UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Strahinja Milojević

OBRASCI ZA PROJEKTOVANJE VIDEO IGARA I NJIHOVA IMPLEMENTACIJA KORIŠĆENJEM OKRUŽENJA UNITY

master rad

Beograd, 2020.

Mentor:

doc. dr Aleksandar Kartelj
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

prof. dr Filip Marić
Univerzitet u Beogradu, Matematički fakultet

doc. dr Milan Banković
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Naslov master rada: Obrasci za projektovanje video igara i njihova implementacija korišćenjem okruženja Unity

Rezime: Obrasci za projektovanje su veoma značajni u razvoju softvera, pa tako i u razvoju video igara. Cilj rada je da se prikaže kako se neki opšti obrasci za projektovanje mogu primeniti u razvoju video igara, kao i pregled nekih obrazaca koji su specifični baš za video igre. Rad takođe sadrži i osnove okruženja Unity, kao i praktičnu implementaciju nekih obrazaca na primeru strateške igre iz žanra Odbrana tornjevima korišćenjem pogodnosti samog okruženja.

Ključne reči: obrasci za projektovanje, video igre, Unity, Singlton, Strategija, Dekorator, Model komponenta-objekat, Stanje, Fabrika, Prototip, Komanda, Kolekcija objekata, Posmatrač, Odbrana tornjevima, C#

Sadržaj

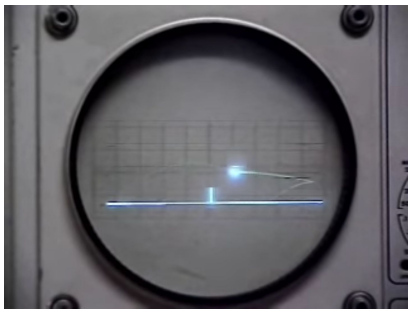
Glava 1

Uvod

U ovom radu biće predstavljen značaj primene obrazaca za projektovanje u procesu razvoja video igara, pregled najkorišćenijih obrazaca, kao i njihova praktična implementacija korišćenjem okruženja Unity. Pre svega toga, važno je istaći kako su se računarske igre razvijale kroz istoriju, koje sve tehnike programiranja igara postoje, kao i pregled osnovne arhitekture i mogućnosti okruženja Unity.

1.1 Istorijat razvoja računarskih igara

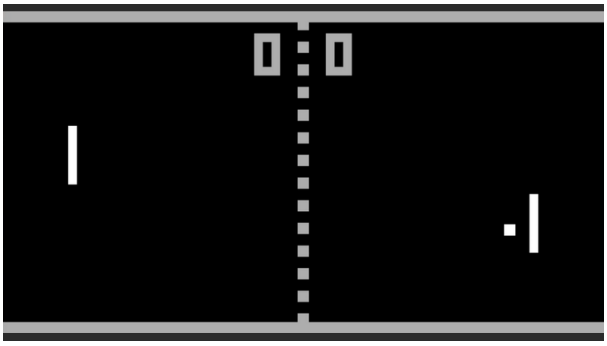
Prva video igra nastala je krajem pedesetih godina dvadesetog veka. Fizičar Vilijam Higinbotam, poznat i kao član tima koji je proizveo prvu nuklearnu bombu, napravio je 1958. igru koju je nazvao *Tenis za dvoje*. Izvršavala se na analognom računaru *Doner Model 30* a za prikaz je iskorišćen osciloskop - uređaj koji služi za prikazivanje promene električnog napona u nekom vremenskom periodu. Igra je zapravo bila simulacija tenisa za dva igrača koji su upravljali aluminijumskim kontrolerima nalik džojsticima.



Slika 1.1: Tenis za dvoje

Sledeći bitan korak u istoriji video igara jeste njihova komercijalizacija, odnosno nastanak *arkadnih igara*. Arkadnim igrama nazivamo one koje se igraju na automatima i dostupne su na javnim mestima. Godine 1972. nastaje kompanija *Atari* iz saradnje Nolana Bašnela i Teda Debnija. Prva njihova objavljena arkadna video igra bila je simulacija stonog tenisa koju su nazvali *Pong*. Taj naziv se zadržao za sve igrice ovog tipa do danas. U ovom periodu kreće i korišćenje mikroprocesora što utiče na fizičko smanjivanje računara, povećavanje performansi i pad cene.

Zahvaljujući razvitku visoko integrisanih mikročipova, cena računara dodatno pada, pa oni postaju pristupačni i običnim korisnicima, a ne samo institutima i kompanijama. Zahvaljujući tome, prve kućne konzole za igre nastaju sredinom sedamdesetih godina dvadesetog veka. Prvim kućnim konzolama za igre smatra se *Odisej* edicija, odnosno *Odisej 100* i *Odisej 200*. Na njima su korisnici mogli da igraju kopije igre Pong u svojim domovima. Igra je bila slična kao arkadna verzija, s tim što se upravljalo džojsticima umesto dugmićima na automatu.



Slika 1.2: Pong

Sa razvitkom programskih jezika visokog nivoa kao što su Basic i C, programiranje, pa i programiranje video igara, postaje pristupačnije i proširuje se krug ljudi koji ih kreiraju. Nastaju i prvi kućni računari opšte namene, a samim tim i prve igre za kućne računare. Krajem sedamdesetih i početkom osamdesetih godina kreće i fizička distribucija igara preko flopi disketa. Za ovaj period se vezuje i nastanak prve onlajn igre - *Lavirint* (*Maze Wars*). Ona se smatra i prvom igrom iz FPS žanra (First Person Shooter, odnosno pucačina iz prvog lica). Nastaju i prve 3D igre, a prvi predstavnik 3D igara jeste igra *Spasim*.

Devedesetih godina dvadesetog veka video igre prelaze sa rasterske na vektor-

sku i 3D grafiku. U ovom periodu nastaju i mnogi popularni žanrovi koji danas čine osnovnu klasifikaciju video igara. Razvijaju se konzole, ali i PC igre. Mnoge kompanije koje spadaju u najpoznatije proizvođače video igara današnjice nastale su devedesetih.

Sa razvitkom mobilnih telefona, krajem devedesetih i početkom dvehiljaditih kreće i razvoj igrica za mobilne telefone. U ovom periodu nastaju i takozvani modovi, odnosno pruža se prilika samim igračima da menjaju igru, to jest kreiraju svoje nivoe, mape i pravila. Popularizacijom interneta i povećavanjem brzine protoka podataka, masovno raste i popularnost onlajn igara.

S vremenom su računari postajali sve napredniji, pa se stvarala mogućnost razvoja složenijih video igara. Nastaju i specijalizovana okruženja za kreiranje igara koje olakšavaju njihovu izradu tako što sadrže već gotove složene sisteme koji su neophodni za ravoj današnjih igara, kao što je sistem za renderovanje grafike, ali i gotove implementacije nekih čestih problema. Najpopularnija okruženja za kreiranje modernih igara današnjice jesu *Unity* i *Unreal Engine*, i njih koristi veliki broj kompanija koje se bave proizvodnjom video igara. Današnje video igre mogu biti izuzetno kompleksne, a samim tim njihov razvoj i održavanje moraju biti dobro isplanirani, jer, nepažljivim pristupom, razvoj igre i bilo kakva izmena u kôdu mogu postati prava noćna mora. Igre često dele mnoge zajedničke probleme, pa je dobro koristiti univerzalna rešenja koja rešavaju česte probleme i čine kôd efikasnijim i lepšim. Ta rešenja su najčešće obrasci za projektovanje video igara.

1.2 Pregled tehnika programiranja igara

Kada se govori o tehnikama programiranja, najčešće se govori o programskim paradigmama. Postoje četiri osnovne paradigme:

1. imperativna (proceduralna)
2. objektno-orijentisana
3. funkcionalna
4. logička

Imperativna paradigma se zasniva na nizu naredbi koje menjaju stanje programa, uz izdvajanje naredbi u funkcije i procedure. Vezuje se za tradicionalno programiranje, ali se koristi i danas.

Objektno-orijentisana paradigma je, po mnogim izvorima dostupnim na internetu, najpopularnija paradigma današnjice. Zasniva se na simulaciji spoljašnjeg sveta pomoću objekata koji međusobno interaguju. Fundamentalni pojmovi koji se vezuju za ovu paradigmu jesu klasa, objekat, enkapsulacija, nasleđivanje i polimorfizam. *Klasa* predstavlja šablon za kreiranje konkretnih objekata, dok su *objekti* integralne celine koje se sastoje od stanja i ponašanja. *Nasleđivanje* predstavlja koncept kreiranja novih klasa na osnovu već postojećih. Klase mogu da budu osnovne i izvedene. Izvedena klasa je klasa koja nasleđuje drugu klasu, dok se u suprotnom ona naziva osnovna. Klasa koju druge klase nasleđuju naziva se bazna klasa. Ovaj koncept je izuzetno koristan kada imamo više sličnih objekata koji dele neka svojstva. U tom slučaju najopštija *polja* (podatke koje klasa sadrži) i *metode* (funkcije koje klasa sadrži) izdvajamo u baznu klasu, a sve ostale nasleđuju baznu i dodaju sebi svojstvena polja i metode koji su specifični za nju. Ovim pristupom smanjujemo bespotrebno ponavljanje kôda i pružamo mogućnost izvedenim klasama da koriste elemente iz bazne klase. Metod bazne klase može da ima različito ponašanje u izvedenim klasama. Koncept promene ponašanja metoda naziva se *polimorfizam*. Može biti *ad-hok* (višeznačnost imena funkcija i operatora, na primer + može biti sabiranje brojeva ili konkatenacija stringova u nekim jezicima), *hijerarhijski* (menjanje metoda natklase u njenim potklasama), *parametarski* (takozvano generičko programiranje - metod je napisan bez specifičnog tipa) i *implicitni* (uopštenje parametarskog - dozvoljava pisanje kôda bez navođenja tipova). *Enkapsulacija* predstavlja učaurivanje podataka i funkcija u objekte.

Funkcionalna paradigma se zasniva na evaluaciji funkcija i često koristi rekurziju kao način kontrole toka. Najpribližnija je matematičkom izračunavanju i ima teorijsku podlogu u lambda računu. Karakteriše je čist kôd bez sporednih efekata izračunavanja.

Logička paradigma se dosta razlikuje od ostalih jer je deklarativna, odnosno programira se tako što se precizno opisuje samo rešenje, dok se kod ostalih paradigmi (koje možemo nazvati opšte proceduralnim) opisuje sam postupak kojim se

dolazi do rešenja. Zasnovana je na zadavanju aksioma i pravila izvođenja, a zatim se kreiraju upiti koji koriste zadate aksiome i pravila da bi došli do rešenja. Ima prilično ograničenu upotrebu i nekada se koristila u veštačkoj inteligenciji.

Programiranje video igara bi bilo dosta teško, a možda i nemoguće, ako bi koristili logičku paradigmu, jer je previše ograničena i ne modeluje dobro probleme koji se tu javljaju. Što se tiče funkcionalne, postoje video igre kreirane u funkcionalnim jezicima, ali izražavanje svega korišćenjem funkcija ne pogoduje video igrama, pa samim tim nisu previše razvijane biblioteke pogodne za rad na igrama. Što se tiče imperativne paradigme, svakako je moguće kreirati igre u ovom stilu, ali nije najpogodnije za modelovanje kompleksnijih problema koji se javljaju u današnjim video igrama, mada se može iskoristiti za neke jednostavnije primere. Preostala je objektno-orijentisana paradigma koja je s razlogom najkorišćenija danas. Ako malo bolje razmislimo, većina igara se zasniva na preslikavanju nekih realnih objekata u samu igru. Ti objekti imaju određene karakteristike i akcije koje mogu izvršavati, pa je objekno-orijentisan pristup prirodan i najkorišćeniji u razvoju video igara.

Osim osnovnih, postoji i mnogo potparadigmi koje su često zasnovane na nekoj od osnovne četiri, ali unose neku novu dimenziju programiranja. Neke od njih se koriste u video igrama, a to su komponentna i vizuelna.

Komponentna paradigma je potparadigma koja se zasniva na sklapanju softvera od već gotovih komponenti. Smatra se da je potparadigma objektno-orijentisane. Kada se primenjuje komponentno programiranje, najbitnije je razvojno okruženje koje pruža tu mogućnost, dok same komponente mogu biti u bilo kom jeziku. Kada govorimo o razvojnim okruženjima za kreiranje video igara, u ovu paradigmu možemo svrstati Unity jer sve skripte kojima definišemo ponašanje objekata zapravo predstavljaju komponente nekog objekta. Postoje brojne prednosti ovog pristupa kao što su ponovna iskoristivost komponenti, jednostavno dodavanje i uklanjanje funkcionalnosti kroz dodavanje i uklanjanje komponenti, fleksibilnost kreiranja objekata itd. Najveća mana ovog pristupa jeste međusobna komunikacija između različitih komponenti, pa se često kreiraju globalni kontroleri, što je u redu za neke slučajeve, ali treba izbegavati njihovu masovnu upotrebu.

Vizuelna paradigma se takođe može smatrati potparadigmom OOP. Jedin-

stvena je po tome što nema eksplicitnog programiranja kroz kôd, već se to radi korišćenjem grafičkog korisničkog interfejsa okruženja. Ova potparadigma je najzastupljenija u školama programiranja za najmlađi uzrast jer je mnogo jednostavnije kreirati programe prevlačenjem gotovih metoda i upravljačkih jedinica (poput naredbe grananja), pa je zgodna za apsolutne početnike. Međutim, postoje i moćnija okruženja koja u potpunosti pripadaju vizuelnoj paradigmi, a sadrže veliki broj predefinisanih metoda i kontrola koje korisnik može da koristi. Predstavnik ove paradigme u svetu video igara jeste okruženje *GameMaker*. Za njegovo korišćenje nije neophodno poznavanje programskih jezika, a opet pruža razne mogućnosti za kreiranje video igara različitih žanrova. Naravno, zbog ograničenih mogućnosti ovog okruženja, ne koristi se u kompanijama koje se bave video igrama, ali je pogodno za početnike.

1.3 Okruženje Unity

Unity je razvojno okruženje kreirano za potrebe razvoja video igara. Podržava mnoge platforme među kojima su PC, Android, iOS, WebGL, kao i konzole za igre, kao što su Play Station i Xbox. Dok god se ne koriste specifičnosti same platforme, isti projekat se može kompilirati za sve podržane platforme, što ga čini izuzetno fleksibilnim. Najčešće se koristi uz programski jezik C#. Ima mnoge ugrađene funkcionalnosti, kao što su renderovanje grafike i sistem za fiziku. Može se koristiti i za virtualnu stvarnost, a novije verzije podržavaju i naočare za augmentativnu stvarnost poput Microsoft Hololensa.

