzo ili sporije u odnosu na funkcija g.

Primjer: Kompleksnost obilaska matrice dimenzija n x n se zapisuje kao $O(n^2)$.

Teorija kompleksnosti se može objasniti kao podjela problema odlučivanja u zasebne klase prema njihovoj težini računanja. Klasa kompleksnosti je skup problema odlučivanja koji se mogu računalno odlučivati. Algoritam za problem odlučivanja je konačan niz operacija koje navode izračun koji rješava sve dijelove problema u konačnom vremenu. Algoritam se vidi kao nešto u potpunosti determinirano, da uvijek ima put koji može izabrati da nastavi svoje izvođenje. Nedeterministički algoritam odlikuju dvije faze, faza nagađanja i faza potvrđivanja. Faza nagađanja odabira proizvede string simbola koji daju potencijalno rješenje. Zatim faza potvrđivanja koristi deterministički algoritam koji preispituje taj proizvedeni string u potrazi za rješenjem problema.



Slika 2.1. Klase kompleksnosti [2]

Problem odlučivanja P klase (polinomijalno vrijeme) kompleksnosti ima algoritam koji tako zvanim Turingovim strojem do rješenja problema može doći u polinomijalnom vremenu. NP (nedeterminističko polinomijalno vrijeme) je veća klasa problema odlučivanja. Rješenje problema odlučivanja NP klase kompleksnosti može se provjeriti, ako postoji rješenje, tada mora postojati deterministički algoritam koji može ispitati rezultat u polinomijalnom vremenu. Zato se problemi P klase se smatraju jednostavnim i mogućim za pratiti. NP problem je nedeterministički što znači da nema neki utvrđeni skup pravila koja može slijediti do rješenja, ali rješenje može provjeriti u klasi P. Za sada se zna da je $P \subset NP$, ali još uvijek stoji pitanje vrijedi

li tvrdnja $P = NP$. Postoji li brzo i efikasno rješenje, za svaki problem, čije se rješenje može brzo provjeriti? Može li se NP težak problem riješiti u vremenu potrebnom da bi se riješio P problem? Budući da je rješenje NP problema moguće provjeriti u vremenu potrebnom za P problem. Slika 2.1. ilustrira klase problema i pitanje pripadnosti klase većoj klasi.

2.2.2. Strategije računala

Do rješenja ovakvih problema dolazi se simuliranjem pokusa. Metoda korištena za rješavanje ovog problema je Monte Carlo metoda [4]. Monte Carlo simulacija je metoda procjene vrijednosti, to su simulacije koje nasumično evoluiraju. Većim brojem uzoraka procjena je bliža stvarnoj vjerojatnosti. Monte Carlo se koristi kada postoji neizmjerno velik broj mogućnosti koji je neizvedivo simulirati u potpunosti. Monte Carlo metodom simulacije se ciljano razmatra samo nasumični podskup svih mogućnosti. Potrebno je dopustiti simulaciji da se razvija uzimanjem uzoraka nasumično iz neke velike skupine mogućnosti jer je sve većim uzorkom rezultat pouzdaniji.

Opća procedura za igranje pasijansa, prema [5]:

1. Ustanoviti kolekciju legalnih poteza
2. Odabrati i izvršiti legalni potez
3. Ako su sve karte nalaze u 4 stoga, proglasi pobjedu i završi proceduru
4. Ako nova konfiguracija karata ponovi prethodnu, proglasi gubitak, završi proceduru
5. Ponovi proceduru

Način pamćenja konfiguracije karata je jednostavno da se spremaju sve već prethodno testirane postave. Procedura se još zaustavlja kada se suoči sa beskonačnom petljom okretanja karata iz špila (prebacivanja karata špil-talon), prebacivanja karte između dva stoga u tablici ili prebacivanja karte između stoga tablice i stoga temelja.

Razvoj heurističkih algoritama koji se mogu bolje prilagođavati problemu daju sve bolje rezultate. Heuristika još uvijek ne garantira pronalazak rješenja, ali je brzi način pokušaja dolaska do rješenja. *Rollout* je vrsta sekvencijalne optimizacije u dinamičkom programiranju. Jedan r*ollout* se može smatrati jednom iteracijom temeljne metode. Neka h bude strategija koja

podijeljenim kartama x pridružuje potez $h(x)$. *Rollout* metoda je procedura koja unapređuje početnu strategiju h te pritom generira novu unaprijeđenu strategiju h'. Nova h' *rollout* strategija nakon $h'(x)$ poteza neke procedure može razviti novu strategiju $h''(x)$. Jednostavno je dokazati da konačan broj iteracija će doći do optimalne strategije, ali pri tome računalno vrijeme eksponencijalno raste.

Tablica 2.1. Pokus rješivosti igre [5]

Igrač	**Uspješnost**	**Odigrano igara**	**Prosječno vrijeme po igri**	**99% granica pouzdanosti**
Čovjek	36.6%	2 000	20 min	±2.78%
Heuristika	13.05%	10 000	.021 s	±.882%
1 rollout	31.20%	10 000	.67 s	±1.20%
2 rollouts	47.60%	10 000	7.13 s	±1.30%
3 rollouts	56.83%	10 000	1 min 36 s	±1.30%
4 rollouts	60.51%	1 000	18 min 7 s	±4.00%
5 rollouts	70.20%	200	1 h 45 min	±8.34%

U tablici 5.1. nalaze se rezultati pokusa koji pokazuju koliki postotak igara je riješeno kad je igrao čovjek i kad je korišteno računalo sa sve boljom optimizacijom algoritma za igranje. Za klondike gdje se iz talona izvlači po jedna karta, eksperimentom se došlo do zaključka da je najmanje 81.9% igara rješivo, a najviše 18.1% igara je nerješivo [5].

2.2.3. Zašto je klondike NP problem

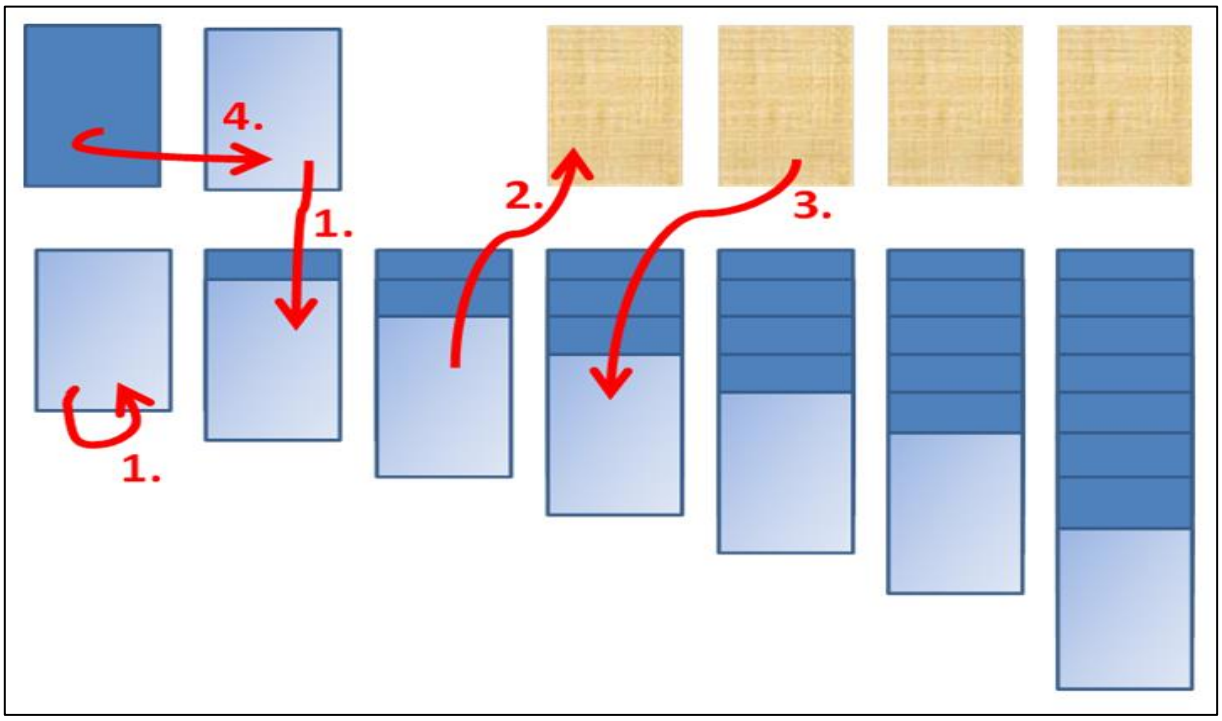
Pretpostavite rješivu klondike instancu igre sa N karata koja ima k stogova tablice. Pretpostavite da je najkraća rješiva sekvenca poteza igre polinomijalna u N. Tada se definiraju 4 vrste poteza, pema [6], ilustrirani su na slici 2.2., a to su:

1. vrsta: Premiještanje karte sa talona, ili bilo koji potez koji preokreće još neotkrivenu kartu iz tablice. Ovo uključuje bilo koji potez sa stoga tablice na drugi stog ili premještanje posljednje otkrivene karte iz tablice na temelj, osim kada takav potez potpuno isprazni stog. Potezi ove vrste su nepovratni.