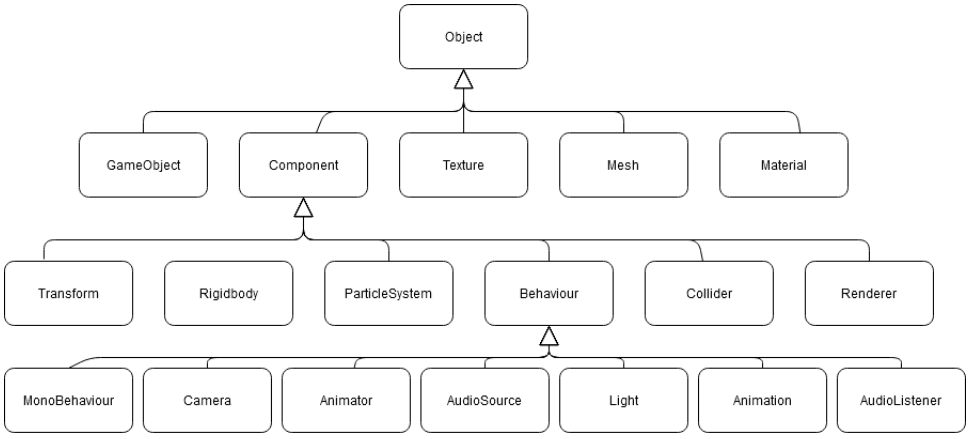
Osnovni pojmovi

Prvi osnovni pojam koji Unity koristi jesu scene. **Scena** sadrži okruženje i menije. Može se posmatrati i kao jedan nivo u igri. Scenu možemo posmatrati kao drvoliku strukturu elemenata. Kreiranjem nove scene, u koreni čvor (*root node*) automatski se dodaju kamera i svetlo. Sceni dodajemo **objekte igre** (*Game Object*), ili kraće - objekte. Objekti imaju svoje **komponente**. Komponente mogu biti razne stvari. Skoro svaki objekat ima komponentu **transform** koja predstavlja poziciju, rotaciju i skaliranje objekta u koordinatnom sistemu. Neke od mogućih komponenti su animator (koji kontroliše animaciju objekta), kolajder (2D ili 3D geometrija oko objekta koja koristi svojstva fizike), sprite renderer (komponenta

zadužena za renderovanje slike) itd. Sledeći bitan pojam jeste **skripta**. Skripta je u stvari samo jedna od komponenti objekta. Skriptom, odnosno kôdom iz skripte, definišemo ponašanje datog objekta. Unity je zasnovan na komponentnoj potparadigmi, što je prilično zgodno za razvoj video igara. Sve skripte u okruženju Unity su komponente objekata kojima pripadaju. Ovaj stil se često koristi i u okruženjima koja nisu komponentna jer su prirodna za razvoj video igara i postiže se korišćenjem obrasca **Model komponenta - objekat** koji implicitno koristimo kroz Unity a o kome će biti više reči kasnije. Sem objekta, kao pojam uvodimo i **prifab** (engl. Prefab) koji predstavlja obrazac objekta igre koji se može instancirati proizvoljno mnogo puta. Ovo je još jedan primer implicitnog korišćenja obrazaca za projektovanje video igara kroz Unity, što značajno doprinosi bržem i efikasnijem razvoju, a utiče i na jednostavnije održavanje kôda i dodavanje novih funkcionalnosti.

Hijerarhija klasa

Okruženje Unity ima svoju internu organizaciju klasa koje koristi. Neke od njih korisnici mogu direktno da naslede i prošire u svojim implementacijama, neke se koriste samo kroz grafički korisnički interfejs, a neke samo predstavljaju osnovne gradivne jedinice i ne bi trebalo koristiti ih direktno. Ove klase naravno su međusobno zavisne i postoji određena hijerarhija među njima. Hijerarhijska organizacija klasa koje koristi okruženje Unity sastoji se iz četiri sloja. Kako slojevi izgledaju i šta sadrže, može se videti na sledećem dijagramu:



Slika 1.3: Hijerarhija klasa okruženja Unity

U osnovi se nalazi klasa *Object*. Ona predstavlja baznu klasu za sve objekte koje

Unity može da koristi. Nije napravljena s namerom da se eksplicitno primenjuje svuda u kôdu, mada se i dalje može naći ponegde, na primer kao povratna vrednost predefinisanog metoda za učitavanje nekog spoljnog resursa iz odgovarajućeg foldera, odnosno *Resources.LoadAll()*. Ova klasa se jedina nalazi u inicijalnom, **prvom sloju** hijerarhije.

Drugi sloj se sastoji iz sledećih pet klasa:

1. *GameObject*
2. *Component*
3. *Texture*
4. *Mesh*
5. *Material*

Klasa *GameObject* je bazna klasa za sve objekte koji se nalaze na sceni. Sadrži nekoliko polja među kojima su *activeInHierarchy* (označava da li je objekat aktivan, odnosno vidljiv i slobodan za interakciju sa ostalima), *layer* (sloj koji određuje redosled renderovanja, najčešće se koristi kada želimo da neki objekat bude prikazan iznad drugog u 2D igri), *tag* (oznaka objekta igre koja može biti korisna za pravljenje određene klasifikacije) itd. Sem toga sadrži i metode za dodavanje i dohvatanje komponenti, traženje podobjekata, uništavanje objekta, upoređivanje tagova i mnoge druge.

Klasa *Component* predstavlja baznu klasu za sve komponente koje se mogu prikačiti za objekat igre. Korisnici nikada ne kreiraju objekte ove klase direktno.

Preostale klase iz ovog sloja jesu *Texture*, *Mesh* i *Material* koriste se za grafiku.

Treći sloj čine klase koje nasleđuju klasu *Component* i mogu se naći kao komponente objekta igre. To su:

1. *Transform*
2. *Rigidbody*

3. *ParticleSystem*
4. *Behaviour*
5. *Collider*
6. *Renderer*

Klasa *Transform* čuva podatke o poziciji, rotaciji i skaliranju objekta i služi za manipulaciju istih. Svaki objekat na sceni je ima. *Transform* može imati i roditeljski objekat, odnosno *parent*, što omogućava postavljanje pozicije, rotacije i skaliranja hijerarhijski, odnosno relativno u odnosu na roditeljski objekat. Takođe je moguća i iteracija kroz decu objekte.

Rigidbody kontroliše objekat kroz simulaciju fizike. Samim tim što objektu dodamo ovu komponentu, dopuštamo sistemu za fiziku da ga detektuje i koristi. Na primer, gravitacija će biti primenjena na sve objekte koji imaju ovu komponentu, čak i bez dodavanja bilo kakvog kôda. Često se koristi u kombinaciji sa *Collider* klasom za detekciju kolizije. Klasa *Collider* je zapravo bazna za sve tipove kolajdera, kao što su sferni kolajder, kvadar kolajder itd. Najčešće ide u paru sa *Rigidbody*.

Klasa *ParticleSystem* služi za kontrolu sistema za čestice. U njoj se može podešavati oblik, frekvencija, broj kreiranih čestica i mnogi drugi parametri.

Renderer je klasa koja pruža mogućnost renderovanja i ona je zaslužna za to da se objekti zaista pojave na ekranu. Sem za renderovanje objekata, koristi se i za renderovanje sistema čestica ili mreža, odnosno *Mesh* objekata.

Poslednja klasa u ovom sloju jeste *Behaviour*. Koristi se za komponente koje se mogu uključiti ili isključiti. Predstavlja baznu klasu za sve klase četvrtog sloja.

Četvrti sloj se sastoji iz sledećih klasa:

1. *AudioListener*
2. *AudioSource*
3. *Animator*

4. *Animation*
5. *Camera*
6. *Light*
7. *MonoBehaviour*

Klasa *AudioListener* snima zvukove oko sebe i reprodukuje ih kroz zvučnike ili slušalice igrača. Moguće je imati samo jedan *AudioListener* na sceni. Uz ovu klasu ide i *AudioSource* koju korisnik dodaje kao komponentu objektu igre koji treba da proizvede zvukove. Da bi ih proizvela, neophodno je da postoji i *AudioListener* na sceni. Klasa *AudioSource* sadrži puno metoda i promenljivih kojima se kontrolišu razni parametri zvuka, kao što su izvor, jačina, visina, ponavljanje, i mnogi drugi.

Sledeće dve klase koje idu u paru jesu *Animator* i *Animation*. *Animator* predstavlja interfejs za kontrolu sistema za animaciju, dok se *Animation* koristi za puštanje same animacije.

Klasa *Camera* služi za upravljanje kamerom, odnosno njena najosnovnija svrha jeste prikazivanje scene na odgovarajući način. Uz nju ide i klasa *Light* za kontrolisanje osvetljenja na sceni.

Poslednja klasa u ovom sloju je *MonoBehaviour* i najznačajnija je za samog programera jer predstavlja baznu klasu za sve skripte. Sadrži neke od ključnih metoda neophodnih za kreiranje igre kao što su *Start()* (definiše ponašanje na početku, odnosno u trenutku kada skripta postane aktivna, što je najčešće prilikom učitavanja scene), *Update()* (definiše ponašanje objekta sa ovom skriptom u svakom frejmu), *OnDisable()* (definiše ponašanje pre nego što skripta postane neaktivna) itd.

Glava 2

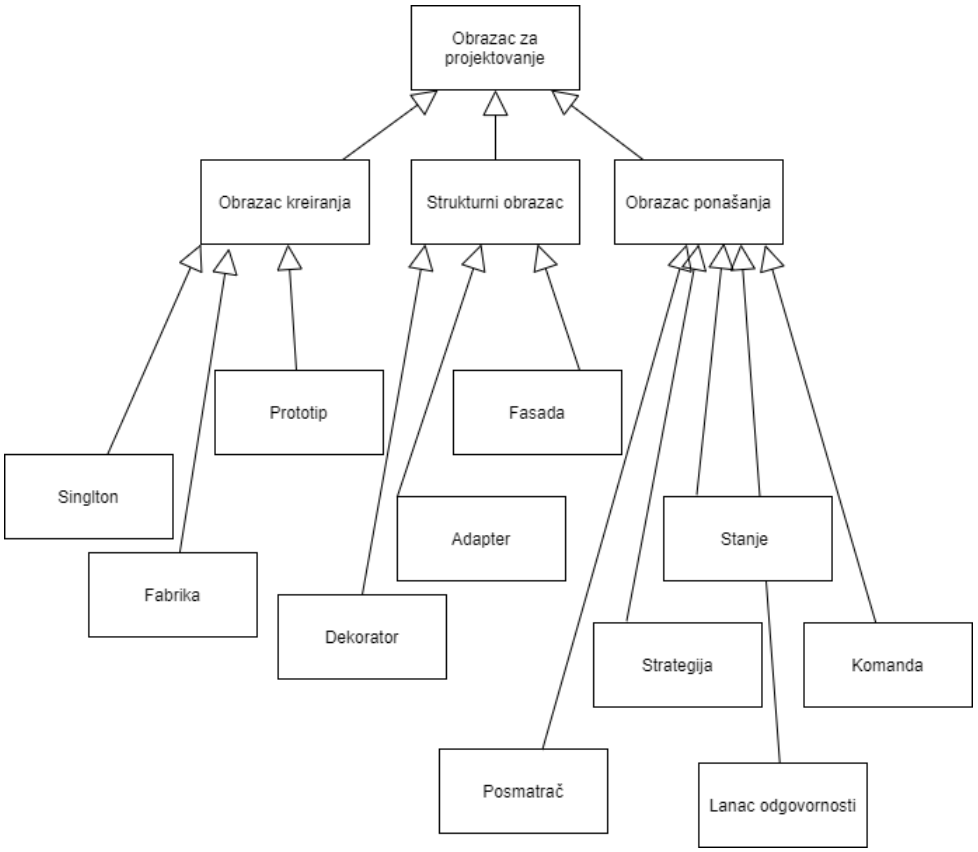
Obrasci za projektovanje

Kao što je ranije naglašeno, najkorišćenija paradigma današnjice jeste objektno orijentisana paradigma, odnosno OOP. Aplikacije napisane u istom stilu, odnosno paradigmi, imaju puno zajedničkih svojstava. Osim što dele osnovne koncepte, često dele i zajedničke probleme. Ako se problem ponovi u istom obliku, može se izdvojiti u posebni modul, klasu ili biblioteku, i zatim iznova koristiti na svim mestima gde se ponovo javi. Međutim, ako se problem ne ponavlja u identičnom obliku već postoje određene varijacije, u tom slučaju nije moguće iskoristiti gotovo rešenje izdvojeno u modul ili biblioteku. Za probleme koje su konceptualno ekvivalentni ali nedovoljno slični da se mogu implementirati istom kôdom primenjujemo obrasce za projektovanje.

Obrazac za projektovanje predstavlja ponovno upotrebljivo rešenje za neki problem koji se često javlja prilikom razvoja softvera. On nije gotov dizajn koji se direktno može pretvoriti u kôd, već šablon namenjen za rešavanje nekog opštijeg problema u praksi. Obrasci za projektovanje u objektno orijentisanom programiranju su obično opisi objekata i klasa koji međusobno komuniciraju, bez određivanja konkretnih klasa i objekata.

Obrasci za projektovanje se mogu podeliti u tri grupe:

1. obrasci kreiranja
2. strukturni obrasci
3. obrasci ponašanja



Slika 2.1: Neki obrasci za projektovanje

Obrasci kreiranja predstavljaju podvrstu obrazaca za projektovanje i bave se mehanizmima kreiranja objekata. Često osnovni način kreiranja objekata nije dovoljno pogodan za neke situacije i može da izazove probleme u samom dizajnu, ili da dovede do povećane kompleksnosti. U tim situacijama treba razmotriti korišćenje odgovarajućeg obrasca kreiranja koji rešava zadati problem na način pogodan određenoj situaciji i nudi veću kontrolu prilikom kreiranja objekata. Neki od obrazaca koji pripadaju ovoj grupi jesu Singlton, Prototip, Fabrika itd.

Strukturni obrasci su podvrsta obrazaca za projektovanje čiji je cilj da olakšaju dizajn tako što identifikuju načine ostvarivanja veza između različitih entiteta. Neki od strukturnih obrazaca jesu Dekorator, Fasada, Adapter itd.

Obrasci ponašanja čine podvrstu obrazaca za projektovanje koja se bavi identifikovanjem čestih problema koji se javljaju u komunikaciji između objekata i načinima za njihovo rešavanje. Koriste se kada nam je neophodna fleksibilnost u sprovo-

đenju komunikacije među objektima. Neki od obrazaca ponašanja jesu Posmatrač, Strategija, Stanje, Komanda, Lanac odgovornosti itd.

Video igre imaju mnogo sličnosti među sobom. Neki koncepti koji su se javili još u najranijim igrama zadržani su i do danas, kao što su nivoi, životi, neprijatelji itd. Takođe, ogromnu većinu igara možemo svrstati u neki od standardnih žanrova koji opet dele svoje zajedničke probleme. Za razvoj video igara se najčešće koristi objektno-orijentisana paradigma (ili neka od njenih potparadigmi) pa se često dešava da se neki opšti problemi koji se sreću u objektno orijentisanom programiranju jave i u programiranju video igara. Oni možda na prvi pogled nisu očigledni, ali njihovom apstrakcijom možemo doći do njihove logičke ekvivalentnosti. Obrasci za projektovanje nude rešenja za neke od tih problema. Neki od obrazaca su direktno nasleđeni iz drugih oblasti, neki su prilagođeni video igrama, a neki su baš specifični za video igre. U nastavku će biti obrađeni neki od najkorišćenijih obrazaca za projektovanje koji se primenjuju u video igrama, kao i česti problemi sa kojima se susreću programeri prilikom razvoja, a koji se mogu rešiti ovim obrascima.