2. vrsta: Premještanje karte sa stoga tablice na temelj, ali da nije potez **1.** vrste.

3. vrsta: Premještanje karte s temelja u tablicu.

4. vrsta: Otkrivanje nove karte iz talona.



Slika 2.2. Vrste poteza [6]

Prema [6], neka ℓ bude najkraća sekvenca koraka za prelazak igre. Takva sekvenca sadrži $N - k$ koraka **1.** vrste jer je točno toliko neotkrivenih karata na početku igre, odnosno, k karata poznatih u tablici na početku. Tvrdnja ide da dva uzastopna poteza **1.** vrste u ovoj sekvenci su

odvojeni sa $O(N)$ koraka **2.**, **3.** ili **4.** vrste. Tvrdnja za NP gornju granicu slijedi iz tvrdnje da potezi **1.** tipa su nepovratni i ne preostaje nijedna prepreka nakon $N - k$ poteza **1.** vrste, tako da ℓ je $O(N^2)$. Tvrdnja da postoji rješiva sekvenca, znači da postoji jedna sekvenca, gdje se potezi između dva uzastopna poteza **1.** vrste sastoje od niza od najviše N poteza **2.**, najviše N poteza **3.** i najviše N poteza **4.** vrste. Potezi **2.** i **3.** vrste se ne miješaju sa potezima **4.** vrste tako da se svi potezi **4.** vrste mogu se odgoditi sve dok potezi **2.** i **3.** vrste nisu završili. Pretpostavlja se da će se potezi **3.** tipa pojaviti posljednji u optimalnoj sekvenci poteza **2.** ili **3.** vrste. Ne može postojati više od N uzastopnih poteza **2.** vrste i ne više od N uzastopnih poteza **3.** vrste. Stoga treba razmotriti konfiguraciju C i optimalan slijed poteza $\mathcal{s}$ **2.** ili **3.** vrste koji vode do konfiguracije C'. Može se dokazati za broj k poteza **3.** vrste da odgađanje poteza **3.** vrste u $\mathcal{s}$ sve do kraja i dalje vodi do konfiguracije C', to je zato što karte koje se nalaze na temeljima bi ionako došle na vrhove hrpa u tablici. Neka je δ prvi potez trećeg tipa u sekvenci $\mathcal{s}$, makne kartu c sa temelja σ_c. Neka C_1 bude konfiguracija do koje se dolazi iz C primjenom prefixa $\mathcal{s}$ sve do, i uključujući δ. Budući da ostatak $\mathcal{s}$ vodi optimalno {po mogućnosti"} od C_1 do C' u $k-1$ poteza **3.** vrste, prema hipotezi indukcije ti potezi **3.** vrste mogu biti odgođeni, kako bi bili nakon poteza **2.** vrste, to zanči da karte mogu čekati na temeljima, ne smetaju slaganju tablice jer se u tablici karte slažu silazno pa bi te karte sa temelja ionake došle na vrh stoga u tablici. Neka je $\mathcal{s}'$ sekvenca koja optimalno vodi od C do C', dobivena preko $\mathcal{s}$ odgađanjem njezinih $k-1$ poteza **3.** vrste sve do zadnjega. Primjećuje se da nijedan potez **2.** vrste uključujući σ_c se ne može pojaviti nakon δ u $\mathcal{s}'$ sve dok c nije vraćen na σ_c jer se niz temelja slaže redoslijedom u istom odijelu karte, nema druge karte iste boje da posluži kao zamjena. Ali premještanje c natrag na σ_c nakon δ bi poništilo δ, što nema smisla za sekvencu $\mathcal{s}'$. Stoga nijednan potez drugog tipa poslije δ u $\mathcal{s}'$ koji uključuje σ_c se ne može izvesti, tako da δ se može odgoditi nakon svih poteza u $\mathcal{s}'$. Stoga između bilo koja dva uzastopna poteza prvog tipa u najkraćoj sekvenci može postojati najviše 3N poteza **2.**, **3.** ili **4.** vrste.

Ako je dijeljenje igre rješivo tada, neka za primjer, x bude najgori mogući slučaj dovođenja karte na svoje mjesto između dva poteza prve vrste. Tada za 52 karte, neka se vremenska kompleksnost zapiše kao $O(x^N) = O(x^{52})$. Za običnu igru je dovoljno reći, za kompleksnost, da je $O(x^{N-k}) = O(x^{45})$ jer je k karata uvijek poznatih na početku. Kada je kompleksnost provjere rješenja manja ili jednako velika kompleksnosti polinoma problema $O(x^{45}) \leq O(x^{52})$ znači da je problem moguće provjeriti u P klasi kompleksnosti. Nezna se kolika bi kompleksnost procedure za pronalazak rješenja igre morala biti da se zaključak može donijeti sa potpunom

sigurnošću za svako dijeljenje, ali je svako rješenje moguće provjeriti u polinomijalnom vremenu pa zato problem pripada NP klasi.

Još uvijek nije uspješno dokazano koja je vjerojatnost rješivosti jedne igre niti koji je najbolji način igranja. Računska teorija složenosti ukazuje kako je računalna kompleksnost rješivosti igre NP-potpun problem, to je samo teoretski prikaz kompleksnosti problema. To je problem na koji se gleda kao najteži problem skupa NP problema, gdje je NP skup problema odlučivanja rješivih u polinomijalnom vremenu pomoću „sretnog“ algoritma. Unatoč tome što je većina problema odlučivanja rješiva nije moguće uvijek do rješenja doći na *brute force* način, zato se koristi nedeterministički model pa algoritam smije raditi nagađanja da se zaobiđe mnoštvo nepotrebnih koraka. Iskorištena nagađanja moraju uvijek garantirati potvrdni odgovor da algoritam nastavi svoje izvođenje prema kraju. NP problemi odlučivanja imaju rješenje koje je moguće provjeriti u polinomijalnom vremenu, što znači da sve dok se na pretpostavku dobije potvrdni odgovor može se i dokazati, provjeriti u polinomijalnom vremenu. Zato je kao dokaz dovoljna samo sekvenca poteza mogućih za izvesti. Upravo zbog tih korištenja pretpostavki, algoritam dobiva naziv sretan. Rješavanje problema je obično puno teže od provjere rješenja problema. Tako kod pasijansa, lagano je implementirati igru, ali teško je pronaći koje korake treba slijediti do rješenja.

2.3. Implementacije za različite platforme

Samo određeni broj klondike pasijans dijeljenja je rješiv u potpunosti. Znatan broj dijeljenja, skupine rješivih dijeljenja jesu vrlo često nerješiva zbog same težine, za rekreativnog igrača. Next Level Games studio je razvio Microsoft Solitaire Collection igru, Microsoft ju je uključio u svoj operacijski sustav. Igra uključuje nekoliko verzija pasijansa, ali ključna stvar kod te igre je da postoje opcije izbora težine igranja. To je takoreći velika kolekcija rješivih dijeljenja koja su kategorizirana po težini igranja. Igraću je raspoloživ odabir razine izazova igranja. Igru obilježavaju nepreskočive reklame trajanja po trideset sekundi, gdje ako se prozor igre ne nalazi u prvom planu, vrijeme odbrojavanja reklame će biti stopirano. Aplikacija ovog završnog rada nema sustav bodovanja, nema izbor težine, ali nema ni relama.

Unity, prema [7] je više platformski softver koji uz plaćenu licencu omogućuje razvoj igara za: iOS, Android, Windows, Universal Windows Platform, Linux, WebGL, PS4, PS5, XBOX ONE, Oculus Rift, AndroidTV, tvOS, Nintendo Switch, ARCore, Microsoft HoloLens, Magic Leap. Baš zato što podržava toliko platformi Unity je dobar izbor za razvoj igara. Tako je za razvoj

igre na nekoj drugoj platformi jednostavno moguće samo uključiti u projekt paket za kontrolu sustava unosa (engl. *Input System Controller*) i koristiti shemu kontrola za pripadajuću platformu. Kao dobar primjer ovoga može se navesti jedna od najpopularnijih kartaških igara ikad, Hearthstone. Blizzard Entertainment je razvio Hearthstone u Unity-u i igra je dostupna na Windows, macOS, iOS, Android platformama.

3. APLIKACIJA ZA IGRANJE KARTAŠKE IGRE PASIJANS

Poglavlje objašnjava uloge osnovnih građevnih blokova Unity projekta, a to su komponente i skripte. Ovo poglavlje je dokumentacija, objašnjava izvedbu ovog programa u Unity-u. Poglavlje opisuje ciljeve, određene razloge strukturiranja klasa i korištenja određenih obrazaca oblikovanja. Poglavlje sadrži slike sa UML dijagramima koji prikazuju strukturu klasa.