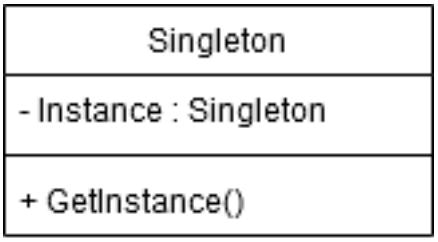
2.1 Singlton

Kada pričamo o programiranju video igara, skoro uvek pričamo o objektno orijentisanom programiranju, odnosno paradigmi. Jedno od glavnih svojstava objektno orijentisanog programiranja jeste sakrivanje podataka, odnosno sprečavanje neželjenog pristupa poljima i metodama klase. Kao što znamo, glavni modifikatori kontrole pristupa su *public*, *private* i *protected*. Modifikator *public* omogućava vidljivost iz svih klasa, private označava vidljivost samo u datoj klasi, dok protected označava vidljivost u datoj klasi i svim njenim potklasama. Još jedna od ključnih reči koje su bitne za ovaj obrazac jeste modifikator *static*. U programskom jeziku C# on se može primeniti na nekoliko stvari. Ako se primeni na klasu, znači da je klasa statička, odnosno da je ona sama jedna i jedina instanca te klase. Ako se primeni na metod, znači da se metod poziva nad samom klasom, a ne nad instancom te klase, čime se obezbeđuje ista implementacija u svim instancama te klase, naravno, nad podacima koje ta instanca trenutno sadrži. Ako se primeni na polje odnosno promenljivu, možemo reći da se radi o globalnoj promenljivoj - instance klase koja ima statičko polje neće same sadržati to polje, već će sve deliti jedinstveno polje u memoriji.

Korišćenje globalnih promenljivih ima dosta mana. Neke od njih su: teško praćenje toka programa jer se promenljiva mogla promeniti od strane bilo kog dela kôda, teže razumevanje samog kôda, dodatne provere da li promenljiva sadrži baš ono što nam je potrebno itd. Naravno, to ne znači da ih nikada i ni po koju cenu ne treba koristiti. Nekad korišćenje globalnih promenljivih pojednostavljuje posao ako je neophodno pristupiti promenljivoj iz bilo kog dela kôda. Ovo je posebno korisno u komponentnom programiranju čiji je glavni problem međusobna komunikacija između različitih komponenti. Sada kada smo se podsetili osnovnih stvari, možemo da pređemo na sam obrazac. Dakle, šta predstavlja Singlton? Obrazac Singlton predstavlja klasu koja ima samo jednu instancu, pa se time obezbeđuje njena jedinstvenost. Spada u obrasce kreiranja. Primer osnovne implementacije ovog obrasca može se naći u sledećem kôdu:

```
using System;

namespace Singleton
{
    class Singleton
    {
        private Singleton() { }

        private static Singleton _instance;

        public static Singleton GetInstance()
        {
            if (_instance == null)
            {
                _instance = new Singleton();
            }
            return _instance;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Singleton s1 = Singleton.GetInstance();
            Singleton s2 = Singleton.GetInstance();

            if (s1 == s2)
            {
                Console.WriteLine("Both variables contain the same
    instance.");
            }
            else
            {
                Console.WriteLine("Variables contain different
    instances.");
            }
    }
}
```

U prethodom jednostavnom primeru implementacije obrasca Singlton možemo videti da klasa Singlton sadrži statičko polje *_instance* i metod *GetInstance* koji vraća tu instancu. Ako trenutno ne postoji nijedna instanca klase, metod će je kreirati, a u slučaju da instanca postoji, vratiće postojeću. Klasa sadrži i privatni konstruktor kako bi onemogućili direktno instaniranje same klase bez korišćenja metoda *GetInstance*. U *Main* metodi su kreirane dve promenljive korišćenjem metoda *GetInstance* koje sadrže istu instancu, zahvaljujući obrascu Singlton.



Slika 2.2: Dijagram klase obrasca Singlton

Ovaj obrazac se veoma često koristi u kombinaciji sa okruženjem Unity jer omogućava pristup iz bilo kog dela kôda, što je generalno problematično u komponentnom pristupu. Međutim, često se koristi bez potrebe i zato treba biti izuzetno pažljiv u donošenju odluke da li je upotreba singltona opravdana. Česta greška koja se pravi jeste kreiranje takozvanih upravljačkih klasa za skoro sve u igri koja se kreira. Ovo samo dodatno komplikuje stvari bez potrebe i čini naš kôd nečitljivim i teško je izmeniti ga kasnije. Međutim, u određenim situacijama ga ima smisla koristiti i predstavlja najlakše rešenje.

2.2 Strategija

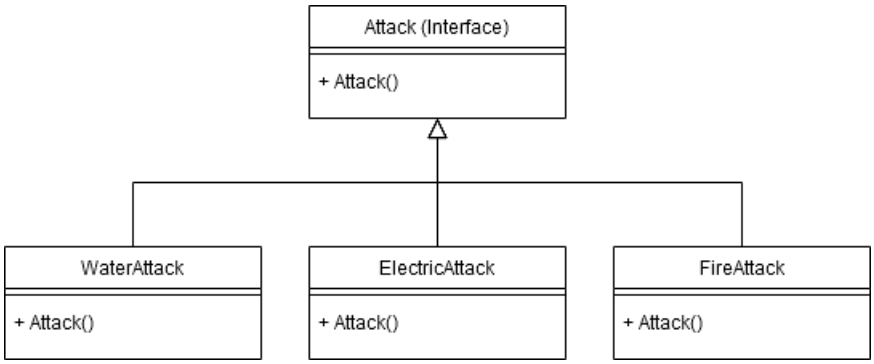
Svrha ovog obrasca za projektovanje jeste grupisanje određenih ponašanja. Ovakva grupisana ponašanja imaju interfejs preko kog se kontrolišu. Pogledajmo najpre primer implementacije ovog obrasca:

```
using System;
using System.Collections.Generic;

namespace Strategy
{
    class Context
    {

        private IStrategy _strategy;

        public Context() {}

        public Context(IStrategy strategy)
        {
            this._strategy = strategy;
        }

        public void SetStrategy(IStrategy strategy)
        {
            this._strategy = strategy;
        }

        public void DoLogic()
        {
            Console.WriteLine("Context: Sorting data using the strategy");
            var result = this._strategy.Execute(new List<string> { "a", "b", "c", "d", "e" });

            string resultStr = string.Empty;
            foreach (var element in result as List<string>)
            {
                resultStr += element + ",";
            }

            Console.WriteLine(resultStr);
        }
    }

    public interface IStrategy
    {
```

```
40          object Execute(object data);
41      }
42
43      class ConcreteStrategyA : IStrategy
44      {
45          public object Execute(object data)
46          {
47              var list = data as List<string>;
48              list.Sort();
49
50              return list;
51          }
52      }
53
54      class ConcreteStrategyB : IStrategy
55      {
56          public object Execute(object data)
57          {
58              var list = data as List<string>;
59              list.Sort();
60              list.Reverse();
61
62              return list;
63          }
64      }
65
66      class Program
67      {
68          static void Main(string[] args)
69          {
70              var context = new Context();
71
72              Console.WriteLine("Client: Strategy is set to normal
    sorting.");
73              context.SetStrategy(new ConcreteStrategyA());
74              context.DoLogic();
75
76              Console.WriteLine();
77
78              Console.WriteLine("Client: Strategy is set to reverse
    sorting.");
79              context.SetStrategy(new ConcreteStrategyB());
```

```
80                context.DoLogic();
81            }
82        }
83    }
```

U prethodnoj implementaciji vidimo interfejs *IStrategy* koji sadrži metod *Execute*. Taj metod odlučuje koji algoritam, odnosno strategija, će se primeniti u našem konkretnom slučaju. Ovde dakle vršimo enkapsulaciju različitih ponašanja i time stvaramo fleksibilnost. Konkretne strategije (u ovom slučaju *ConcreteStrategyA* i *ConcreteStrategyB*) implementiraju ovaj interfejs i definišu svoj metod *Execute*. U primeru vidimo i klasu *Context* koja sadrži polje *_strategy* koje postavljamo iz *Main* metode kako bi promenili strategiju koju želimo da iskoristimo u metodi *DoLogic*.

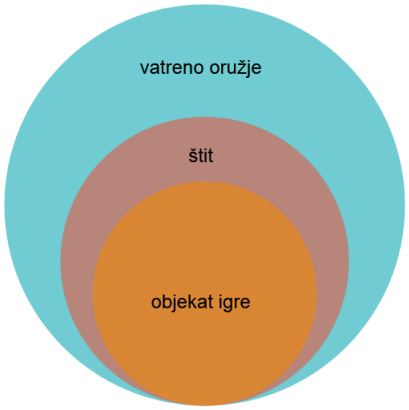


Slika 2.3: Dijagram klase obrasca Strategija

Ovaj obrazac je, kao i prethodni, univerzalan i nije ograničen na razvoj video igara, ali ima dosta smisla koristiti ga u ovoj oblasti. Na primer, ako imamo igru u kojoj karakter može da napadne vatrom, vodom i strujom, dobro je grupisati ova ponašanja korišćenjem strategija. Metod *Execute* bi u ovom slučaju bio napad, a konkretno ponašanje svakog od ova tri napada bi bilo grupisano i izvršavalo bi se istim metodom, u zavisnosti od trenutnih parametara. Same komponente možemo dodavati, brisati ili menjati dinamički pa nam ovaj obrazac pruža nephodnu fleksibilnost.

2.3 Dekorator

Obrazac za projektovanje Dekorator se koristi kada je potrebno dinamički dodati ponašanje objektu. Spada u strukturne obrasce. Grafički se ovaj obrazac može predstaviti ovako:



Slika 2.4: Dekorator

Možemo posmatrati dekorator kao programsku verziju ruske lutke koja sadrži svoje manje verzije unutra. Najmanja lutka će uvek biti objekat sa osnovnim funkcijama. Dekorator klase služe za raslojavanje objekta i dinamičko dodavanje još ponašanja na vrh objekta - najveću lutku. Pogledajmo primer implementacije:

```
using System;

namespace Decorator
{

    public abstract class Component
    {
        public abstract string Operation();
    }

    class ConcreteComponent : Component
    {
        public override string Operation()
        {
            return "ConcreteComponent";
        }
    }

    abstract class Decorator : Component
    {
        protected Component _component;

        public Decorator(Component component)
```

```
24          {
25              this._component = component;
26          }
27
28          public void SetComponent(Component component)
29          {
30              this._component = component;
31          }
32
33          public override string Operation()
34          {
35              if (this._component != null)
36              {
37                  return this._component.Operation();
38              }
39              else
40              {
41                  return string.Empty;
42              }
43          }
44      }
45
46      class ConcreteDecoratorA : Decorator
47      {
48          public ConcreteDecoratorA(Component comp) : base(comp)
49          {
50          }
51
52          public override string Operation()
53          {
54              return $"ConcreteDecoratorA({base.Operation()})";
55          }
56      }
57
58      class ConcreteDecoratorB : Decorator
59      {
60          public ConcreteDecoratorB(Component comp) : base(comp)
61          {
62          }
63
64          public override string Operation()
65          {
```

```
66              return $"ConcreteDecoratorB({base.Operation()})";
67          }
68      }
69
70      public class Client
71      {
72          public void ClientCode(Component component)
73          {
74              Console.WriteLine("RESULT: " + component.Operation());
75          }
76      }
77
78      class Program
79      {
80          static void Main(string[] args)
81          {
82              Client client = new Client();
83
84              var simple = new ConcreteComponent();
85              Console.WriteLine("Simple component:");
86              client.ClientCode(simple);
87              Console.WriteLine();
88
89              ConcreteDecoratorA decorator1 = new ConcreteDecoratorA(
    simple);
90              ConcreteDecoratorB decorator2 = new ConcreteDecoratorB(
    decorator1);
91              Console.WriteLine("Decorated component:");
92              client.ClientCode(decorator2);
93          }
94      }
95  }
```

U prethodnom kôdu imamo apstraktnu klasu *Component* i njen apstraktni metod *Operation*. Apstraktna klasa *Decorator* nasleđuju klasu *Component*. Sadrži polje *_component* koje se može postaviti, i metod *Operation* koji delegira posao unutrašnjem sloju, odnosno klasi *Component*. Konkretni dekoratori (u ovom slučaju *ConcreteDecoratorA* i *ConcreteDecoratorB* predstavljaju novi spoljni sloj objekta koji dodaje određene funkcionalnosti. U pređašnjem primeru, konkretni dekorator A dodaje tekst ConcreteOperatorA(...) i delegira posao unutrašnjim slojevima. Slično je i sa konkretnim dekoratorom B. U *Main* metodi imamo promenljivu *simple* koja

predstavlja osnovnu komponentu. Dekorator A je novi sloj osnovne komponente, dok je dekorator B novi sloj dekoratora A. Ovim vidimo da dekoratore možemo i da nadovezujemo jedan na drugi a ne samo na određene osnovne komponente. Time stvaramo kompleksnije slojevite objekte gde svaki sloj, odnosno dekorator, odradi svoj deo posla i delegira dalje ostalim slojevima. Rezulati ispisa prethodnih primera biće redom *ConcreteComponent* (kada imamo samo osnovnu komponentu) i *ConcreteDecoratorB(ConcreteDecoratorA(ConcreteComponent))* (kada osnovnoj komponenti dodamo još dva sloja, odnosno dekoratora).

U svetu video igara, ako imamo objekat sa svojim osnovnim funkcijama i želimo da mu dodamo nekoliko funkcionalnosti, na primer štit i vatreno oružje, svaku od tih komponenti možemo dinamički dodati kao dekorator na sam objekat, odnosno proširimo sam objekat još jednim slojem za svaku od funkcionalnosti koje želimo. U našem slučaju na esencijalni objekat najpre dodamo sloj koji predstavlja štit, a zatim na novonastali objekat dodamo još jedan sloj koji predstavlja vatreno oružje.

Obrasci Strategija i Dekorator imaju dosta sličnosti, samo što Strategija menja samu srž objekta tako što menja način izvršavanja nekog unutrašnjeg ponašanja, dok Dekorator menja spoljašnjost dodavanjem sloja za svaku funkcionalnost koju želimo. Dekorator može biti dosta koristan, međutim, ima svoja ograničenja, kao što su nemogućnost dinamičkog uklanjanja slojeva i nemogućnost interakcije među slojevima - što je često neophodno u radi sa video igrama. Idealno bi bilo kada bi iskombinovali prednosti oba obrasca - Strategije i Dekoratora, u jedan obrazac. Ovaj problem rešava sledeći obrazac koji možemo nazvati Model komponenta - objekat.