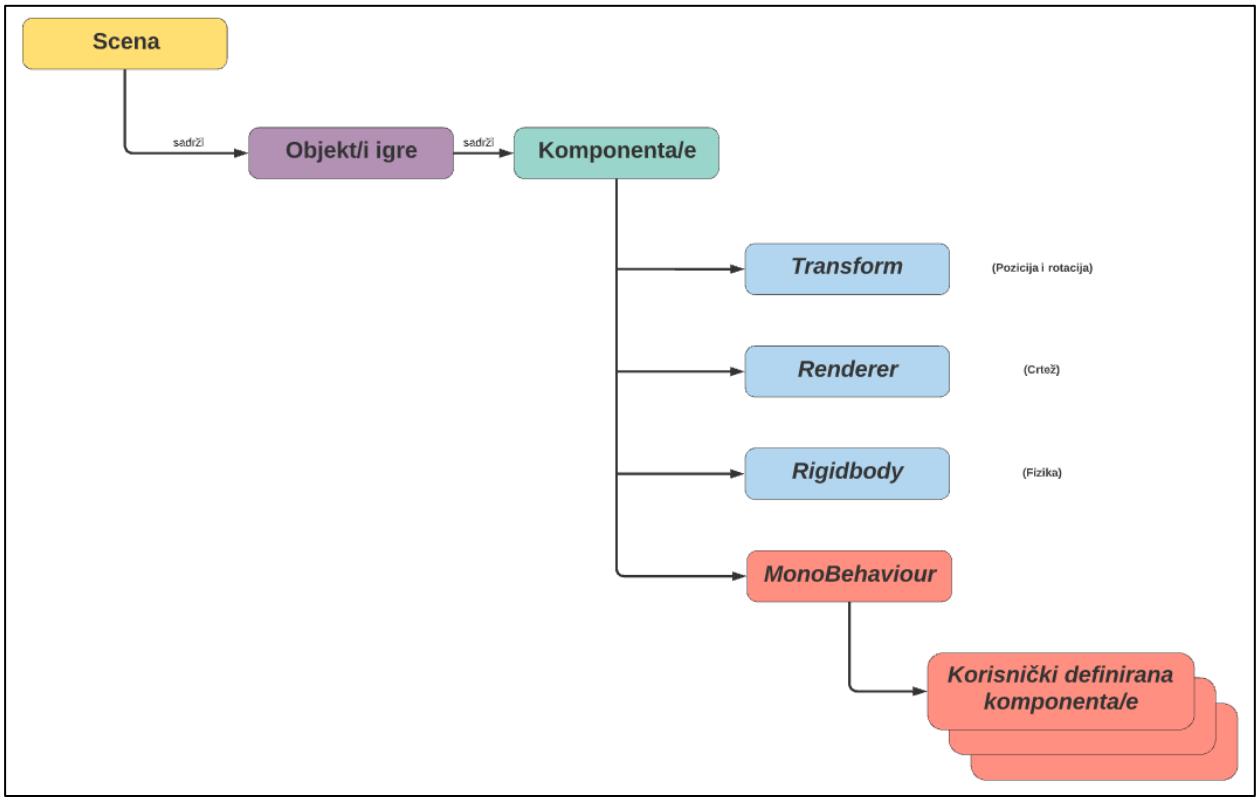
3.1. Arhitektura Unity *Engine*-a

Termin *Game Engine*, prema [8] se pojavio 1990-tih godina kad su razvojni timovi počeli odvajati jezgru softvera igre (renderiranje grafike, detekciju kolizija) i dizajn. Razina separacije je omogućila razvoj novih proizvoda sa minimalnim promjenama na enginu igre. Postalo je moguće da se mijenjaju samo one stvari koje čine iskustvo igrača kao što su novi dizajn svijeta, likovi, itd.

Unity je softver koji omogućuje višeplatformski razvoj video igara, razvijen je s ciljem objedinjenja industrije za razvoj video igara. Većina *Game Engine-a* je izgrađena da pokreće jednu igru ili određeni žanr igre na odabranoj platformi. Unity (hrv. *Jedinstvo, ujedinjenost*) ujedinjuje najbolje prakse industrije i integrira ih u jednu cjelinu stoga ga još nazivaju i žanr-agnostičkim. Izvorni kod samog Unity-a nije dostupan, ali zato sami engine brine o memoriji. Dvjie glavne osnove arhitekture Unity-a, glavna stupa izgradnje su: komponente i API za izradu skripti.

3.1.1. Komponente

Objekt igre (engl. *Game Object*) je najosnovniji koncept u Unity-u, to je svaki objekt u igri, od svjetla, kamere, predmeta do specijalnih efekata. Slika 3.1. će bolje ilustrirati kako je objekt smješten na sceni i koje komponente sadrži.



Slika 3.1. Prikaz hijerarhije objekata u Unity-u [9]

Scena (engl. *Scene*) je kolekcija svih objekata igre, dok je objekt igre kolekcija komponenata (engl. C*omponents*). Preko komponenata objekti igre dobivaju svojstva i funkcionalnosti, objekt dobiva smislen oblik da predstavlja nešto u igri. Preko komponente se postavljaju svojstva fizike, konfiguracije, animacije, teksture, ponašanja i događaji. Primjerice, svaki objekt igre ima Transform komponentu, koja se ne može izbrisati, a ona omogućuje pozicioniranje objekta u igrinom svijetu, odnosno sceni.

- **Transform** komponenta još uključuje mogućnost slaganja hijerarhije stabla objekata igre što opet znatno olakšava pozicioniranje objekata na sceni jer omogućuje jednostavnije pozicioniranje objekata relativno njihovom roditelju na sceni.

- **Renderer** komponenta prikazuje objekte na ekranu, preko nje se objektu postavlja tekstura.

- **RigidBody** postavlja pravila fizike, reakcijske sile, gravitaciju, masu.

- **MonoBehaviour** je osnovna klasa iz UnityEngine biblioteke koju moraju naslijediti sve Unity skripte

3.1.2. API za izradu skripti

API za izradu skripti (engl. S*cripting Application Programming Interface*) je kolekcija biblioteka koje sadrže već gotove klase, sučelja, događaje, te tako omogućuju rad sa *Game Engine-om*. Unity je dizajniran s naumom da se u razvoj igara mogu uključiti programeri različitog predznanja programiranja, a isto tako da je moguć razvoj bilo kakve vrste video igre. Izvorni kod Unity-a je pisan u C++ zbog potrebe za preciznijom kontrolom nad memorijom i korištenjem procesora i zato korisnici ne moraju brinuti o memoriji unutar samog *engine-a*. Postignut je fleksibilniji pristup razvoju video igara tako da je dodana nova razina apstrakcije na razinu jezgre pisane u C++. Na ovaj način je zaštićen unutarnji mehanizam *engine-a* od loše implementiranog koda. Osnovne funkcionalnosti i biblioteke *engine*-a su uručene toj višoj razini apstrakcije preko API-a za izradu skripti, podržava C# i Java jezike za skriptanje.

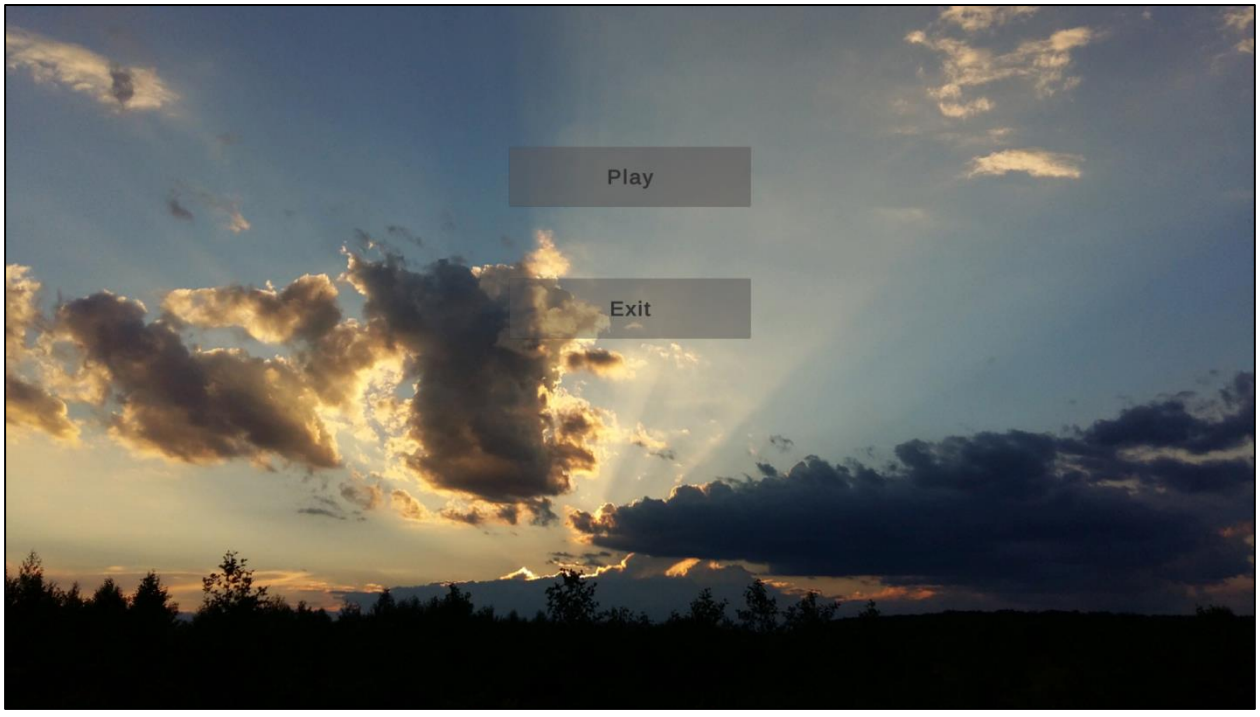
MonoBehaviour klasa se može smatrati nacrtom za kreiranje novih komponenata. Tako novokreirana skripta implementiranjem MonoBehaviour klase povezuje se sa unutarnjim mehanizmom Unity-a. Skriptama se opisuju ponašanja objekata igre.

3.2. Implementacija igre

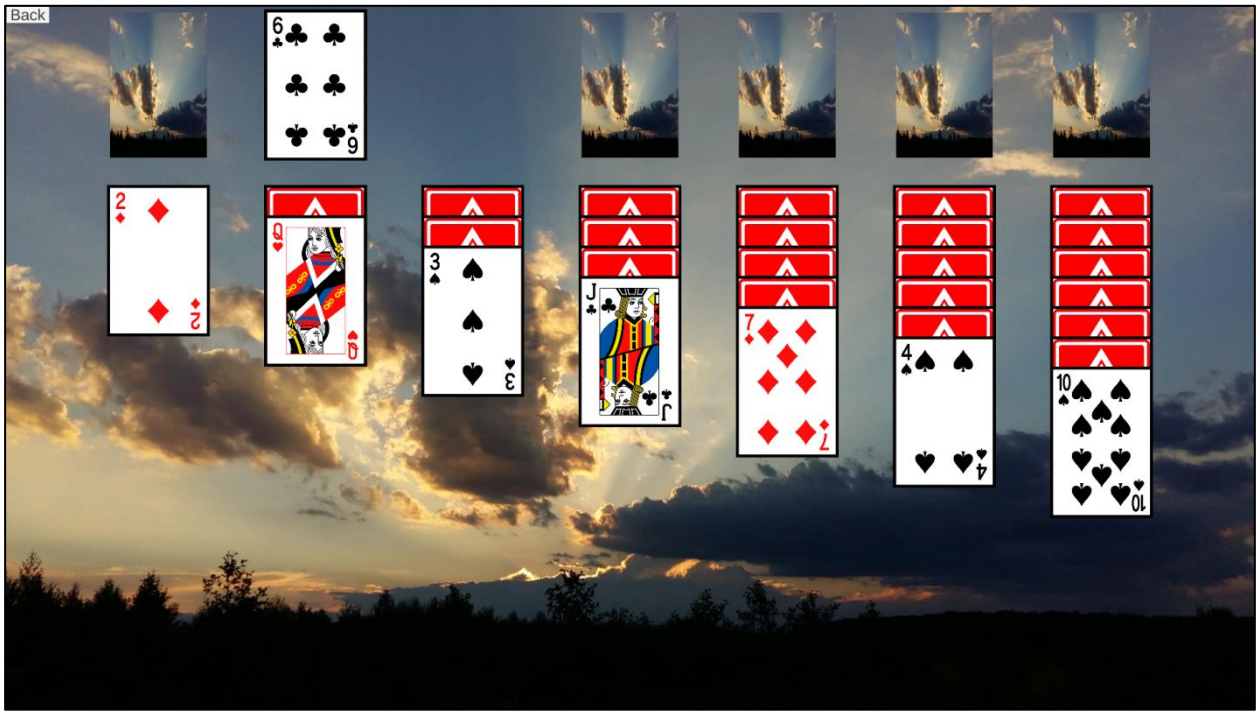
Ovo poglavlje služi kao dokumentacija koda igre. Poglavlje sadrži UML klasne dijagrame, opisuje strukturu klasa, navodi razloge i ciljeve korištenja određenih oblikovnih obrazaca, te njihove pozitivne i negativne učinke. Igra je implementirana u Unity Game Engine-u (verzija: 2019.4.13f1). Logika igre je pisana u C# programskom jeziku, a za uređivanje programskog koda korišten je Microsoft Visual Studio 2019. UML klasni dijagrami su napravljeni na Lucidchart web stranici [10].

3.2.1. Dizajn aplikacije

Postoje dvije scene u projektu. Prva scena je glavni izbornik (slika 3.2.) iz kojeg se dolazi na scenu za igranje (slika 3.3.). Za izlazak iz aplikacije potrebno se je vratiti na glavni meni, ali svaki povratak na glavni meni uništava scenu igre. U gornjem desnom kutu scene igre se nalazi tipka za povratan na glavni meni. Povratakom u glavni meni će se izgubiti sav napredak na trenutnom dijeljenju. Stoga je dodan skočni prozor za potvdu izlazska iz scene igre.



Slika 3.2. Slika zaslona glavnog izbornika



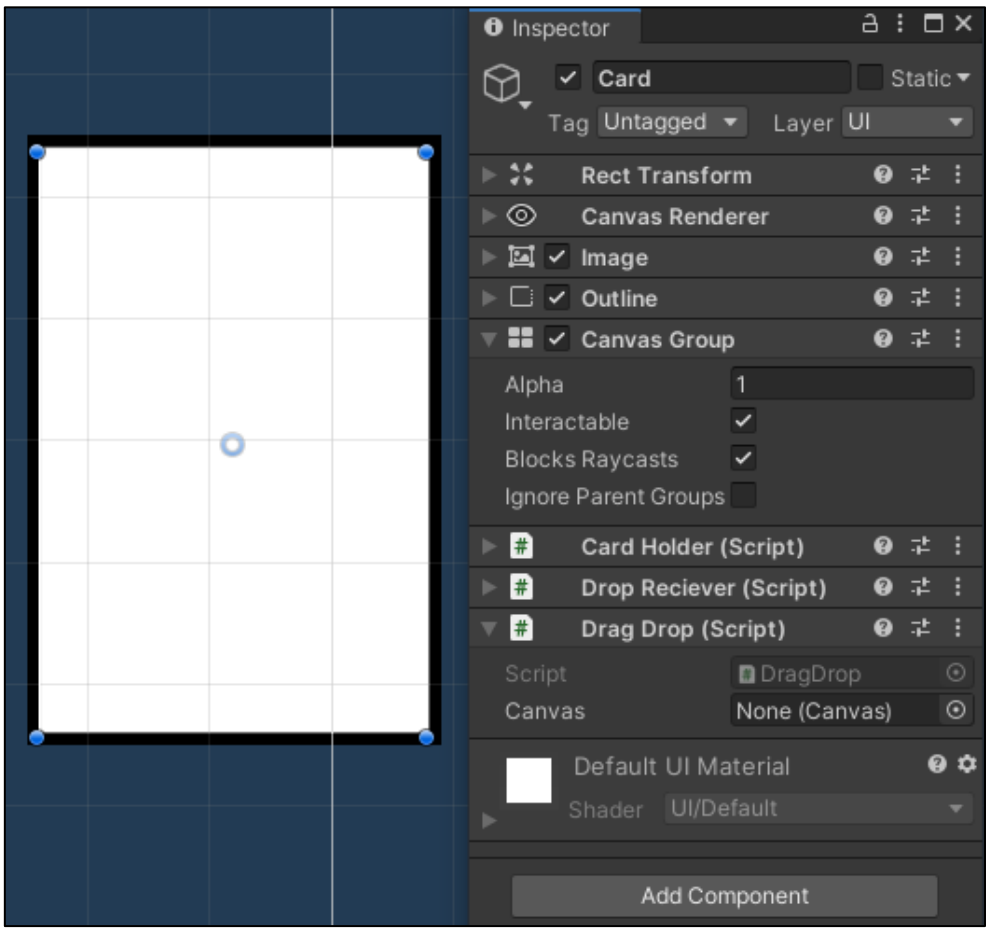
Slika 3.3. Slika zaslona igre

3.2.2. Preddefinirani objekti

Za potrebe implementacije pasijansa preddefinirana su tri objekta (engl. *Prefab*) kojima se gradi igra. To su već konfigurirani objekti spremljeni u projektu. Koriste se kao nacrt i to je njihova glavna karakteristika gdje je omogućeno ponovno stvaranje istih objekata po jednom nacrtu. Tako za slučaj klondike pasijansa je potrebno instancirati mjesto za talon, 52 igraće karte, 7 temelja na koje se karte moraju slagati silaznim i 4 temelja na koje se karte moraju slagati uzlaznim redoslijedom. Objekti su uneseni kao 2D igrini objekti iz kategorije objekata korisničkog sučelja, stoga svaki na sebi ima *Image* komponentu preko koje se postavlja textura. Svi ti objekti po izgledu su slični, istih su dimenzija i pravokutnog oblika. Razlika je u komponentama koje sadržavaju u sebi. Objekti su konfigurirani da sadrže potrebne skripte za zamišljenu funkcionalnost. Svi objekti instancirani po istom nacrtu imat će iste komponente, tako da kontroliranjem procesa instantacije moguće je vrlo jednostavno mijenjati njihove parametre.