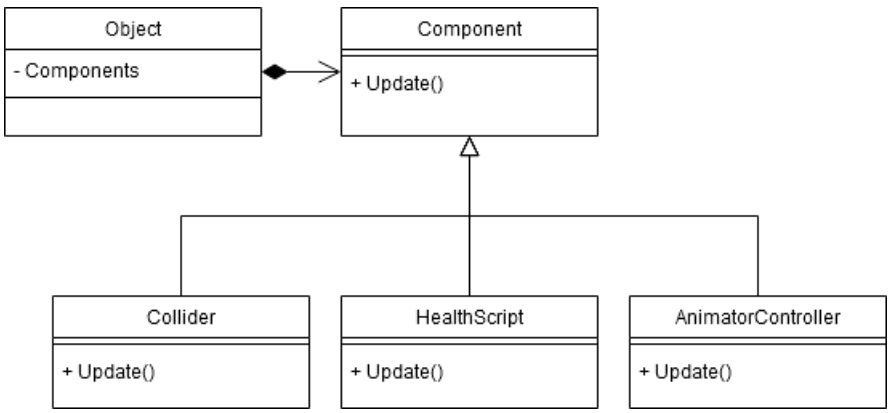
2.4 Model komponenta - objekat

Ideja obrasca Model komponenta - objekat se zasniva na obrnutom obrascu Dekorator. Dakle umesto raslojavanja samog objekta dodavanjem dekoratora spolja, u ovom obrascu dekoratori se nalaze unutar samog objekta, u nekom nizu ili listi. Može se implementirati na puno načina, a najjednostavnija verzija izgleda ovako:

```
using System;
using System.Collections.Generic;

namespace ComponentObjectModel
{
    class Object {
        List<Component> components;

        public Object() {
            components = new List<Component>();
        }

        public void AddComponent(Component component) {
            if (!components.Contains(component))
                components.Add(component);
        }

        public void RemoveComponent(Component component) {
            if (components.Contains(component)
                components.Remove(component);
        }
    }

    public abstract class Component {
        public abstract void Update();
    }

    class ConcreteComponentA : Component {

        public ConcreteComponentA() {}

        public override void Update() {
            Console.WriteLine("A");
        }

    }

    class ConcreteComponentB : Component {

        public ConcreteComponentB() {}

```

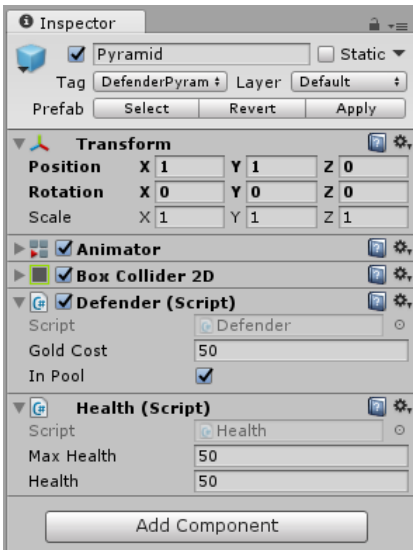
```
42          public override void Update() {
43              Console.WriteLine("B");
44          }
45
46      }
47
48      class Program
49      {
50          static void Main(string[] args)
51          {
52              Component concreteComponentA = new ConcreteComponentA()
    ;
53              Component concreteComopnentB = new ConcreteComponentB()
    ;
54
55              Object object = new Object();
56              object.AddComponent(concreteComponentA);
57              object.AddComponent(concreteComponentB);
58
59          }
60      }
61  }
```

Dakle naš objekat možemo posmatrati kao skup komponenti, odnosno ponašanja. Komponente mogu biti razne i svaka komponenta dodaje neku funkcionalnost samom objektu. Video igre se zasnivaju na *Update* metodu koji se izvršava u svakom frejmu. Svaka komponenta ima *Update* metod kojim definiše šta ta komponenta radi u svakom frejmu. Struktura ovog obrasca prikazana je na priloženom dijagramu.



Slika 2.5: Model komponenta - objekat: dijagram klasa

Same komponente možemo dodavati, brisati ili menjati dinamički pa nam ovaj obrazac pruža nephodnu fleksibilnost. Ovaj obrazac za projektovanje je uglavnom svojstven video igrama, mada se može koristiti i u druge svrhe. Pošto je komponenti pristup pogodan za kreiranje video igara, mnoga okruženja su zasnovana na komponentnoj paradigmi i imaju već implementirane složene verzije ovog obrasca. To je slučaj i sa okruženjem Unity. Kad god kreiramo objekat igre u okruženju Unity, on već sadrži listu sa komponentama.



Slika 2.6: Primer komponenti jednog objekta

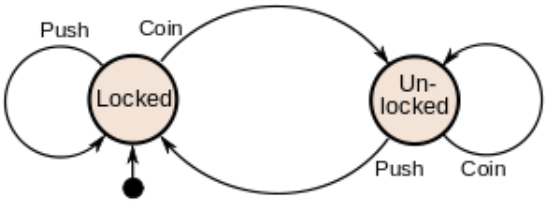
Unity ima puno gotovih komponenti koje možemo koristiti u razvoju. Na primer, svaki objekat inicijalno dobije komponentu Transform koja brine o položaju, rotaciji i skaliranju. Dodatna ponašanja objekta definišemo samim skriptama koje pišemo i dodajemo u listu komponenti tog objekta, jer je skripta u ovom okruženju zapravo samo jedna od komponenti. Zahvaljujući tome neophodno je malo izmeniti uobičajeni način razmišljanja i izdvajati funkcionalne celine u više ponovno iskoristivih komponenti.

Zahvaljujući prednostima ovog obrasca u odnosu na prethodna dva, imamo maksimalnu fleksibilnost, ali i pristupačnost, pa jedna komponenta može jednostavno pristupiti ostalim komponentama tog objekta. Recimo ako imamo komponentu koja treba da bude zadužena za kretanje objekta, ona će morati da pristupa komponenti Transform i menja njene parametre, konkretno koordinate objekta. Komponente

takođe možemo i dinamički da dodajemo i brišemo po potrebi, što nismo mogli sa prethodnim obrascima.

2.5 Stanje

Pre upoznavanja ovog obrasca, podsetimo se osnovnih stvari o konačnim automatima. **Konačni automat** predstavlja matematički model izračunavanja. Najbitnije odlike automata su njegov **skup stanja** i **relacije prelaza**. Najčešće se prikazuje grafom - čvorovi grafa prestavljaju stanja, a grane prelaske.



Slika 2.7: Primer konačnog automata

Prelazak iz stanja A u stanje B se može izvršiti samo ako postoji definisan prelaz između njih. Obrazac za projektovanje Stanje koristi svojstva konačnih automata kako bi modelovao realne situacije. Izuzetno je pogodan za računarske igre jer se često javlja potreba da određene objekte svrstamo u nekakva stanja. Može se implementirati na sledeći način:

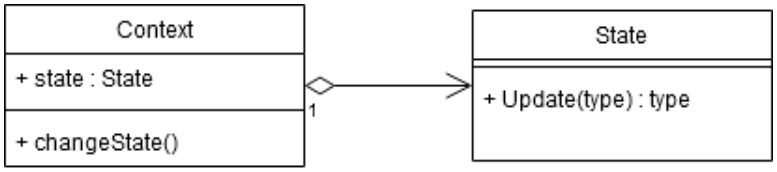
```
using System;

namespace State
{
    class Context
    {
        private State _state = null;

        public Context(State state)
        {
            this.TransitionTo(state);
        }

        public void TransitionTo(State state)
        {
            Console.WriteLine($"Context: Transition to {state.
    GetType().Name}.");
            this._state = state;
            this._state.SetContext(this);
        }

        public void Request1()
        {
            this._state.Handle1();
        }

        public void Request2()
        {
            this._state.Handle2();
        }
    }

    abstract class State
    {
        protected Context _context;

        public void SetContext(Context context)
        {
            this._context = context;
        }

```

```
41          public abstract void Handle1();
42
43          public abstract void Handle2();
44      }
45
46      class ConcreteStateA : State
47      {
48          public override void Handle1()
49          {
50              Console.WriteLine("ConcreteStateA handles request1.");
51              Console.WriteLine("ConcreteStateA wants to change the
    state of the context.");
52              this._context.TransitionTo(new ConcreteStateB());
53          }
54
55          public override void Handle2()
56          {
57              Console.WriteLine("ConcreteStateA handles request2.");
58          }
59      }
60
61      class ConcreteStateB : State
62      {
63          public override void Handle1()
64          {
65              Console.Write("ConcreteStateB handles request1.");
66          }
67
68          public override void Handle2()
69          {
70              Console.WriteLine("ConcreteStateB handles request2.");
71              Console.WriteLine("ConcreteStateB wants to change the
    state of the context.");
72              this._context.TransitionTo(new ConcreteStateA());
73          }
74      }
75
76      class Program
77      {
78          static void Main(string[] args)
79          {
80              var context = new Context(new ConcreteStateA());
```

```
81              context.Request1();
82              context.Request2();
83          }
84      }
85  }
```

Apstraktna klasa *State* sadrži polje *_context* i metode *Handle1* i *Handle2*. Konkretna stanja nasleđuju apstraktnu klasu i sadrže svoje implementacije metoda *Handle1* i *Handle2*. Metodi *Handle* mogu da izazovu promenu stanja. Oni zapravo predstavljaju prelaze konačnog automata koji ovaj kôd modeluje. Klasa *Context* sadrži polje *_state* koje predstavlja tekuće stanje i metod *TransitionTo* koji vrši promenu stanja. Možemo smatrati da je *Context* zapravo tekuće stanje automata u nekom trenutku vremena. U *Main* metodi vidimo dva prelaza koji između ostalog vrše promenu stanja.

Komponenta za animaciju okruženja Unity već sadrži implementiranu podršku za kreiranje automata kojim će se određivati stanja nekog objekta. Pri kreiranju automata za animaciju, kreira se i početno stanje *Entry* iz kog se dalje definišu prelasci i nova stanja. Jedno stanje u principu predstavlja jednu animaciju, iz koje se okidačima pokreće određeni deo kôda koji utiče na dalji tok igre. Na primer, kada se objekat napadač nalazi u stanju napada, on će kroz svoju animaciju koristeći okidač pozvati metodu koja zaista izvršava napad. Dakle, ovaj obrazac nam pomaže u kreiranju veštačke inteligencije tako što objektima dodelimo određeno ponašanje u zavisnosti od trenutnog stanja.



Slika 2.8: Dijagram klase obrasca Stanje

2.6 Fabrika

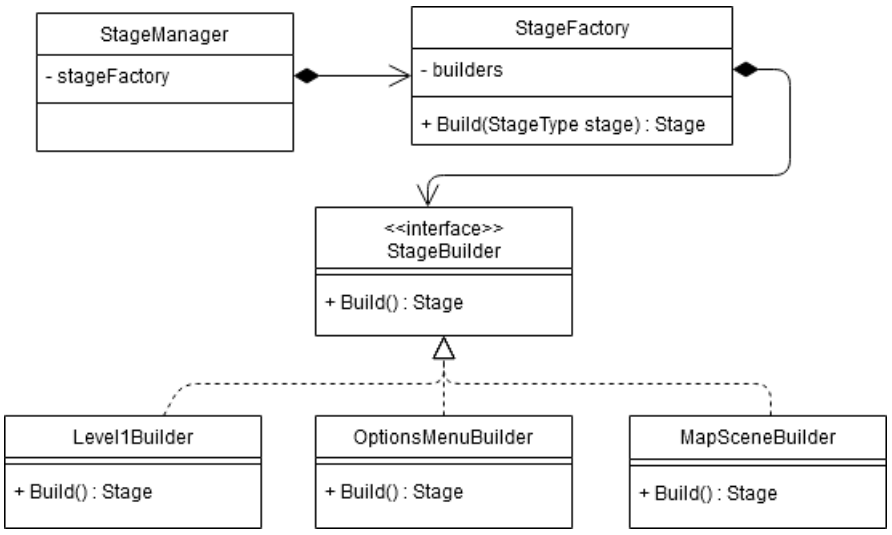
Obrazac za projektovanje Fabrika koristi se kako bi se izbeglo direktno pozivanje konstruktora prilikom kreiranja objekata. Ideja je da se omogući kreiranje instanci izvedenih klasa bez specifikacije koju klasu, odnosno instancu, želimo da kreiramo. Ovo se postiže kreiranjem interfejsa za kreiranje objekata, dopuštajući potklasama da same definišu koji objekat treba instancirati. Primer implementacije ovog obrasca:

```
using System;

namespace RefactoringGuru.DesignPatterns.FactoryMethod.Conceptual
{

    abstract class Creator
    {

        public abstract IProduct FactoryMethod();

        public string SomeOperation()
        {
            var product = FactoryMethod();
            var result = "Creator: The same creator's code has just
    worked with "
                + product.Operation();

            return result;
        }
    }

    class ConcreteCreator1 : Creator
    {

        public override IProduct FactoryMethod()
        {
            return new ConcreteProduct1();
        }
    }

    class ConcreteCreator2 : Creator
    {
        public override IProduct FactoryMethod()
```

```
33          {
34              return new ConcreteProduct2();
35          }
36      }
37
38      public interface IProduct
39      {
40          string Operation();
41      }
42
43      class ConcreteProduct1 : IProduct
44      {
45          public string Operation()
46          {
47              return "{Result of ConcreteProduct1}";
48          }
49      }
50
51      class ConcreteProduct2 : IProduct
52      {
53          public string Operation()
54          {
55              return "{Result of ConcreteProduct2}";
56          }
57      }
58
59      class Client
60      {
61          public void Main()
62          {
63              Console.WriteLine("Launched with the ConcreteCreator1."
    );
64              ClientCode(new ConcreteCreator1());
65
66              Console.WriteLine("");
67
68              Console.WriteLine("Launched with the ConcreteCreator2."
    );
69              ClientCode(new ConcreteCreator2());
70          }
71
72          public void ClientCode(Creator creator)
```

```
73          {
74              Console.WriteLine(creator.SomeOperation());
75          }
76      }
77
78      class Program
79      {
80          static void Main(string[] args)
81          {
82              new Client().Main();
83          }
84      }
85  }
```

I implementaciji imamo apstraktnu klasu *Creator* koja sadrži metod *FactoryMethod*. Konkretni kreatori nasleđuju ovu klasu i sadrže svoje implementacije ovog metoda koje kreiraju konkretan proizvod. U *Main* metodi se u zavisnosti od kreatora dobija konkretan proizvod koji taj kreator proizvodi.



Slika 2.9: Dijagram klase obrasca Fabrika

Ovaj obrazac se koristi i u kreiranju video igara. Na primer, ako želimo da implementiramo sistem za učitavanje scena, možemo iskoristiti ovaj obrazac. Klasa *StageFactory* bi sadržala kolekciju *builders* objekata za kreiranje koji bi bili tipa *StageBuilderInterface*. Svaka scena bi implementirala ovaj interfejs i metod *build()* i na taj način bi naša fabrika za kreiranje scena kreirala različite scene, iako ona sama ne zna kako se kreira svaki od objekata. Bez korišćenja ovog obrasca, morali bi

da imamo neki ogroman switch koji bi vodio računa o tome šta se tačno instancira i kako. Dodavanje nove scene bi bilo prilično teško i nepregledno, dok se korišćenjem ovog obrasca dodavanje nove scene svodi na implementiranje interfejsa.