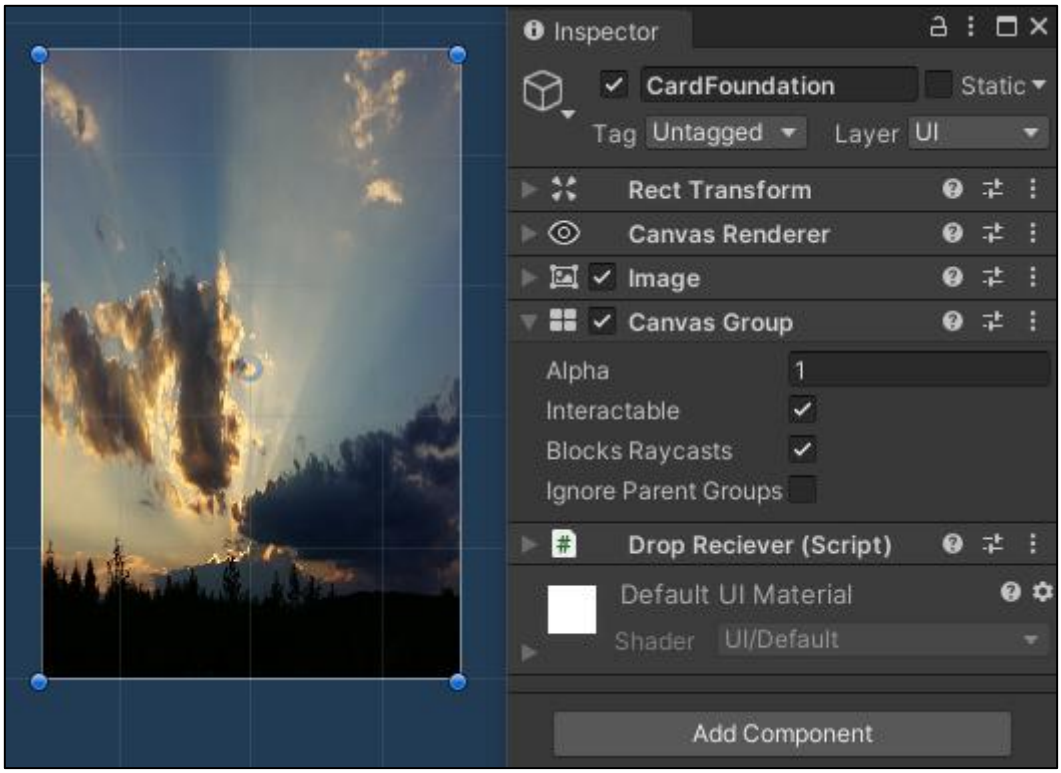
Sve te objekte bilo bi moguće postaviti odmah na scenu pa na taj način dohvaćati njihove reference i postavljati im texture i vrijednosti, ali vrlo efikasnije rješenje za ovo je instancirati objekte u programskom kodu. Na ovaj način objektu se može brzo pristupiti, referenca objekta je

odmah dostupna jer je na tome istome mjestu objekt instanciran, tako da se objektu odmah mogu postaviti potrebni parametri. Fokus izdvajanja objekta da se koristi kao nacrt pruža mogućnost ponovnog korištenja gotovih dijelova projekta, za veliki broj instanci istog objekta ili primjerice za nadogradnju programa na neku drugu verziju. Proširenje na drugu verziju pasijansa koja ima slične mehanike igranja, čije se mehanike mogu postići starim objektima uz male preinake koda. To bi značilo da se korištenjem kompozicije ugradi drugačija funkcionalnost Korištenjem takvih struktura prilikom razvoja ostavlja se mogućnost jednostavnog proširivanja.



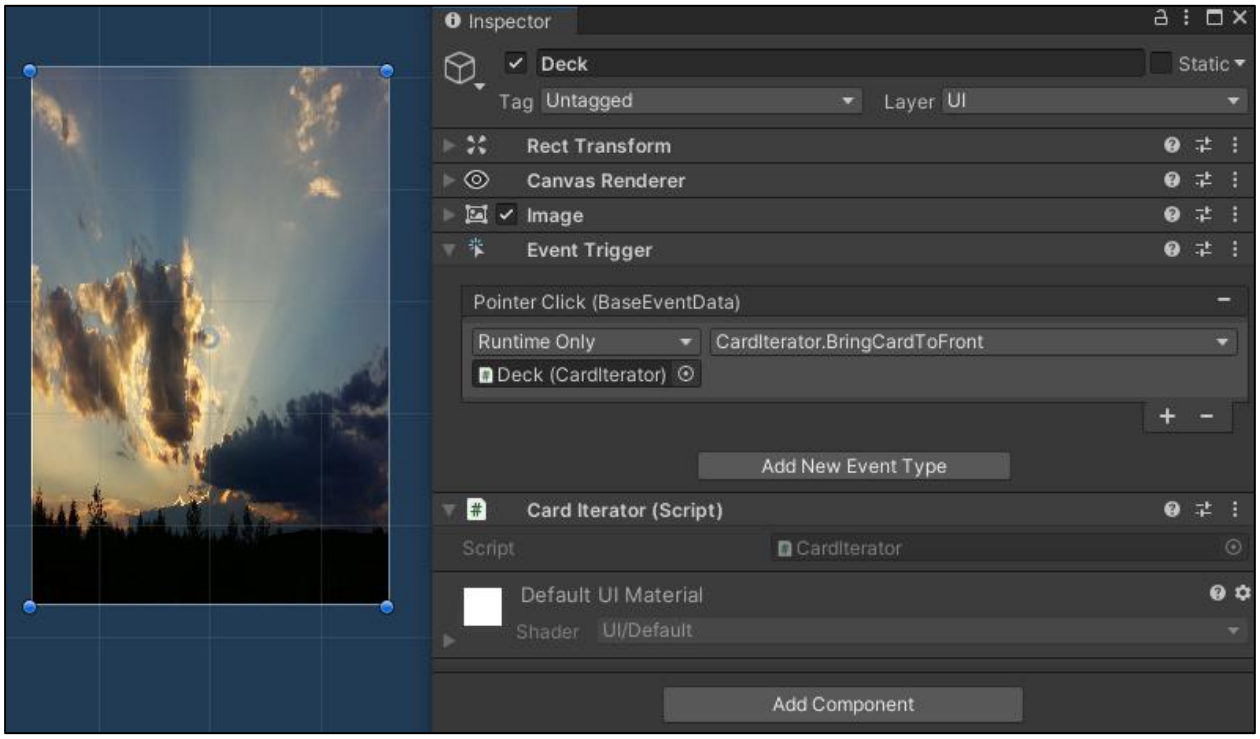
Slika 3.4. *Prefab* i inspektor karte

Prvi predgotovljeni objekt reprezentira igraću kartu (slika 3.4.). On u sebi sadrži skriptu koja se koristi kao skladište za igraću kartu dodijeljenu tom objektu, u sebi ima spremljenu boju i vrijednost karte za daljnje usporedbe i teksturu te karte. Ima još skriptu koja omogućuje da se karta može vući po ekranu i skriptu koja prima kartu kada se ispusti.



Slika 3.5. *Prefab* i inspektor temelja

Drugi predgotovljeni objekt je temelj, on može samo primiti određenu vrstu karte tako da za razliku od objekta karte on ne može biti povlačen po ekranu i zato ima samo skriptu za primanje karte (slika 3.5.).