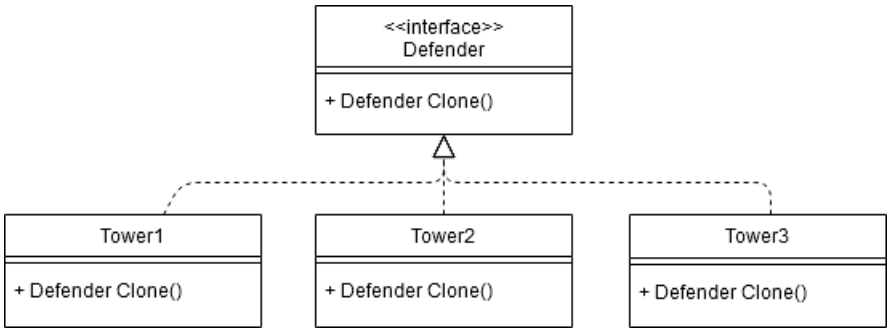
2.7 Prototip

U prethodnom odeljku smo videli kako se mogu kreirati objekti korišćenjem fabrika na primeru scena. Međutim, objekti koji se aktivno koriste u igri (na primer napadači, branitelji itd.) su u principu kolekcija najrazličitijih komponenti, i imaju tendenciju da se često menjaju tokom razvoja igre. Pošto komponente mogu biti drastično različite i da se često menjaju/dodaju, obrazac za projektovanje Fabrika nije baš preterano pogodan za ovakve objekte. Zato uvodimo obrazac Prototip. Ovaj obrazac nam nudi način za kreiranje kopija objekata bez znanja o samom tipu objekta. Ideja je da se napravi sam prototip objekta koji će se kasnije kopirati kad god nam zatreba. Prototip je obrazac kreiranja i zasniva se na kloniranju objekta, a to se može izvesti ovako:

```
1  using System;
2
3  namespace Prototype
4  {
5      public class Person
6      {
7          public int Age;
8          public DateTime BirthDate;
9          public string Name;
10         public IdInfo IdInfo;
11
12         public Person Clone()
13         {
14             Person clone = (Person) this.MemberwiseClone();
15             clone.IdInfo = new IdInfo(IdInfo.IdNumber);
16             clone.Name = String.Copy(Name);
17             return clone;
18         }
19     }
20
21     public class IdInfo
22     {
```

```
23          public int IdNumber;
24
25          public IdInfo(int idNumber)
26          {
27              this.IdNumber = idNumber;
28          }
29      }
30
31      class Program
32      {
33          static void Main(string[] args)
34          {
35              Person p1 = new Person();
36              p1.Age = 108;
37              p1.BirthDate = Convert.ToDateTime("1912-06-23");
38              p1.Name = "Alan Turing";
39              p1.IdInfo = new IdInfo(666);
40
41              Person p2 = p1.Clone();
42
43              Console.WriteLine("   p1 instance values: ");
44              DisplayValues(p1);
45              Console.WriteLine("   p2 instance values:");
46              DisplayValues(p2);
47
48          }
49
50          public static void DisplayValues(Person p)
51          {
52              Console.WriteLine("      Name: {0:s}, Age: {1:d},
    BirthDate: {2:MM/dd/yy}",
53                  p.Name, p.Age, p.BirthDate);
54              Console.WriteLine("      ID#: {0:d}", p.IdInfo.IdNumber
    );
55          }
56      }
57  }
```

Ovaj princip se naziva i duboko kopiranje, gde se, osim samog objekta, u potpunosti kopiraju i svi složeni objekti i strukture podataka. Često se u implementaciji, osim metoda *clone*, nađe i neki zajednički interfejs koji ima tu metodu. U tom slučaju konkretne klase implementiraju interfejs i svoju verziju metoda *clone*.



Slika 2.10: Dijagram klase obrasca Prototip

Unity već ima implementiranu naprednu verziju ovog obrasca kroz sistem prifaba. Prifab predstavlja prototip objekta koji se zatim instancira korišćenjem metoda *Instantiate*. Kada se instancira, kopija objekta sadrži sve komponente koje ima i originalni objekat, odnosno prifab.

2.8 Komanda

Još jedan od obrazaca koji su u principu opšti, ali se dosta primenjuju u video igrama, jeste Komanda. Često se sreće potreba da se izvrše neke akcije bez previše upuštanja u to kako će se one izvršiti. Uzmimo džojstik kao primer. Na njemu imamo više tastera. Svaki taster treba da uradi neku akciju, ali detalje same akcije ne treba da zna sam džojstik. Ovaj obrazac se dosta primenjuje i u slučajevima kada je potrebno čuvati informacije o akcijama i poništavati ih, odnosno za implementaciju *undo* i *redo* komandi. Primer implementacije ovog obrasca:

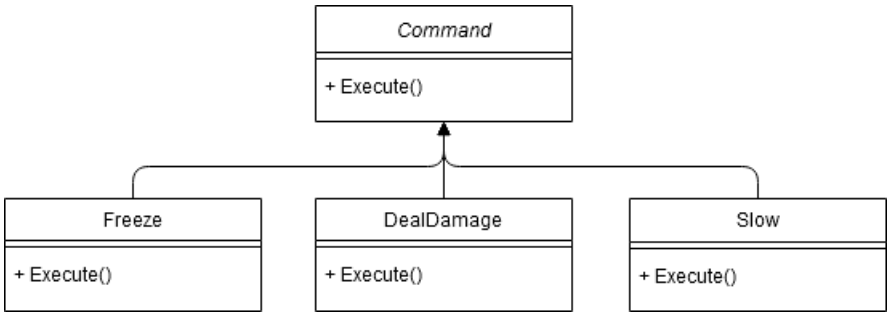
```
using System;

namespace Command
{

    public interface ICommand
    {
        void Execute();
    }

    class SimpleCommand : ICommand
    {
        private string _payload = string.Empty;

        public SimpleCommand(string payload)
        {
            this._payload = payload;
        }

        public void Execute()
        {
            Console.WriteLine($"Printing ({this._payload})");
        }
    }

    class ComplexCommand : ICommand
    {
        private Receiver _receiver;

        private string _a;

        private string _b;

        public ComplexCommand(Receiver receiver, string a, string b
    )
        {
            this._receiver = receiver;
            this._a = a;
            this._b = b;
        }

```

```
41          public void Execute()
42          {
43              Console.WriteLine("ComplexCommand");
44              this._receiver.DoSomething(this._a);
45              this._receiver.DoSomethingElse(this._b);
46          }
47      }
48
49      class Receiver
50      {
51          public void DoSomething(string a)
52          {
53              Console.WriteLine($"Receiver: Working on ({a}.)");
54          }
55
56          public void DoSomethingElse(string b)
57          {
58              Console.WriteLine($"Receiver: Also working on ({b}.)");
59          }
60      }
61
62      class Invoker
63      {
64          private ICommand _onStart;
65
66          private ICommand _onFinish;
67
68          public void SetOnStart(ICommand command)
69          {
70              this._onStart = command;
71          }
72
73          public void SetOnFinish(ICommand command)
74          {
75              this._onFinish = command;
76          }
77
78          public void DoSomethingImportant()
79          {
80
81              if (this._onStart is ICommand)
82              {
```

```
83                  this._onStart.Execute();
84              }
85
86              Console.WriteLine("Invoker: ... some job done ...");
87
88              if (this._onFinish is ICommand)
89              {
90                  this._onFinish.Execute();
91              }
92          }
93      }
94
95      class Program
96      {
97          static void Main(string[] args)
98          {
99
100             Invoker invoker = new Invoker();
101             invoker.SetOnStart(new SimpleCommand("Say Hi!"));
102             Receiver receiver = new Receiver();
103             invoker.SetOnFinish(new ComplexCommand(receiver, "Send
    email", "Save report"));
104
105             invoker.DoSomethingImportant();
106         }
107     }
108 }
```

U implementaciji ovog obrasca najčešće se koristi interfejs ili apstraktna klasa *Command* koja sadrži metod *Execute*, a konkretne komande nasleđuju ovu klasu i implementiraju svoju verziju ovog metoda. Sada se izvršavanje akcija ili niza akcija svodi na kreiranje odgovarajućih komandi i njihovog izvršavanja koje je uniformno zahvaljujući apstraktnoj natklasi koju dele. U primeru implementacije u *Main* metodi imamo pomoćni objekat *Invoker* koji sadrži dve komande, *_onStart* i *_onFinish*. Komande mogu biti raznovrsne i različite kompleksnosti, kao što su *SimpleCommand* i *ComplexCommand* u primeru. Njihovo izvršavanje se sada svodi na postavljanje odgovarajuće komande za početak i kraj izvršavanja.

Ovim smo kreirali izuzetno fleksibilan sistem koji može na jednostavan način da izvrši proizvoljan niz akcija koje se često ponavljaju.



Slika 2.11: Dijagram klase obrasca Komanda

2.9 Kolekcija objekata

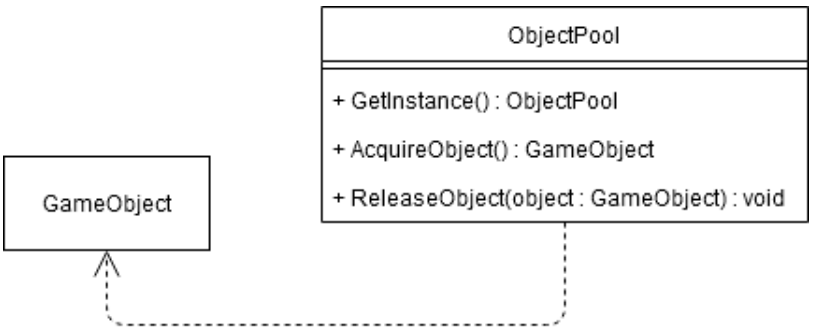
Jedan od obrazaca za projektovanje koji se dosta često koriste u video igrama jeste Kolekcija objekata (*Object Pool*). Koristi se da bi poboljšao performanse same igre. U video igrama često imamo objekte koji se instanciraju više puta. Na primer, u pucačkim igrama će se jako često javiti potreba za instanciranjem novog objekta koji predstavlja metak. Kreiranje i uništavanje objekata spada u dosta skupe operacije, i kada bi se u svakom frejmu kreirale i uništavale hiljade objekata, to bi ozbiljno narušilo performanse same igre. Zato se uvodi takozvana *kolekcija objekata*, odnosno *object pool*. Ideja je da se objekti koje treba uništiti ne unište zaista, već samo prebace na drugo mesto van scene igre. Takođe, umesto kreiranja novog objekta, prvo treba potražiti da li taj objekat postoji u kolekciji, i u slučaju da postoji, uzeti odgovarajući objekat odatle i proglasiti ga aktivnim. Primer jednostavne implementacije ovog obrasca:

```
1  using System;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  namespace ObjectPool
6  {
7
8      class ObjectPool
9      {
10         private List<GameObject> pooledObjects;
11
12         public ObjectPool(int numberOfObjects)
13         {
```

```
14              pooledObjects = new List<GameObject>();
15              for (int i=0; i<numberOfObjects; i++)
16              {
17                  pooledObjects.Add(new GameObject());
18              }
19          }
20
21          public GameObject GetPooledObject()
22          {
23              for (int i=0; i<pooledObjects.Count; i++)
24              {
25                  if (!pooledObjects[i].activeInHierarchy)
26                  {
27                      pooledObjects[i].SetActive(true);
28                      return pooledObjects[i];
29                  }
30              }
31          return null;
32          }
33
34          public void AddObjectToPool(GameObject o)
35          {
36              obj.SetActive(false);
37          }
38      }
39
40      class Program
41      {
42          static void Main(string[] args)
43          {
44              ObjectPool pool = new ObjectPool(5);
45              GameObject go = pool.GetPooledObject();
46              pool.AddObjectToPool(go);
47          }
48      }
49  }
```

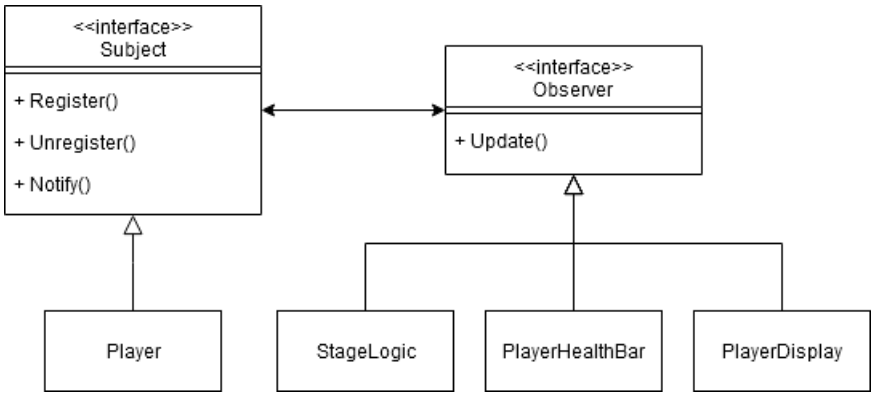
U pređašnjem primeru klasa *ObjectPool* pri kreiranju napravi inicijalni broj objekata koji se mogu koristiti. Umesto kreiranja novog objekta, koristi se *ObjectPool* za dohvatanje novih objekata, koji u stvari samo aktivira već postojeći objekat. Kada dođe vreme da se objekat uništi, umesto toga on se deaktivira, odnosno vraća u *ObjectPool* kako bi se ponovo iskoristio kada zatreba. Ovaj obrazac se često koristi

u kombinaciji sa Singltonom kako bi se osigurali da imamo samo jedan *ObjectPool*.



Slika 2.12: Kolekcija objekata

2.10 Posmatrač



Slika 2.13: Posmatrač

Obrazac za projektovanje Posmatrač definiše jedan-prema-više relaciju između objekata. Kada se stanje jednog objekta promeni, svi ostali koji imaju zavisnost ka njemu budu obavešteni. Imena objekata u ovom obrascu su najčešće **subjekat** i **posmatrač**. Subjekat sadrži podatke koje posmatrači treba da znaju. Može se shvatiti i kao sistem pretplate (subscribe). Na primer, subjekat se može posmatrati kao YouTube kanal, a posmatrači su svi ljudi koji su pretplaćeni na taj kanal. Kada stigne novi video, svi oni dobijaju notifikaciju, odnosno obaveštenje o tome. Alternativa ovog pristupa bi bila da posmatrači stalno proveravaju da li ima novih informacija, što bi izazvalo česte bespotrebne akcije. Implementacija objekta najčešće sadrži metode *registerObserver*, *unregisterObserver* i *notify*, dok posmatrači

uglavnom imaju metod *update*. Ovaj obrazac je u principu opšti, ali se dosta primenjuje u video igrama. Na primer, često u igri imamo objekat igrač, i mnoge objekte koji zavise od ovog objekta i koje možemo predstaviti kao posmatrače. Tu se na primer mogu naći razni slajderi koji prikazuju trenutno stanje igrača (health bar, ammo bar itd.), sam izgled igrača, logika na samoj sceni itd. Primer implementacije ovog obrasca:

```
using System;
using System.Collections.Generic;
using System.Threading;

namespace Observer
{
    public interface IObserver
    {
        void Update(ISubject subject);
    }

    public interface ISubject
    {
        void Register(IObserver observer);

        void Unregister(IObserver observer);

        void Notify();
    }

    public class Subject : ISubject
    {

        public int State { get; set; } = -0;

        private List<IObserver> _observers = new List<IObserver>();

        public void Register(IObserver observer)
        {
            Console.WriteLine("Subject: Registered an observer.");
            this._observers.Add(observer);
        }

        public void Unregister(IObserver observer)
        {
```

```
36              this._observers.Remove(observer);
37              Console.WriteLine("Subject: Unregistered an observer.")
    ;
38          }
39
40          public void Notify()
41          {
42              Console.WriteLine("Subject: Notifying observers...");
43
44              foreach (var observer in _observers)
45              {
46                  observer.Update(this);
47              }
48          }
49
50          public void SomeBusinessLogic()
51          {
52              Console.WriteLine("\nSubject: Doing some work");
53              this.State = new Random().Next(0, 10);
54
55              Thread.Sleep(15);
56
57              Console.WriteLine("Subject: My state has just changed
    to: " + this.State);
58              this.Notify();
59          }
60      }
61
62      class ConcreteObserverA : IObserver
63      {
64          public void Update(ISubject subject)
65          {
66              if ((subject as Subject).State < 3)
67              {
68                  Console.WriteLine("ConcreteObserverA: Reacted to
    the event.");
69              }
70          }
71      }
72
73      class ConcreteObserverB : IObserver
74      {
```

```
75          public void Update(ISubject subject)
76          {
77              if ((subject as Subject).State == 0 || (subject as
    Subject).State >= 2)
78              {
79                  Console.WriteLine("ConcreteObserverB: Reacted to
    the event.");
80              }
81          }
82      }
83
84      class Program
85      {
86          static void Main(string[] args)
87          {
88              var subject = new Subject();
89              var observerA = new ConcreteObserverA();
90              subject.Register(observerA);
91
92              var observerB = new ConcreteObserverB();
93              subject.Register(observerB);
94
95              subject.SomeBusinessLogic();
96              subject.SomeBusinessLogic();
97
98              subject.Unregister(observerB);
99
100             subject.SomeBusinessLogic();
101         }
102     }
103 }
```

Glava 3

Opis igre

Praktični deo ovog rada čini igra *Mummy Invasion*. Pripada žanru Odbrana tornjevima (engl. *Tower Defense*). Svrha igre je odbraniti se od najezde mumija koristeći odbrambene tornjeve. Postoje 4 vrste tornjeva - piramida, koja generiše neophodne resurse, bloker koji služi da zadrži neprijatelje, i dve vrste tornjeva koji ispaljuju projektile i uništavaju neprijatelje. Igra se protiv računara i cilj je preživeti dovoljno dugo a da nijedna mumija ne pređe na drugi kraj terena. Igra trenutno ima 5 nivoa, svaki teži od prethodnog, kao i 3 moguća podešavanja težine - lako, srednje, teško.