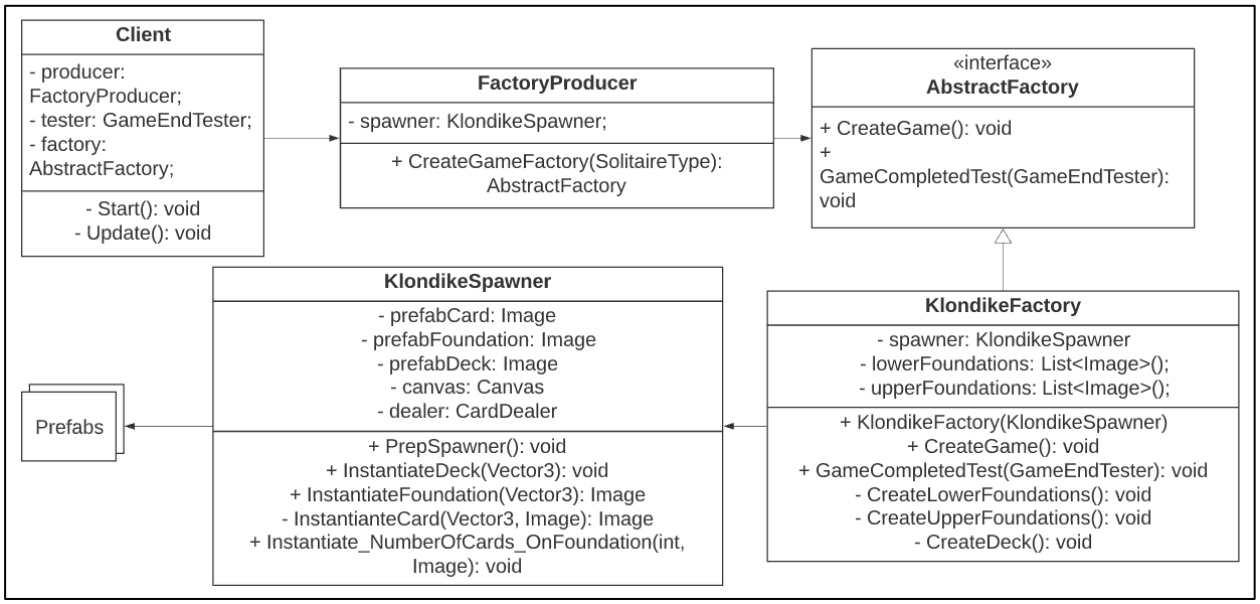


Slika 3.6. *Prefab* i inspektor talona

Komponente talon objekta su *CardIterator* skripta i okidač događaja (engl. *Event Trigger*). Klikom na objekt zove se funkcija unutar *CardIterator* skripte (slika 3.6.). Svaki klikom na objekt skripta ciklički jednu po jednu kartu, okreće karte iz talona i svaku iduću novu kartu stavlja na vrh.

3.2.3. Kreiranje objekata scene

Pri pokretanju igre iz glavnog menija dostupno je samo prazno platno (engl. *Canvas*). Platno je ostavljeno prazno kako ne bi sama scena morala nadzirati proces kreacije igrinih objekata. U igri je potrebno napraviti raspored, razmještaj karata i ostalih elemenata koji su definirani pravilima igre. Kod klondike pasijansa su to tablica, temelji i talon. Proces kreiranja tih elemenata je izdvojen u jedan objekt kako bi se ostavila mogućnost dodavanja novih verzija bez narušavanja postojeće strukture. Tipovi pasijans igara jesu igre svaka za sebe, stoga korištenje apstraktne tvornice ostavlja priliku za proširenje programa.

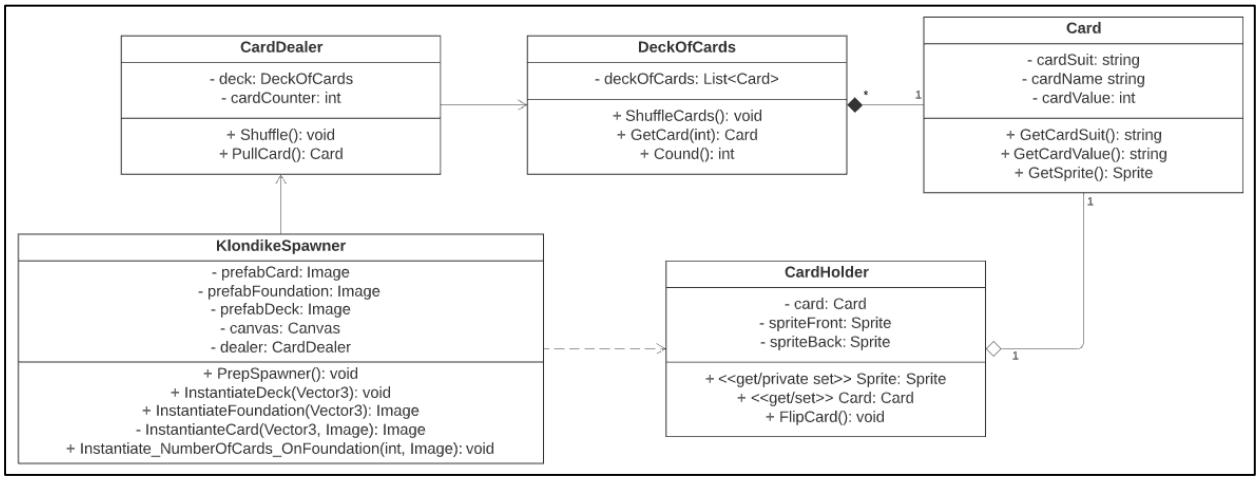


Slika 3.7. Apstraktna tvornica

Slika 3.7. prikazuje strukturu klasa za implementaciju oblikovnog obrasca apstraktne tvornice. Stvaranje objekata se izdvaja iz klijentskog koda. *Client* u sebi ima referencu tipa *AbstractFactory* i tako mu je ostavljena opcija izbora kreiranja tvornice gdje se izborom tvornice bira vrsta pasijans igre koja će se igrati, a *FactoryProducer* je zadužen za stvaranje konkretne tvornice koja odgovara zaprimljenom argumentu. Kreiranje igre je izdvojeno u konkretnu tvornicu da bi ostatak sustava bio neovisan o načinu stvaranja. Apstraktna tvornica tako dozvoljava dodavanje novih načina kreiranja i preko tog sučelja osigurava kompatibilnost tvornica ako će doći do nadogradnji u budućnosti. *KlondikeSpawner* skripta sadrži reference na preddefinirane objekte i po tim nacrtima se instanciraju objekti na scenu. *KlondikeFactory* nadzire kreiranje, a *KlondikeSpawner* kreira objekte na poziv tvornice. Tvornica u sebi čuva dvije liste referenci temelja, gornji i donji temelji. Tvornica unutar funkcija za kreiranje temelja ima for petlju i svakom iteracijom izračunava se vektor pozicije te poziva funkciju u *KlondikeSpawner* skripti koja stvara objekt. Postoji klasa *GameEndTester* za provjeru završetka igre, objekt te klase se predaje kao argument metodi *GameCompletedTest()* koja se nalazi unutar tvornice jer tvornica čuva reference na temelje. Testiranje kreće od temelja jer su karte složene u strukturu stabla, a temelj je početni čvor.

3.2.4. Kreiranje i dodjela karata

Karta je ovdje podatkovni objet koji sadrži svoju vrijednost, ime i odijelo. Taj podatkovi objekt se tada sprema u igrin objekt koji se prikazuje na sceni.



Slika 3.8. Iterator

Slika 3.8. prikazuje strukturu klasa za implementaciju *iterator* oblikovnog obrasca. *CardHolder* je skripta igrinog objekta, objekta na sceni, a jedna instanca *Card* objekta je prava igraća karta.

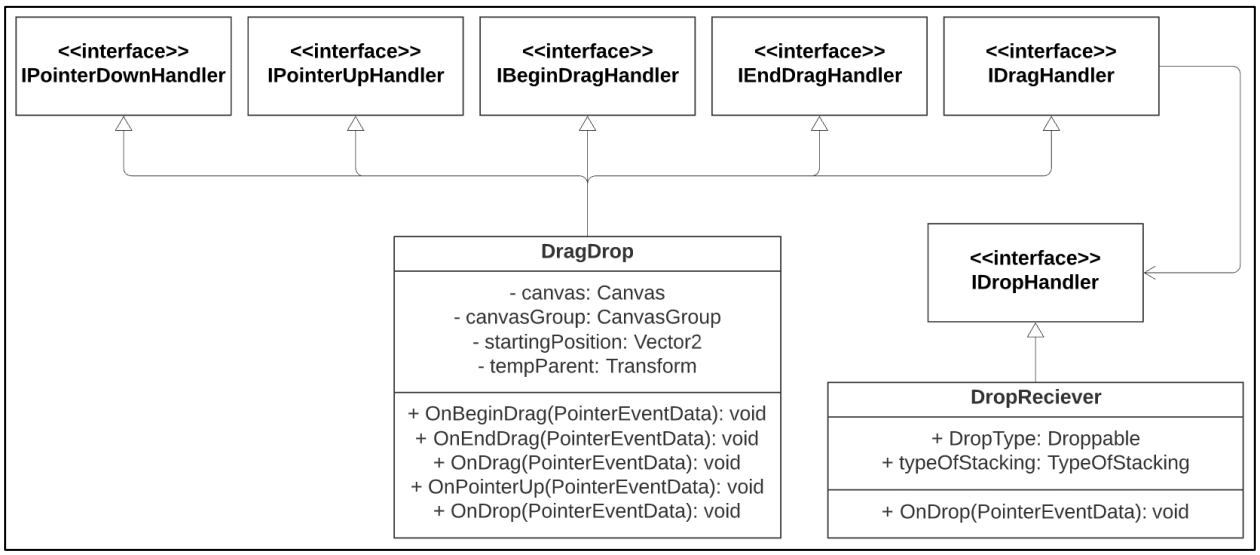
```
public Sprite GetSprite()
{
	return Resources.Load<Sprite>("Textures/" + this.cardSuit + "s/" + cardName +
	"_of_" + cardSuit + "s";
}
```

CardHolder skripta se koristi kao spremnik igraće karte. *Card* objekt je prava igraća karta, čuva podatke koji opisuju kartu: ime, odijelo i vrijednost. Mehanika igre koristi ove podatke za provjeru igrivosti, a *CardHolder* skripta pristupa *Card* objektu za dohvaćanje teksture. Metoda učitava zatraženi tip imovine (engl. *Asset*) iz projekta koji je spremljen na predanoj putanji.