3.1 Scene

Igra se sastoji iz 4 osnovne i 3 pomoćne scene. Osnovne scene su:

- Početna scena
- Scena za opcije
- Mapa
- Scena igre

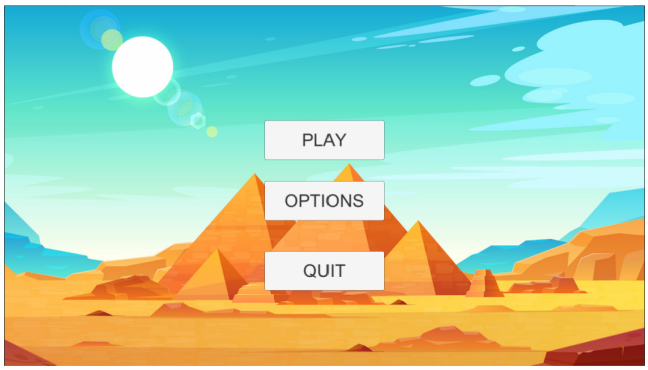
Pomoćne scene nisu ključne za igru i teorijski se mogu izbaciti. Tu spadaju:

- Ulazna scena (Splash Screen)
- Scena pobede

- Scena poraza

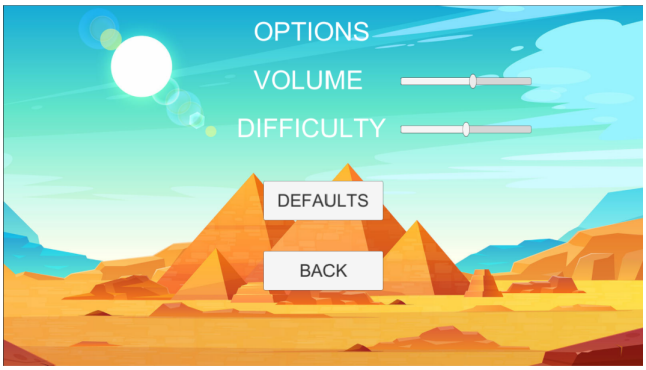
Sadrže samo sliku i odgovarajući tekst.

Početna scena je prilično jednostavna - sadrži samo 3 dugmeta. Jedno za prelaz na scenu mape, jedno za prelaz na scenu za opcije i jedno za izlaz iz aplikacije.



Slika 3.1: Početna scena

Scena za opcije služi za podešavanje zvuka i težine same igre. Opcije se konfigurišu korišćenjem slajdera. Slajder za težinu ima samo tri podeoka (lako, srednje, teško), dok je slajder za zvuk neprekidan. Na sceni se nalazi i dugme *defaults* koje postavlja slajdere na podrazumevane vrednosti (srednju težinu i blago pojačan zvuk). Postoji i dugme *back* koje služi za povratak na prethodnu scenu. Pritiskom dugmeta *back* se takođe čuvaju podešavanja u samom uređaju koji se koristi za igru, to jest računaru, tabletu ili mobilnom telefonu.

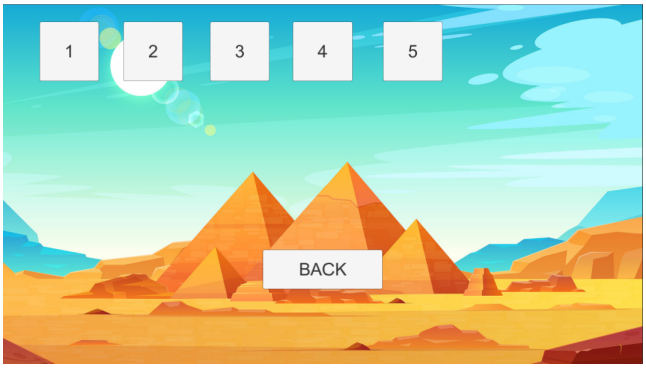


Slika 3.2: Scena za opcije

Sledeća scena jeste **Mapa**. Ona služi za odabir nivoa. Kada se igra pokrene prvi put, na njoj će se naći samo dugme za pokretanje prvog nivoa, odnosno dugme sa brojem 1. Sa svakim pređenim nivoom, u ovu scenu će se dodavati po jedno novo

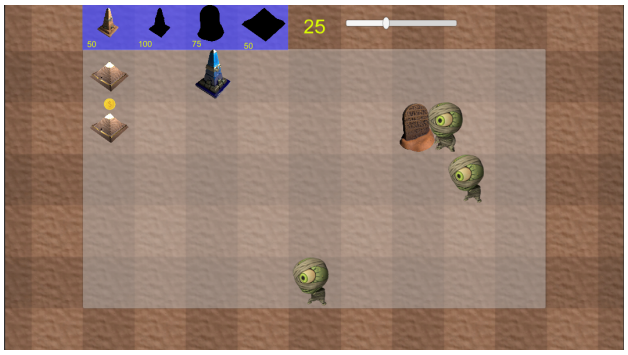
dugme koje pokreće odgovarajući nivo, i tako sve dok se ne otključaju svi nivoi. Kada se jednom otključa, nivo se može igrati ponovo proizvoljan broj puta. Podaci o pređenim nivoima će, između ostalog, biti sačuvani i na samom uređaju koji korisnik koristi za igru, koristeći *PlayerPrefs*.

PlayerPrefs je skript API okruženja Unity koji služi za čuvanje podataka na samom uređaju, nezavisno od specifikacija samog uređaja i operativnog sistema. Često se koristi kako bi se sačuvalo stanje igre i kada se izađe iz nje. Količina memorije je mala i moguće je čuvati samo osnovne tipove (string, integer, float). Igra *Mummy Invasion* koristi ovu pogodnost kako bi lokalno sačuvala konfiguraciju. Podaci koji se čuvaju na uređajima jesu broj nivoa, težina i podešavanje zvuka.



Slika 3.3: Mapa

Scena igre je najvažnija scena. Sastoji se iz matrice 5x9 koja sadrzi polja za postavljanje tornjeva, menija sa dugmićima za kreiranje tornjeva, displeja za prikaz resursa (zlatnika) i slajdera koji označava progres igre.



Slika 3.4: Scena igre

Korisnik postavlja novi toranj tako što iz menija sa dugmićima najpre odabere toranj koji želi da napravi, a zatim klikom na jedno od polja u matrici postavlja odabrani toranj na željeno mesto. Toranj iz menija za tornjeve odabirom, odnosno klikom, postaje potpuno vidljiv, dok se kod neaktivnih vidi samo kontura. Toranj je moguće postaviti samo na slobodno polje u matrici - nije moguće gomilanje više tornjeva na jedno polje. Takođe, nije moguće postavljanje tornjeva van matrice, odnosno van svetlih polja u sceni igre. Da bi se toranj napravio, neophodno je da igrač ima dovoljno resursa, odnosno zlatnika, za njegovo kreiranje. Trenutnu količinu zlatnika igrač može videti u svakom trenutku na displeju između menija sa tornjevima i slajdera za progres igre. Slajder za progres igre se pomera tokom vremena, i kada stigne do kraja, označava uspešan kraj tekućeg nivoa.

Sem vidljivih delova, postoje još dva koja nisu vidljiva, a to su objekti za kreiranje neprijatelja (za svaki red po jedan) (*spawners*) i objekat za uništavanje (*shredder*). Objekat za kreiranje neprijatelja sadrži niz objekata koje treba kreirati i kreira ih kada bude vreme za to. Uništavač se nalazi skroz desno. Sadrži kolajder i uništava svaki objekat koji se sudari sa njim. Njegov cilj je uništavanje zalutalih projektila koji bi, u nedostatku ovog objekta, nastavili da se kreću bez potrebe i izazivali bi curenje memorije.

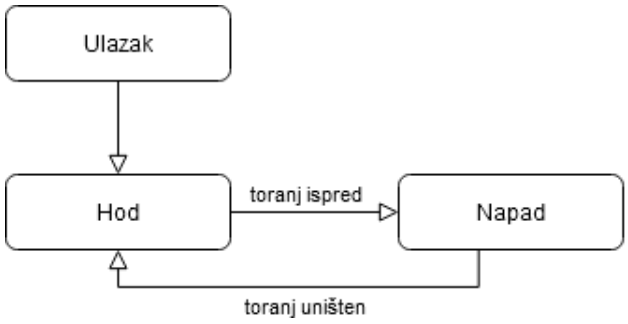
Glavni elementi u ovoj igri su napadači, odnosno mumije, i branitelji, odnosno tornjevi. Za svaki od njih postoji prifab, odnosno prototip objekta koji sadrži grafiku, kolajdere, skripte za ponašanje i sve ostale neophodne stvari.

3.2 Napadači

Napadači su mumije koje se stvaraju na desnom kraju ekrana i pokušavaju da prodru do levog dela, u čemu igrač treba da ih spreči. Ova igra sadrži dva tipa napadača, ali se razlikuju samo u konfiguraciji i grafici. Bela mumija je slabija ali se češće javlja, dok je crvena mumija jača ali se ređe pojavljuje.

Slika 3.5: Napadači

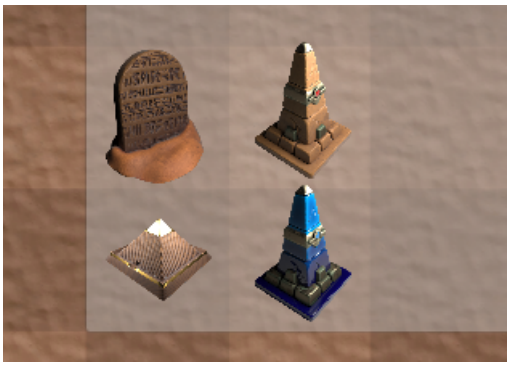
Ponašanje mumija određeno je njihovim stanjima. Kada se kreira, mumija je u stanju ulaska. Nakon toga prelazi u hod sve dok ne naiđe na toranj, odnosno kada im se poklope kolajderi. Tada mumija prelazi u stanje napada. U tom stanju polako uništava toranj sve dok ga potpuno ne uništi. Kada je toranj uništen, mumija se vraća u stanje hoda.



Slika 3.6: Ponašanje mumija

3.3 Branitelji

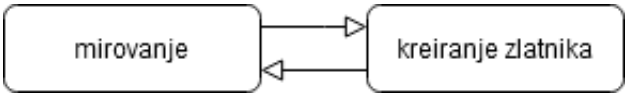
Branitelji su objekti koje igrač kreira kako bi se odbranio od mumija. Postoje četiri vrste branitelja: piramida - branitelj koji generiše resurse, bloker - izdržljiv objekat koji služi kao štit, i dve vrste tornjeva (običan i ledeni) koji ispaljuju projektile na napadače.



Slika 3.7: Branitelji - bloker, toranj, piramida i ledeni toranj

Piramida je branitelj koji generiše resurse. Za kreiranje novih branitelja, neophodni su resursi, odnosno zlatnici. Igrač započinje nivo sa malom količinom zlatnika, ali da bi napredovao, neophodno je generisati nove resurse koje piramida proizvodi.

Ponašanje piramide je prilično jednostavno - nakon kreiranja nalazi se u stanju mirovanja i u njemu je neko vreme. Nakon određenog vremena prelazi u stanje u kom kreira zlatnik. Vrednost kreiranog zlatnika se automatski dodaje u raspoloživu količinu zlatnika koje poseduje igrač, što se može videti na displeju za zlatnike. Kada se kreira zlatnik, piramida se vraća nazad u stanje mirovanja.



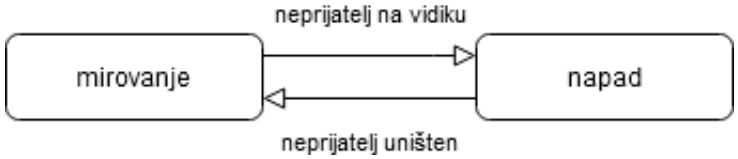
Slika 3.8: Ponašanje piramide

Bloker je izdržljiv branitelj kog mumije ne mogu lako uništiti. Služi da zadržava mumije dok ih ostali tornjevi uništavaju. On nema nikakvo specijalno ponašanje.

Tornjevi su branitelji koji ispaljuju projektile na napadače. Postoje dve vrste tornjeva - običan i ledeni. Ledeni dodatno usporava svakog napadača kog pogodi projektilom, ali je skuplji od običnog, to jest njegovo kreiranje zahteva više zlatnika. Svaki projektil koji pogodi metu polako oštećuju mumiju sve dok se ona ne bude potpuno uništena.

Ponašanje tornjeva određeno je njihovim stanjima. Nakon kreiranja, tornjevi su u stanju mirovanja. Kada toranj detektuje napadača u liniji ispred sebe, prelazi u

stanje napada. Dok god je u stanju napada, toranj ispaljuje projektile ispred sebe. Kada projektili unište napadača, toranj se vraća u stanje mirovanja.



Slika 3.9: Ponašanje tornjeva

3.4 Tok igre

Kada uđe u igru, korisnik najpre nekoliko sekundi vidi ulaznu scenu (splash screen) nakon čega se prelazi u početnu scenu. Iz nje može preći u scenu za opcije ili u mapu. Preporuka je da se prilikom prvog pokretanja najpre uđe u scenu za opcije kako bi igrač podesio željenu težinu i zvuk. Kada su opcije podešene, korisnik prelazi na scenu mape gde bira jedan od nivoa. Na početku će samo prvi nivo biti otključan, a sa svakim prelaskom nivoa otključava se sledeći. Sa odabirom nivoa, prelazi se u scenu igre. U njoj korisnik kreira tornjeve odabirom željenog tornja iz menija, a zatim klikom na polje gde želi da toranj bude postavljen. Dok traje igra, slajder sa vremenom se polako pomera. Da bi prešao nivo, igrač mora da preživi dovoljno dugo. Vreme preživljavanja neophodno za prelazak nivoa povećava se sa svakim nivoom. Ako mumije prodru do leve strane ekrana pre nego što istekne vreme preživljavanja, igrač biva prebačen u scenu poraza odakle može ponovo pokrenuti nivo, ili izaći iz igre. Kada pređe i poslednji nivo, igrač prelazi na scenu pobede. Nakon toga i dalje može prelaziti stare nivoe, možda sa drugim podešavanjima za težinu.

Glava 4

Implementacija

Za implementaciju igre *Mummy invasion* korišćen je Unity uz programski jezik C#. Tokom kreiranja igre korišćeni su neki od često korišćenih obrazaca za projektovanje koji su pogodni za pravljenje igara iz ovog žanra. Za neke obrasce je iskorišćena gotova implementacija koju pruža okruženje Unity, dok su neki ručno implementirani. Igra se može naći na sledećoj adresi: https://github.com/strahinja94/mummy-invasion.