Kada *KlondikeSpawner* instancira igrin objekt karte, tada pristupa *CardHolder* skripti instanciranog objekta i unutra pohranjuje izvučeni *Card* objekt. *Card* objekt je izvučen iz *DeckOfCards* kolekcije posredstvom *CardDealer* objekta. *CardDealer* je iterator koji na zahtjev *KlondikeSpawner*-a uzima kartu iz kolekcije i vodi računa o broju izvučenih karata. Iterator omogućuje pristup elementima kolekcije bez da je potrebno poznavati strukturu kolekcije.

3.2.5. Metoda povlačenja i spuštanja

Korištenjem miša se moguće znatno brže i jednostavnije kretati nego korištenjem samo tipkovnice. Ovdje je objašnjena implementacija povlačenja i spuštanja, i korištena sučelja da bi se to postiglo.



Slika 3.9. *Drag and drop*

Slika 3.9. prikazuje strukturu klasa kako implementirana *drag and drop* metoda. Korištena sučelja se nalaze u *UnityEngine.EventSystems* biblioteci, to je sustav odgovoran za rukovanje događajima unutar scene [11]. Najčešće korištenje sustava je da samo čuva stanje i delegira funkcionalnost na neku komponentu koju je moguće prepisati (engl. o*verridable component*).

IPointerDownHandler

- public void **OnPointerDown**(EventSystems.PointerEventData **eventData**);
- Opis: Detektira klik pokazivača sve do otpuštanja.
- Zadaća: Sprema se trenutna lokacija objekta kako bi se kasnije mogao vratiti na svoje mjesto ako ispuštanje ne uspije.

IPointerUpHandler

- public void **OnPointerUp**(EventSystems.PointerEventData **eventData**);

- Opis: Poziva se nakon otpuštanja pokazivača, ali je potrebno implementirati *IpointerDownHandler*.
- Zadaća: Ako ispušteni objekt nije prihvaćen onda se ovdje taj ispušteni (povlačeni) objekt vraća na početnu lokaciju prije povlačenja.

IDragHandler

- public void **OnDrag**(EventSystems.PointerEventData **eventData**);
- Opis: Poziva se svaki put kada se pokazivač pomakne.
- Zadaća: EventData.delta je pomak pokazivača od prijašnjeg frame-a, taj podatak se mora podijeliti sa faktorom skaliranja platna jer se prema faktoru skaliranja veličina platna prilagođava veličini ekrana. Inače, bez korigiranja skale platna, pokazivač se za sve veći pomak sve više udaljava od objekta kojeg vuče.
- `rectTransform.anchoredPosition += eventData.delta / canvas.scaleFactor;`

IBeginDragHandler

- public void **OnBeginDrag**(EventSystems.PointerEventData **eventData**);
- Opis: Poziva se na početaku povlačenja. Potrebno je implementirati *IDragHandler*.
- Zadaća: Postavlja platno za roditelja objekta kako bi karta bila u prvom planu prilikom povlačenja po ekranu. Po zadanom, povlačeni objekt je taj koji hvata događaj, tako da je potrebno dopustiti koliziju da bi objekt ispod, objekt koji hvata mogao uhvatiti događaj ispuštanja. Potrebno je isključiti svojstvo *CanvasGroup* komponente.
- `canvasGroup.blocksRaycasts = false;`

IEndDragHandler

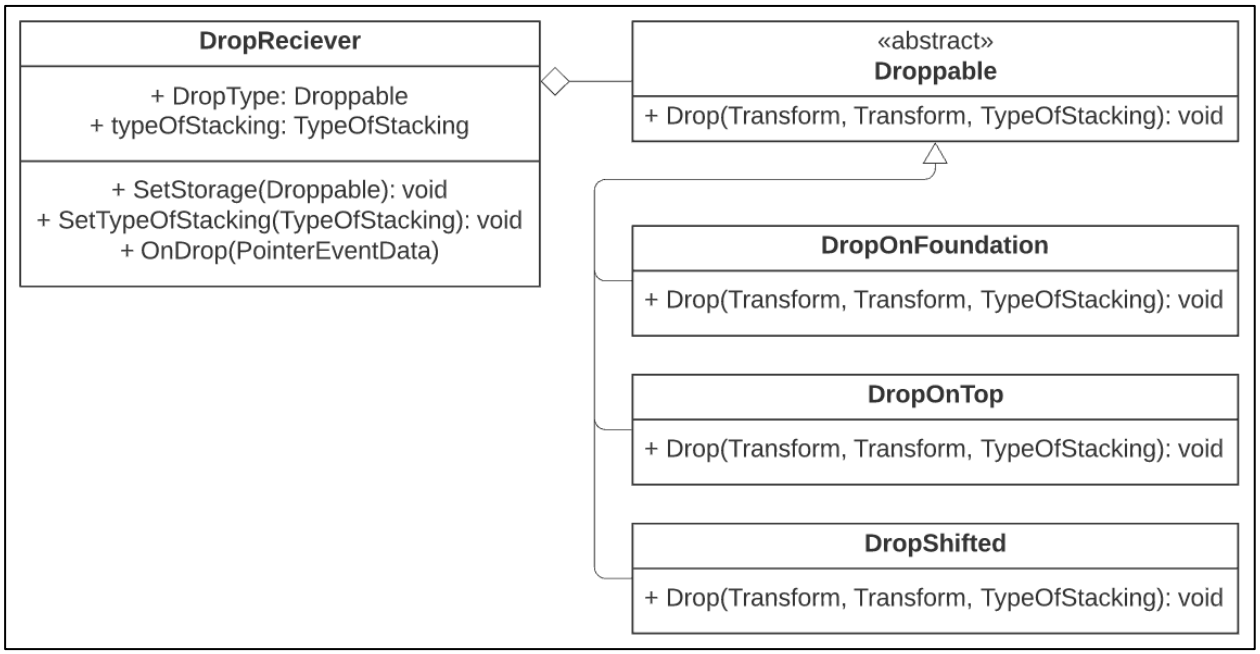
- public void **OnEndDrag**(EventSystems.PointerEventData **eventData**);
- Opis: Poziva se na završetku povlačenja. Potrebno je implementirati *IDragHandler*.

- Zadaća: Isključuje koliziju povlačenog objekta kako ne bi mogao prihvatiti objekt ako se ne nalazi na odgovarajućem mjestu. To znači da ako se karta nalazi negdje u nizu, a ne na samom vrhu niza, nije u mogućnosti prihvatiti kartu na sebe.

IDropHandler

- public void **OnDrag**(EventSystems.PointerEventData **eventData**);
- Opis: Poziva ga osnovni ulazni modul na objektu koji treba prihvatiti ispuštanje.
- Zadaća: Nalazi se u objektima za koje je potrebno da imaju mehanizam za prihvaćanje karte, služi za hvatanje događaja ispuštanja.

3.2.6. Primanje spuštene karte

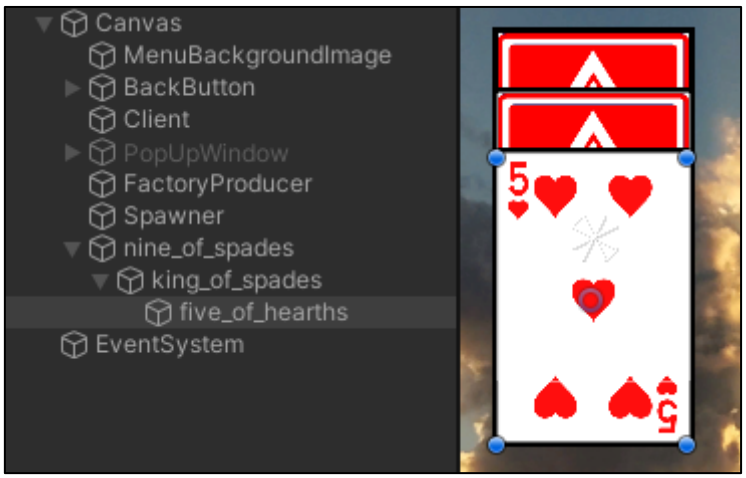


Slika 3.10. Strategija

Slika 3.9. prikazuje strukturu klasa za implementaciju oblikovnog obrasca strategija. Po ispuštanju objekta poziva se metoda u skripti *DropReciever*. Metoda prima kao argument podatke pokazivača jer se pokazivačem pomicao objekt po ekranu pa preko tog podatka se može dobiti referenca objekta koji je ispušten. *DropReciever* sadrži referencu apstraktnog tipa *Droppable* koji ujedinjuje različite strategije. Na strategije se delegira posao odlaganja objekta na prikladan način. Postoji više strategija jer prilikom odlaganja na donje temelje karte se slažu

malo pomaknute jedna na drugu kako bi se vidjela karta ispod. Tako se dobije još prostora za taktiziranje. Za slaganje na gornje temelje nije potrebno da karte ispod budu vidljive jer je pravilo da se slaže jedno odijelo karte po temelju, uzlaznim redoslijedom bez preskakanja. Ispuštanje na temelj je dodano kao zasebna strategija zbog dva različita redoslijeda slaganja i zato što karte trebale prekriti temelj, temelj treba biti vidljiv samo kada nema karti na sebi kako bi bilo vidljivo gdje je moguće odložiti kartu. Postignuto je da *DropReciever* bude neovisan o konkretnoj izabranoj strategiji, postalo je moguće efikasno izmijeniti funkciju izmjenom strategije, čak i tijekom izvođenja.