4.1 Muzika i obrazac Singlton

Kao i skoro sve druge igre, i ova igra sadrži muziku. Zvukove i pozadinsku muziku u igrici je neophodno pozivati iz raznih scena i delova kôda, pa je zgodno napraviti jedinstvenu klasu koja će voditi računa o tome. Klasa koja je zadužena za kontrolu zvukova jeste klasa *MusicManager*. Postojanje više instanci klase *MusicManager* nema smisla i može prouzrokovati probleme poput neželjenog preklapanja muzika iz raznih nivoa, pa je u ovoj konkretnoj situaciji najbolje koristiti obrazac Singlton. Pogledajmo najpre kako izgleda sama klasa:

```
using UnityEngine;

public class MusicManager : MonoBehaviour {

    public AudioSound sound;

    private static MusicManager _instance;

    public static MusicManager Instance
    {
        get
        {
            if (_instance == null)
            {
                _instance = GameObject.FindObjectOfType<
   MusicManager>();
                InitLoadAudio();
                DontDestroyOnLoad(_instance.gameObject);
            }
            return _instance;
        }
    }

    void Awake()
    {
        if (_instance == null)
        {
            AudioSound snd = gameObject.GetComponentInChildren<
   AudioSound>();
            _instance = this;
            _instance.sound = snd;
            InitLoadAudio();
            DontDestroyOnLoad(this);
        }
        else
        {
            if (this != _instance)
                Destroy(this.gameObject);
        }
    }

```

```
40      public static void InitLoadAudio()
41      {
42          _instance.sound.InitLoad();
43      }
44
45      public void PlaySound(string fxFilename)
46      {
47          sound.PlaySound(fxFilename);
48      }
49
50      public void StopSound(string fxFilename)
51      {
52          sound.StopSound(fxFilename);
53      }
54
55      public void PlaySoundLoop(string fxFilename)
56      {
57          sound.PlaySoundLoop(fxFilename);
58      }
59
60      public void StopSoundLoop(string fxFilename)
61      {
62          sound.StopSoundLoop(fxFilename);
63      }
64
65      public void MuteSound()
66      {
67          sound.soundVolumeMultiplier = 0;
68          sound.SetVolume();
69      }
70
71      public void UnMuteSound()
72      {
73          sound.soundVolumeMultiplier = 1;
74          sound.SetVolume();
75      }
76
77      public void SetSoundMultiplier(float multiplierValue)
78      {
79          if (multiplierValue < 0)
80              multiplierValue = 0;
81
```

```
82          if (multiplierValue > 1)
83              multiplierValue = 1;
84
85          sound.soundVolumeMultiplier = multiplierValue;
86          sound.SetVolume();
87      }
88
89      public bool IsPlaying()
90      {
91          return sound.IsPlaying();
92      }
93
94  }
```

MusicManager objekat na sceni sadrži skriptu *MusicManager* i podobjekat *Sound* koji sadrži komponentu Unity-ja *Audio Source*, kao i ručno pisanu skriptu *Audio Sound*. Sama skripta *MusicManager* sadrži privatnu instancu *_instance* kao i *Instance* koji se može dohvatiti bilo gde iz kôda. Dohvatanje iz drugih klasa izgleda ovako:

```
1 MusicManager.Instance.StopSound("PlaySceneMusic");
```

Kada se pozove *MusicManager.Instance*, ako je privatna instanca null, Unity će naći odgovarajući objekat na sceni i postaviti privatnu instancu na već postojeći objekat *MusicManager*. Takođe će pozvati init metod i reći okruženju Unity da ne uništava objekat prilikom učitavanja drugih scena. Ako privatna instanca već postoji, ona će se vratiti i neće se kreirati nova. Ovim smo implementirali obrazac Singlton i osigurali se da će postojati samo jedna instanca klase *MusicManager* odakle god da se poziva i u ma kojoj sceni se nalazili u datom trenutku.

Ova klasa pre pokretanja, odnosno u *Awake* metodi postavlja pomoćnu klasu *AudioSound* i učitava zvukove iz odgovarajućeg foldera sa resursima, ako instanca nije postavljena. Ako instanca postoji a razlikuje se od tekuće, objekat igre će biti uništen.

Sem implementacije singltona, ova klasa predstavlja interfejs za korišćenje zvukova iz klase *AudioSound*. Omogućava puštanje i zaustavljanje zvuka, podešavanje jačine zvuka kao i proveru da li se neki zvuk reprodukuje trenutno.

4.2 Objekti igre: obrasci Prototip i Model komponenta - objekat

Objekat igre je jedan od najosnovnijih pojmova kojima Unity barata. Objekti igre u ovom okruženju zasnovani su na obrascu Model komponenta - objekat. Svaki objekat sadrži niz različitih komponenti. Svaka skripta koju napišemo predstavlja ništa drugo do jednu od komponenti objekta. Ovo je izuzetno pogodno za programiranje video igara jer se mnoge komponente ponavljaju, pa je ovim pristupom moguće sklopiti odgovarajući objekat od prethodno napravljenih komponenti.

Glavni akteri igre su napadači i branitelji. Iako suštinski različiti, oni i dalje dele neka svojstva, odnosno komponente. Od gotovih komponenti okruženja Unity, dele komponente *Transform* (za kontrolu pozicije, rotacije i skaliranja), *Animator* (za animaciju objekta), *Rigidbody* i *Collider* (za detekciju kolizije). Od ručno pisanih skripti, i napadači i branitelji sadrže skriptu *Health* koja čuva podatke o inicijalnom i trenutnom zdravlju objekta i uništava objekat kada njegovo zdravlje padne ispod nule.

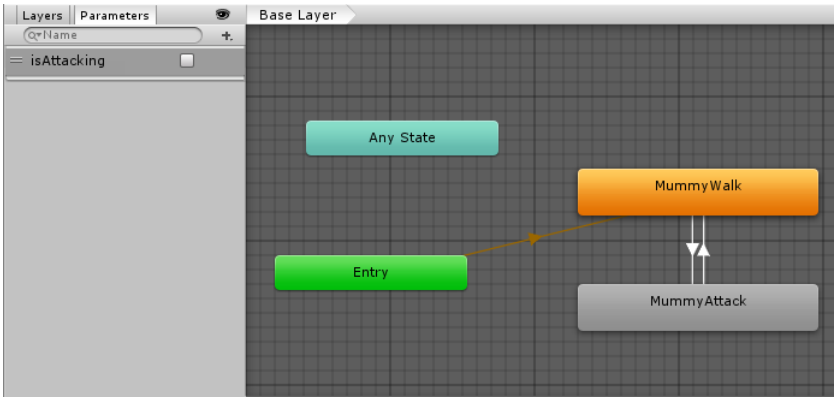
Skripte specifične za napadače jesu *Attacker* i *Mummy*, o kojima će biti više reči u delu Stanje, dok je skripta koja definiše branitelje *Defender*. Glavna promenljiva u toj skripti jeste promenljiva *goldCost* koja je konfigurabilna iz inspektora okruženja Unity i predstavlja cenu kreiranja branitelja. Sadrži i metod *AddGold* koji koristi samo piramida. Tornjevi (običan i ledeni) sadrže još i skriptu *Shooter* koja će biti obrađena malo kasnije.

Još jedan od već implementiranih obrazaca koje Unity sadrži jeste Prototip. Implementiran je kroz sistem prifaba. Jednom napravljen objekat igre, sa svim svojim skriptama, komponentama i podobjektima, može se sačuvati kao prifab i zatim (re)kreirati proizvoljno mnogo puta. Metod *Clone* ovog obrasca u okruženju Unity se naziva *Instantiate* i kao argument prima prifab, odnosno originalni objekat koji treba klonirati. Ovaj obrazac je iskorišćen za sve objekte koji se često kreiraju, a to su svi napadači, branitelji i projektili koje tornjevi ispaljuju na napadače.

4.3 Animatori i obrazac Stanje

Jedna od bitnih stvari svake video igre jeste animacija objekata. Unity sadrži podršku za animacije a njenu osnovu čine *Animator Controller* i *Animation*. Ceo sistem je zasnovan na konačnim automatima i obrascu za projektovanje Stanje. *Animator Controller* se sastoji iz stanja, prelaza i parametara. *Animation* predstavlja jedno stanje u animator kontroleru i predstavlja jednu animaciju. Prelazi iz jednog u drugo stanje često zavise od stanja parametara.

Ako uzmemo primer napadača, odnosno mumije, njen animator kontroler izgleda ovako:

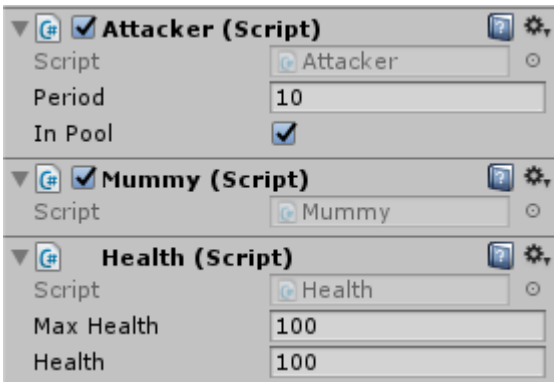


Slika 4.1: Animator kontroler napadača

Kao što vidimo, osim podrazumevanih stanja *Entry* i *Any State*, mumija sadrži stanja *MummyWalk* i *MummyAttack*, kao i parametar *isAttacking*. Vrednost parametra isAttacking se postavlja iz kôda i u zavisnosti od njega prelazi se u stanje *MummyWalk* ili *MummyAttack*. Svako stanje ima svoju animaciju i događaje (*animation event*) koji pokreću odgovarajuće funkcije. U ovom primeru, kada mumija pređe u stanje *MummyAttack*, prestaće da se kreće i krenuće da napada trenutnu metu postepeno joj smanjujući vrednost promenljive *health* u skripti *Health* samog branitelja zahvaljujući događaju u animaciji koji poziva metod *StrikeCurrentTarget*. Kada se vrednost promenljive *isAttacking* vrati na false, promeniće se stanje u *MummyWalk* i mumija će ponovo početi da se kreće. Skripte koje kontrolišu ponašanje mumije jesu *Attacker* i *Mummy*.

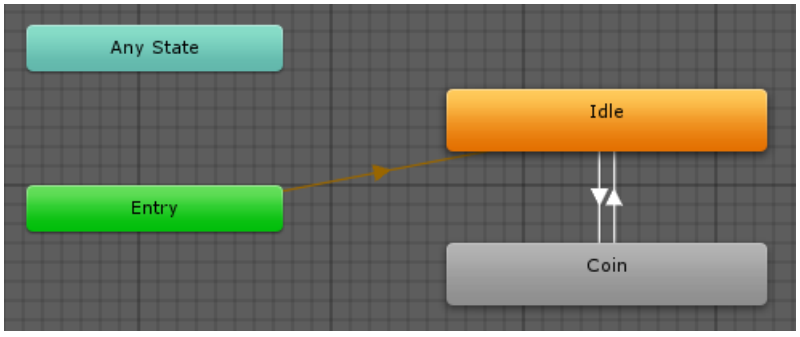
Skripta *Attacker* sadrži promenljive *period*, *currentSpeed* i *inPool*. Početne vred-

nosti su konfigurabilne iz inspektora Unity-ja. *Period* utiče na brzinu kreiranja napadača od strane *Spawner* objekata, *currentSpeed* kontroliše brzinu kretanja napadača, dok *inPool* označava da li je objekat trenutno aktivan (detalji će biti obrađeni u delu o klasi *ObjectsPool*). Takođe detektuje koliziju sa braniteljem i indirektno menja stanje mumije promenom vrednosti promenljive *isAttacking*. Skripta *Attacker* je odgovorna za kretanje napadača, biranje mete napada, kontrolu brzine napadača, kao i za sam čin napadanja.



Slika 4.2: Primer konfiguracije napadača

Što se tiče branitelja, njihovo ponašanje je takođe modelovano stanjima. Ali najpre par reči o ekonomiji igre. Resurs koji se koristi jesu zlatnici. Igrač kreće sa nekom početnom sumom, ali za pobedu je svakako neophodno generisati još resursa. Generatori resursa u igri jesu piramide.



Slika 4.3: Animator kontroler piramide

Animator kontroler piramide sadrži dva stanja, *Idle*, odnosno mirovanje, i *Coin*, odnosno proizvodnja zlatnika. Njen animator kontroler ne sadrži promenljive, već

su prelazi definisani vremenom. Posle određenog vremena, prelazi se u stanje koje proizvodi novčić i koje poziva metod *AddGold*, a čim se završi animacija vraća se u stanje mirovanja. Glavna skripta za ekonomiju igre je GoldDisplay. Odgovorna je za dodavanje i trošenje zlata, kao i za ažuriranje tekstualnog polja za prikaz tekućeg stanja zlata.

Tornjevi imaju sličnu logiku kao i napadači - dva stanja, mirovanje i napad, i boolean promenljiva *isAttacking*. Prelazi su definisani u zavisnosti od promenljive. Kada se detektuje napadač u liniji, promenljiva se postavlja na true i toranj prelazi u stanje napada koje poziva metod *Fire* za ispaljivanje projektila. Kada je neprijatelj uništen i nema novih neprijatelja u istoj liniji, promenljiva se vraća na false i toranj prelazi u stanje mirovanja.

Korišćenje okruženja Unity nam dosta olakšava kontrolu ponašanja objekata i zahvaljujući implementaciji obrasca Stanje omogućava nam da kreiramo objekte koji će sami znati kako da se ponašaju, odnosno kreira neki vid veštačke inteligencije. Umesto eksplicitnog pozivanja metoda za promenu stanja, to se vrši automatski konfigurisanjem prelaza između stanja, najčešće na osnovu promenljivih ili trigera.

4.4 Projektili i obrazac Komanda

Vratimo se na tornjeve. Skripte koje imaju tornjevi su ranije opisane skripte *Health* i *Defender* kao i skripta *Shooter*, koja je zadužena za ispaljivanje projektila i inicijalna podešavanja na početku (nameštanje odgovarajuće linije u kojoj se toranj nalazi). Kada se detektuje napadač u istoj liniji odbrane gde je i toranj, menja se vrednost promenljive iz animatora koja prebacuje stanje iz mirovanja u napad (i obrnuto). Postoje dve vrste tornjeva - običan i ledeni. Razlika je u tipu projektila koji ispaljuju (i u ceni samog tornja).

Projektili su objekti kreirani od strane tornjeva koji uništavaju napadače. Svaki toranj ima svoju vrstu projektila - običan ima energetsku loptu, a ledeni ledenu loptu. Energetska samo oštećuje napadača, dok ga ledena uz to i usporava. Iako za sada postoje samo dve vrste projektila, moguće je obogatiti igru raznim vrstama tornjeva i projektila koji bi radili najrazličitije stvari. Još neke funkcionalnosti koje bi projektili mogli da imaju jesu potpuno zaustavljanje napadača određeni broj se-

kundi, eksplozija koja bi zahvatila i obližnje napadače a ne samo metu, prolazak kroz više neprijatelja itd. Sem ovih, sasvim je moguće da želimo da napravimo projektil koji oštećuje napadača, prolazi kroz više neprijatelja i istovremeno usporava sve napadače. Ako bi svaki projektil pravili nezavisno od drugih, pravljenje ovakvih projektila bi bilo izuzetno naporno i bilo bi puno ponavljanja kôda. Zato je za ovaj problem dobro iskoristiti obrazac za projektovanje Komanda.