3.2.7. Hijerarhija slaganja



Slika 3.11. Hijerarhija slaganja

Hijerarhija objekata u Unity-u je strukture stabla, stoga su korištena gotova svojstva *Transform* komponente za postavljanje roditelja i djeteta (slika 3.11.). Konkretna strategija kada zaprimi ispušteni objekt ima dvije zadaće. Prvo se po pravilima igre provjerava može li karta biti ispuštena na to mjesto, ako može tada se objekt primatelj postavlja kao roditelj ispuštenom objektu. Temelj u tom slučaju je korjenski čvor i karte su njegova djeca. Za rješenje klondike verzije dovoljno je temelj sa kartama promatrati kao niz jer svaki čvor u tom stablu ima samo jedno djete. Niz kao dvostruko povezani popis gdje jedan element niza ima referancu na prethodni element kao roditelj i na slijedeći element kao dijete. Kada se objekt postavi kao dijete drugom objektu on se po zadanom nalazi u prvom planu, kao najgornji element prikazan na ekranu, tj. prekriva sve elemente u hijerarhiji koji su iznad njega. Karte, po pravilima igre nije moguće umetati u sredinu niza, već samo na vrh i zato na jednoj razini stabla stoji samo jedna

karta. Ta provjera je riješena preko *childCount* svojstva iz Transform komponente, to je broj djece trenutnog objekta. Uz pomoć *childCount* svojstva vrlo je lako utvrditi posljednju kartu u nizu, ako objekt nema djece on mora biti posljednji u nizu.

4. ZAKLJUČAK

Napredak u razvoju umjetne inteligencije za igranje pasijansa pospješuje utvrđivanje rješivosti igre, ali i dalje ne objašnjava što igru čini rješivom. Kako bi se za jedno dijeljenje dokazalo da je nerješivo potrebna je velika računalna moć. Kako bi se za dijeljenje dokazalo da je rješivo dovoljno je pronaći samo jedan put do rješenja. Pronaći rezultat za jednu igru pasijansa, u teoriji, moguće je pronaći u eksponencijalnom vremenu tako da se isprobaju sve mogućnosti. U praksi ipak još uvijek nije moguće, ali kombinaciju koraka do rješenja je lako moguće provjeriti u polinomijalnom vremenu pa prema tome pasijans pripada NP klasi kompleksnosti. To znači da do rješenja nije moguće doći u polinomijalnom vremenu. Zapravo, nitko još nije uspio dokazati da rješenje, ako bi se pronašla dovoljno napredna metoda rješavanja, nije moguće pronaći u polinomijalnom vremenu. Tako da, kada bi metoda efikasnog rješavanja takvih problema bila pronađena moglo bi se reći da su P i NP problemi istog skupa problema.

Unity je odlična platforma za započeti razvoj video igara. Jako je popularan jer je besplatan, uz što ide i velika količina besplatnih izvora za učenje, dodatno uz službenu dokumentaciju. Uz to, Unity kao takva velika platforma za razvoj igara ima svoju trgovinu imovinama (engl. *Assets Store*) sa hrpom gotovih 2D i 3D objekata, tekstura, animacija i zvukova, besplatnih ili uz plaćanje. Podržava veliki broj platformi i moguće je razvijati gotovo bilo koji žanr igara.

Aplikacija ovog završnog rada može se nadograditi tako da se dodaju: opcija izmjene slike pozadine, izmjena dizajna karata, animacije, ugođajna glazba, sustav bodovanja. Izgled sučelja igre jako puno doprinosi ugođaju igranja. Umjetnički stil igre najviše utječe na prvi dojam igrača. Dodavanjem sustava bodovanja igru bi se učinilu kompetitivnijom, igrači bi se mogli rangirati na bodovnim ljestvicama.

LITERATURA

[1] The Psychological Benefits of Playing Solitaire, MobilityWare, Irvine CA, 21.5.2021., https://www.mobilityware.com/post/the-psychological-benefits-of-playing-solitaire, 2.8.2021.

[2] H. Papadimitriou, Computational Complexity: A guide to the theory of NP-completeness, Addison-Wesley Publishing Company, Massachussetts, 1994.

[3] E. Demaine, Computational Complexity, MIT OpenCourseWare, 2011., https://www.youtube.com/watch?v=moPtwq_cVH8&t=2057s, 2.9.2021.

[4] G. S. Fishman, Monte Carlo Concepts, Algorithms and Application, Springer-Verlog, New York, 1996.

[5] X. Yan, P. Diaconis, P. Rusmevichientong, B. V. Roy, Solitaire: Man Versus Machine, Advances in Neural Infromation Processing Systems 17 (NIPS 2004), Vancouver, 2004.

[6] .L. Longpre, P. McKenzie, The Complexity of Solitaire, Mathematical Foundations of Computer Science 2007., 2007932973, str. 182-193, Prag, 2007.

[7] Build once, deploy anywhere: Unparalleled platform support, Unity Technologies, 2021, https://unity.com/features/multiplatform, 2.9.2021

[8] J. Gregory, Game Engine Architecture, A K Peters/CRC Press, 2009.

[9] D. Baron, Hands-On Game Development Patterns with Unity 2019, Pact Publishing Ltd., Birmingham, 2019.

[10] Lucidchart, Lucidchart Software Inc. 2021., https://www.lucidchart.com/, 16.9.2021.

[11] Unity Manual: Creating Gameplay, Unity Technologies, 2021., https://docs.unity3d.com/Manual/UnityManual.html, 2.8.2021

POPIS SLIKA

Slika 2.1. Klase kompleksnosti [2] 5
Slika 2.2. Vrste poteza [6] 8
Slika 3.1. Prikaz hijerarhije objekata u Unity-u [9] 13
Slika 3.2. Slika zaslona glavnog izbornika 15
Slika 3.3. Slika zaslona igre 16
Slika 3.4. *Prefab* i inspektor karte 17
Slika 3.5. *Prefab* i inspektor temelja 18
Slika 3.6. *Prefab* i inspektor talona 19
Slika 3.7. Apstraktna tvornica 20
Slika 3.8. Iterator 21
Slika 3.9. *Drag and drop* 22
Slika 3.10. Strategija 24
Slika 3.11. Hijerarhija slaganja 25

POPIS TABLICA

Tablica 2.1. Pokus rješivosti igre [5] 7

SAŽETAK

U ovom radu je predstavljena implementacija klondike solitaire igre u Unity Game Engine-u. U uvodu se nalazi kratki osvrt na povijest igre i utjecaj igre koji ima na igrača. Opisan je problem kompleksnosti pronalaska rješenja klondike igre. Dalje su objašnjena pravila igre i arhitektura Unity-a kako bi implementirane mehanike bile bolje pojašnjene. Navode se korišteni oblikovni obrasci i objašnjene su korištene funkcionalnosti Unity-a.

Keywords:

klondike pasijans, problem rješivosti pasijansa, Unity game engine

ABSTRACT

SOLITAIRE

In this paper, the implemenatation of klondike solitaire game in Unity Game Engine is presented. The introduction part provides a brief overview of the game and the games influence on the player. The following chapter descibes the problem of the complexity of finding a solution of klondike. In the following chapters are further explained rules of the game and Unity architecture in order to better explain implemented mechanics. There are listed design patterns and functionalities of Unity used in the project.

Keywords:

Klondike solitaire, solitaire solvability problem, Unity game engine

ŽIVOTOPIS

Tomislav Markovica rođen je u Bjelovaru 10. kolovoza 1998. Pohađao je Osnovnu školu Ivana Lackovića Croate u Kalinovcu. Nakon osnovne škole, 2013. godine upisuje Srednju strukovnu školu Đurđevac, smjer tehničar za računalstvo. Nakon završetka srednje škole, 2017. godine upisuje preddiplomski sveučilišni studij računarstva na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija Osijek.