Ponašanje ovih objekata je definisano skriptom *Projectile*. Zasnovana je na apstraktnoj klasi *ProjectileCommand* sa apstraktnim metodom *Execute*. Konkretne komande nasleđuju apstraktnu klasu i sadrže svoju implementaciju metoda *Execute*. Komande koje su do sada implementirane su *DealDamage* i *Freeze*. Klasa *ProjectileCommand* izgleda ovako:

```
1  using UnityEngine;
2
3  public abstract class ProjectileCommand
4  {
5      public enum Status { DONE, NOT_READY }
6      public abstract Status Execute(Collider2D collider, float
       damage);
7  }
8
9  public class DealDamage : ProjectileCommand
10 {
11     public override Status Execute(Collider2D collider, float
       damage)
12     {
13         Attacker attacker = collider.gameObject.GetComponent<
       Attacker>();
14         Health health = collider.gameObject.GetComponent<Health>();
15
16         if (attacker && health)
17         {
18             health.DealDamage(damage);
19             return Status.DONE;
20         }
21         else
22         {
23             return Status.NOT_READY;
24         }
```

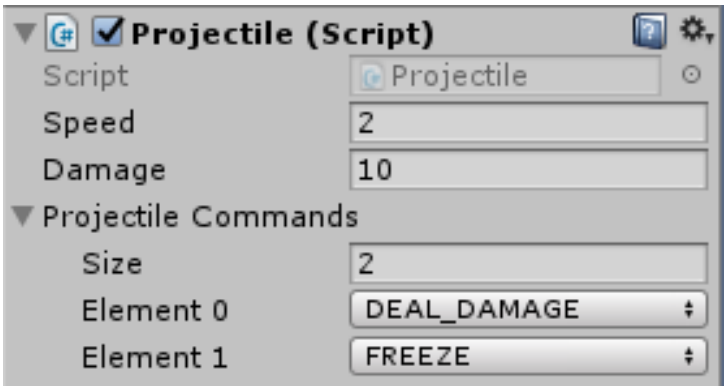
```
25      }
26  }
27
28  public class Freeze : ProjectileCommand
29  {
30      public override Status Execute(Collider2D collider, float
        damage)
31      {
32          Attacker attacker = collider.gameObject.GetComponent<
        Attacker>();
33          if (attacker)
34          {
35              if (attacker.GetSpeed() > 0)
36              {
37                  attacker.SetSpeed(0.1f);
38              }
39              return Status.DONE;
40          }
41          else
42          {
43              return Status.NOT_READY;
44          }
45      }
46  }
```

Što se tiče klase *Projectile*, ona sadrži listu komandi koje treba da izvrši, kao i rečnik (ključ enum, vrednost komanda) svih komandi. Odgovorna je za kretanje projektila, kao i za izvršavanje svih komandi koje taj projektil treba da odradi kada se susretne sa neprijateljom. Zahvaljujući ovom obrascu, zadavanje komandi koje projektil izvršava može se vrlo jednostavno uraditi iz inspektora okruženja Unity, pa se lako mogu kreirati nove vrste projektila uz fleksibilnost dodavanja kombinacije stvari koje taj projektil treba da uradi. Sama klasa izgleda ovako:

```
1  using System.Collections.Generic;
2  using UnityEngine;
3
4  public class Projectile : MonoBehaviour {
5
6      public float speed, damage;
7      public enum ProjectileCommandEnum {DEAL_DAMAGE, FREEZE}
8      public List<ProjectileCommandEnum> projectileCommands;
```

```
9       private Dictionary<ProjectileCommandEnum, ProjectileCommand>
    dict;
10
11      void Start () {
12          dict = new Dictionary<ProjectileCommandEnum,
    ProjectileCommand>()
13          {
14              { ProjectileCommandEnum.DEAL_DAMAGE, new DealDamage()},
15              { ProjectileCommandEnum.FREEZE, new Freeze() }
16          };
17      }
18
19      void Update ()
20      {
21          transform.Translate(Vector3.right * speed * Time.deltaTime)
    ;
22  }
23
24      void OnTriggerEnter2D(Collider2D collider)
25      {
26          List<ProjectileCommand.Status> statuses = new List<
    ProjectileCommand.Status>();
27          foreach (ProjectileCommandEnum projectileCommand in
    projectileCommands)
28          {
29              ProjectileCommand.Status status = dict[
    projectileCommand].Execute(collider, damage);
30              statuses.Add(status);
31          }
32          if (!statuses.Contains(ProjectileCommand.Status.NOT_READY))
33          {
34              ObjectsPool.instance.AddObjectToPool(gameObject);
35          }
36      }
37  }
```

Zahvaljujući primeni obrasca Komanda, projektili su potpuno konfigurabilni iz inspektora okruženja Unity. Moguće je podesiti brzinu projektila, količinu štete koju čine kada udare u napadača, kao i listu komandi koje projektil treba da odradi.



Slika 4.4: Zadavanje ponašanja projektila iz inspektora

4.5 Kreiranje i uništavanje objekata i obrazac Kolekcija objekata

U video igrama često postoje objekti koji se kreiraju i uništavaju veoma često. U igri *Mummy invasion* se stalno pojavljuju mumije, često se kreiraju i uništavaju branitelji, kao i projektili koje tornjevi ispaljuju. Kreiranje i uništavanje objekata u okruženju Unity predstavlja dosta skupu operaciju pa bi kreiranje i uništavanje po nekoliko objekata u svakom frejmu drastično uticalo na performanse same igre. Zato uvodimo obrazac za projektovanje Kolekcija objekata (*Objects Pool*).

Za kreiranje i uništavanje objekata u igri *Mummy invasion* zadužena je klasa *ObjectsPool*, zasnovana na istoimenom obrascu za projektovanje. Sadrži dve liste - *itemsToPool*, koja sadrži listu svih tipova objekata koji će se često kreirati i uništavati, i *pooledObjects*, koja sadrži konkretne instance tih objekata. Na početku se lista pooledObjects napuni zadatim brojem objekata svih zadatih tipova. Oni predstavljaju inicijalan broj u rezervoaru. Kada se svi ti objekti iskoriste, kreiraju se nove instance i dodaju u rezervoar. Kada je negde iz kôda potrebno kreirati novi objekat, metod će prvo proći kroz listu objekata koji trenutno postoje i pokušati da nađe aktivan traženi objekat koristeći sistem tagova okruženja Unity. Svaki objekat sadrži tag koji predstavlja opisni string ključnih reči koje su nadovezane jedna na drugu. Na primer, tag može biti *„DefenderBlocker"*. Sistem tagova se koristi na još nekim mestima u kôdu kada je potrebno prepoznati tip objekta. Negde je potrebno naći sve objekte tipa branitelj, a negde konkretnog branitelja, pa ovaj pristup sa nadovezivanjem ključnih reči radi u oba slučaja. Dakle, ako ne postoji aktivan traženi

objekat, onda se kreira nova instanca tog objekta i dodaje u listu *pooledObjects* i vraća pozivaocu. Kada je potrebno uništiti neki objekat, metoda *AddObjectsToPool* deaktivira zadati objekat i premešta ga u rezervoar objekat. Treba napomenuti da ovde nema dodavanja u listu *pooledObjects* jer se objekat već nalazi tamo jer je tu smešten u toku kreiranja. Ovim pristupom se značajno poboljšavaju performanse jer se izbegava često instanciranje i uništavanje objekata, već se objekti samo premeštaju i (de)aktiviraju. Za implemetaciju se može iskoristiti i neka druga struktura podataka, na primer red (*queue*). Sama skripta izgleda ovako:

```
1  using System.Collections.Generic;
2  using UnityEngine;
3
4  [System.Serializable]
5  public class ObjectPoolItem
6  {
7      public int amountToPool;
8      public GameObject objectToPool;
9      public bool shouldExpand = true;
10 }
11
12 public class ObjectsPool : MonoBehaviour {
13
14     public static ObjectsPool instance;
15     public List<GameObject> pooledObjects;
16     public List<ObjectPoolItem> itemsToPool;
17     private DefenderSpawner defenderSpawner;
18
19     void Awake()
20     {
21         instance = this;
22     }
23
24     private void Start()
25     {
26         defenderSpawner = GameObject.FindObjectOfType<
       DefenderSpawner>();
27         pooledObjects = new List<GameObject>();
28         foreach (ObjectPoolItem item in itemsToPool)
29         {
30             for (int i = 0; i < item.amountToPool; i++)
```

```
31              {
32                  GameObject obj = Instantiate(item.objectToPool) as
    GameObject;
33                  obj.transform.parent = gameObject.transform;
34                  obj.transform.position = gameObject.transform.
    position;
35                  obj.SetActive(false);
36                  pooledObjects.Add(obj);
37              }
38          }
39      }
40
41      public GameObject GetPooledObject(string tag)
42      {
43          for (int i=0; i<pooledObjects.Count; i++)
44          {
45              if (!pooledObjects[i].activeInHierarchy &&
    pooledObjects[i].tag.Contains(tag))
46              {
47                  if (pooledObjects[i].tag.Contains("Defender"))
48                  {
49                      pooledObjects[i].GetComponent<Defender>().
    inPool = false;
50                  }
51
52                  return pooledObjects[i];
53              }
54          }
55          foreach (ObjectPoolItem item in itemsToPool)
56          {
57              if (item.objectToPool.tag.Contains(tag))
58              {
59                  if (item.shouldExpand)
60                  {
61                      GameObject obj = Instantiate(item.objectToPool)
     as GameObject;
62                      obj.SetActive(false);
63                      pooledObjects.Add(obj);
64                      if (obj.tag.Contains("Defender"))
65                      {
66                          obj.GetComponent<Defender>().inPool = false
    ;
```

```
67                      }
68                      return obj;
69                  }
70              }
71          }
72          return null;
73      }
74
75      public void AddObjectToPool(GameObject obj)
76      {
77          if (obj.tag.Contains("Defender"))
78          {
79              obj.GetComponent<Defender>().inPool = true;
80              defenderSpawner.matrix[(int)obj.transform.position.x -
    1, (int)obj.transform.position.y - 1] = 0;
81          }
82          obj.gameObject.transform.parent = gameObject.transform;
83          obj.gameObject.transform.position = gameObject.transform.
    position;
84          obj.gameObject.SetActive(false);
85          Health healthComponent = obj.GetComponent<Health>();
86          if (healthComponent != null)
87          {
88              healthComponent.health = healthComponent.maxHealth;
89          }
90      }
91  }
```

4.6 Kontrola toka igre

Još jedna bitna stvar u implementaciji igre jeste kontrola toka. Igra mora da zna kada i kako treba da se završi. Za kontrolu vremenskog toka igre zadužena je klasa *GameTimer*. Ona preračunava dužinu trajanja tekućeg nivoa, pomera slajder i po isteku vremena otključava nivoe. Izgleda ovako:

```
using UnityEngine;
using UnityEngine.UI;

public class GameTimer : MonoBehaviour {

    public float baseLevelSeconds = 50;
    public float levelSeconds = 50;
    private Slider slider;
    private bool isEndOfLevel = false;
    private LevelManager levelManager;
    private GameObject winLabel;

  void Start ()
    {
        slider = GetComponent<Slider>();
        levelManager = GameObject.FindObjectOfType<LevelManager>();
        winLabel = GameObject.Find("YouWonLabel");
        winLabel.SetActive(false);
        levelSeconds = baseLevelSeconds + 10 * MetaData.
    currentLevel;
  }

  void Update ()
    {
        slider.value = Time.timeSinceLevelLoad / levelSeconds;

        if (!isEndOfLevel && Time.timeSinceLevelLoad >=
    levelSeconds)
        {
            winLabel.SetActive(true);
            MusicManager.Instance.PlaySound("quest_complete");
            if (MetaData.currentLevel == MetaData.maxLevel)
            {
                Invoke("GoToWinScene", 3.3f);
            }
            else
            {
                Invoke("GoToMap", 3.3f);
            }
            if (MetaData.currentLevel == MetaData.levelsUnlocked)
            {
```

```
40                  MetaData.levelsUnlocked++;
41                  PlayerPrefsManager.SetLevel(MetaData.levelsUnlocked
    );
42              }
43              isEndOfLevel = true;
44          }
45      }
46
47      private void GoToMap()
48      {
49          MusicManager.Instance.StopSound("PlaySceneMusic");
50          levelManager.LoadLevel("Map");
51      }
52
53      private void GoToWinScene()
54      {
55          MusicManager.Instance.StopSound("PlaySceneMusic");
56          levelManager.LoadLevel("Win");
57      }
58  }
```

Glavni podaci o trenutnom nivou, ukupnom broju pređenih nivoa i težini igre nalaze se u klasi *MetaData*. Igra koristi te podatke da bi odredila trajanje nivoa, a ti podaci se koriste i prilikom kreiranja napadača, jer od njih zavisi frekvencija pojavljivanja mumija. Ova klasa brine o uspešno završetku igre. Međutim, igra se može završiti i neuspešno po igrača, u slučaju da ne uspe da se odbrani i mumije prodru na drugu stranu terena. Taj slučaj je pokriven postojanjem nevidljivog objekta sa skriptom *LoseCollider* u sceni igre sa leve strane, ispred svih polja. On se aktivira kada se neprijatelj sudari sa njim i prebacuje na scenu poraza.

```
using UnityEngine;

public class LoseCollider : MonoBehaviour {

    private LevelManager levelManager;

    void Start()
    {
        levelManager = GameObject.FindObjectOfType<LevelManager>();
    }

    void OnTriggerEnter2D()
    {
        MusicManager.Instance.StopSound("PlaySceneMusic");
        levelManager.LoadLevel("Lose");
    }
}
```

Glava 5

Zaključak

Video igre su postale ozbiljna industrija i njihovo kreiranje je preraslo na ozbiljan nivo. Da bi kreiranje kompleksnih igara bilo moguće, neophodno je korišćenje odgovarajućih okruženja i efikasno i fleksibilno programiranje, što često podrazumeva obrasce za projektovanje. U ovom radu su prikazani neki od najkorišenijih obrazaca za projektovanje koji se koriste u razvoju video igara. U praktičnom delu rada su neki od njih direktno implementirani, a neki korišćenjem okruženja Unity. Prikazane su i neke od ostalih pogodnosti koje samo okruženje pruža i time olakšava i usmerava sam razvoj. Svrha rada je da pokuša da približi značaj organizovanog pristupa i pomogne svaki dalji razvoj. Važno je uočiti česte probleme koji se u praksi javljaju, kao i odabrati pravi metod za rešavanje istih, jer time olakšavamo svaku buduću izmenu, održavanje i unapređivanje igre.

Mogući pravci daljeg rada su dodavanje novih funkcionalnosti u samu igru, kreiranje drugih igara korišćenjem obrazaca za projektovanje tamo gde ih valja iskoristiti, kao i dalje proučavanje drugih obrazaca za projektovanje koji se možda ne primenjuju toliko često, ali se uvek možemo naći u situaciji kada nam baš oni rešavaju problem na koji smo naišli.

Literatura

[1] Penny de Byl. *Holistic game development with Unity.* Elsevier, 2012.

[2] John P. Doran and Matt Casanova. *Game development patterns and best practices.* Packt Publishing, 2017.

[3] Draw.io. Draw.io. https://www.draw.io/.

[4] gameprogrammingpatterns.com. gameprogrammingpatterns.com. https://gameprogrammingpatterns.com/contents.html.

[5] Saša Malkov. Obrasci za projektovanje - odlomak iz knjige u pripremi. http://poincare.matf.bg.ac.rs/ smalkov/.

[6] Mixamo.com. Mixamo.com. https://www.mixamo.com/.

[7] Unity. Unity user manual. https://docs.unity3d.com/Manual/index.html.